Άσκηση 4η για το Σπίτι – Μικροεπεξεργαστές - Verilog στο Modelsim και τα Δομικά στοιχεία υλικού του MicroCPU

Στην εργασία θα ακολουθήσετε tutorial για εγκατάσταση Modelsim και σχεδιασμό με Verilog και μετά θα κάνετε τις παρακάτω ασκήσεις. Επίσης, θα μελετήσουμε και θα επιβεβαιώσουμε τον σχεδιασμό τμημάτων υλικού ενός μικροεπεξεργαστή που ονομάζεται MicroCPU (MCPU) στο Modelsim.

Άσκηση 4.1

Α) Υλοποιήστε ένα module ίδιο με το example1 από το tutorial, αλλά να αλλάξετε το όνομα του. ΔΕΝ θα είναι «example1», αλλά «d1s#AM», όπου #AM ο αριθμός μητρώου σας. Ομοίως, τα αρχεία πρέπει να λέγονται «d1s#AM.v» και «d1s#AMtb.v», όπου #AM ο αριθμός μητρώου σας. Παρομοίως, το όνομα του project σας θα είναι «p1s#AM».
π.χ. αν ο αριθμός μητρώου σας είναι 1234, τότε το όνομα του project θα είναι «p1s1234», το όνομα του module θα είναι «d1s1234» και τα ονόματα των αρχείων «d1s1234.v» και «d1s1234tb.v»

Παραδοτέα άσκησης 4.1.(Α):

- Τα δύο αρχεία Verilog.
- Ένα printscreen με το παράθυρο Wave μόλις εκτελείται η προσομοίωση, στο οποίο να φροντίσετε να φαίνονται τα σωστά ονόματα των σημάτων.

Β) Θα φτιάξετε ένα νέο αρχείο testbench μέσα στο project που έχετε ήδη φτιάξει από την άσκηση 4.1.(Α), το οποίο θα το ονομάσετε «d1s#AMtb2.v»
Δηλαδή «d1s1234tb2.v», αν ο αριθμός μητρώου σας είναι 1234.
Στο νέο αυτό testbench θα αντιγράψετε το προηγούμενο testbench και θα το τροποποιήσετε κατάλληλα, ώστε να προστεθεί ένας νέος καταχωρητής με όνομα d_correct. Ο νέος αυτός καταχωρητής θα ανανεώνεται κάθε 2ps και θα παίρνει την τιμή λογικό-‘1’ όταν η έξοδος του κυκλώματος dut θα είναι σωστή, αλλιώς θα παίρνει την τιμή λογικό-‘0’.

Παραδοτέα άσκησης 4.1.(Β):

- Το καινούριο αρχείο Verilog με το νέο testbench.
- Ένα printscreen με το παράθυρο Wave μόλις εκτελείται η προσομοίωση στο οποίο να φροντίσετε να φαίνονται τα σωστά ονόματα των σημάτων και να φαίνεται και ο νέος καταχωρητής που θα δείχνει ότι το κύκλωμα λειτουργεί επιτυχώς.

Συνολικά παραδοτέα 4.1 άσκησης: 3 αρχεία Verilog και 2 εικόνες.

Άσκηση 4.2: εξοικείωση με την ALU του MCPU

Στο αρχείο *alutb.v* θα βρείτε ένα testbench της ALU που μεταγλωττίζεται στο module MCPU_Alutb, το οποίο δοκιμάζει τυχαίες τιμές και εντολές στις εισόδους της ALU.
Α) Προσθέστε στο testbench ένα always block το οποίο θα ελέγχει τις αποκρίσεις της ALU και θα ενημερώνει έναν καταχωρητή *iscorrect* για το αν ο υπολογισμός των εξόδων της ALU με βάση τις τυχαίες εισόδους που δοκιμάζονται, έγινε σωστά.
Β) Επίσης, σε δεύτερο χρόνο, θέλω να τροποποιήσετε το testbench που μόλις φτιάξατε ώστε να φτιάχνει ένα instance της ALU με καταχωρητές r1,r2, των 8 bits. Στους νέους αυτούς καταχωρητές θα βάλετε ως είσοδο τον ΑΜ σας όπως φαίνεται παρακάτω:

```
always
begin
#3 r1 = 1; #3 r1 = 2; #3 r1 = 3; #3 r1 = 4;
end
always
begin
#3 r2 = 1; #3 r2 = 2; #3 r2 = 3; #3 r2 = 4;
end
```

όπου 1234 ο αριθμός μητρώου σας.
Θα το αφήσετε να εκτελέσει τις τυχαίες πράξεις με το ΑΜ σας στην είσοδο που θα αλλάζει ψηφίο κάθε 2 ps και θα πάρετε εικόνα από τις κυμματομορφές στην οποία θα φαίνεται ο ΑΜ σας μέσα στις κυμματομορφές – φροντίστε να μορφοποιήσετε τις κυμματομορφές των καταχωρητών σε integer (ή decimal).
Χωρίς το (Β) ερώτημα δεν θα γίνεται δεκτό και το (Α).
Παραδοτέα: 2 κώδικες testbench (Α,Β) και 2 εικόνες κυμματομορφών από το modelsim (waveform).

Άσκηση 4.3: εξοικείωση με την μνήμη του MCPU

Δεν υπάρχει testbench της μνήμης στα αρχεία που έχετε.
Α) Γράψτε ένα testbench για την μνήμη το οποίο θα επιβεβαιώνει την σωστή λειτουργία των τριών διαδικασιών της μνήμης (εγγραφή δεδομένων, ανάγνωση δεδομένων και ανάγνωση εντολών).
Οδηγίες: Αρχικά θα χρησιμοποιήσετε την εγγραφή δεδομένων για να γεμίσετε με τυχαίες λέξεις την μνήμη. Τις τυχαίες λέξεις που γράφετε θα τις αποθηκεύσετε και σε ένα τοπικό αντίγραφο της μνήμης (αντίγραφο του reg [WORD_SIZE-1:0] mem[RAM_SIZE-1:0]) μέσα στο testbench. Ο λόγος που θα κρατήσετε το αντίγραφο είναι για να μπορέσετε να ελέγξετε και τις διαδικασίες ανάγνωσης έπειτα. Έτσι, για να ελέγξετε τις διαδικασίες ανάγνωσης θα διαβάσετε, τόσο με την διαδικασία ανάγνωσης δεδομένων όσο και με της ανάγνωσης εντολών, τις λέξεις που έχετε γράψει στην μνήμη και θα τις αντιπαραβάλετε με το τοπικό αντίγραφο. Περιμένουμε να ταυτίζονται.
Β) Επαναλάβετε το Α, αλλά αυτή την φορά αντί να γράφετε τυχαίες λέξεις στην μνήμη, να γράψετε και να διαβάσετε τον ΑΜ σας σε κάθε λέξη της μνήμης.
Χωρίς το (Β) ερώτημα δεν θα γίνεται δεκτό και το (Α).
Παραδοτέα: 2 κώδικες testbench και δύο εικόνες κυμματομορφής από το modelsim (waveform).

Άσκηση 4.4: εξοικείωση με το αρχείο καταχωρητών
Το MCPU έχει μόνο 4 καταχωρητές.
Α) Τροποποιείστε τον κώδικα ώστε τελικά να υποστηρίζει 16 καταχωρητές. Για αυτό το ερώτημα δεν χρειάζεται να προσθέσετε ούτε μία νέα γραμμή κώδικα. Αρκεί αν τροποποιήσετε κατάλληλα τις παραμέτρους. Επιβεβαιώστε έπειτα με την δημιουργία ενός **προγράμματος/benchmark** που θα εκτελεί εντολές από το σύνολο εντολών του MCPU ότι οι καταχωρητές είναι λειτουργικοί όπως πρέπει για όλες τις εντολές.
Β) Φροντίστε να βάλετε τον ΑΜ σας ώστε να φαίνεται στις κυμματομορφές. Αυτό μπορείτε να το κάνετε ως εξής: χωρίστε τον ΑΜ σας σε δύο νούμερα στο 12 και στο 34 (αν ο ΑΜ σας είναι 1234) και τοποθετήστε το πρώτο στην θέση μνήμης 100 και το δεύτερο στην θέση μνήμης 101. Αυτό θα το κάνετε με εντολές που μεταφέρουν σταθερές προς τους καταχωρητές (SHORT_TO_REG) και έπειτα με εντολές που μεταφέρουν από τους καταχωρητές προς την μνήμη (STORE_TO_MEM). Έπειτα μεταφέρετέ τα περιεχόμενα των θέσεων μνήμης 100 και 101 από τη μνήμη προς τους καταχωρητές κι δώστε τα ως είσοδο σε στις εντολές ADD και XOR. Θέλω μέσα στις κυμματομορφές να φαίνεται ο ΑΜ σας, φροντίστε για την κατάλληλη μορφοποίηση. Δεν χρειάζεται να τεστάρετε όλους τους νέους καταχωρητές

Ακολουθεί tutorial για το modelsim και η περιγραφή του MicroCPU. Επίσης, από εκεί που κατεβάσατε την άσκηση θα χρειαστείτε και τα περιεχόμενα του αρχείου microcpu-rtl.zip.

Περιεχόμενα

Tutorial Modelsim

Tutorial Modelsim

Πως θα προμηθευτούμε το λογισμικό

Θα ασχοληθούμε με το Modelsim στις επόμενες ασκήσεις.

Το Modelsim για τις ασκήσεις Verilog που θα ακολουθήσουν είναι μέρος του Quartus.

Το Quartus μπορείτε να το κατεβάσετε απευθείας από των Altera/Intel εδώ https://fpgasoftware.intel.com/13.0sp1/?edition=web

Πρέπει να εγγραφείτε για να το κατεβάσετε.

Προσοχή θέλουμε μόνο την έκδοση 13.0sp1.

Το λογισμικό τρέχει σε windows και linux.

ΕΝΝΑΛΑΚΤΙΚΑ - **το έχω ήδη κατεβάσει** και θα το βρείτε στα παρακάτω links:

Για Windows:

http://vcas.cs.uoi.gr/wiki/labsoftware/Quartus-web-13.0.1.232-windows.tar

Για Linux:

http://vcas.cs.uoi.gr/wiki//Quartus-web-13.0.1.232-linux.tar

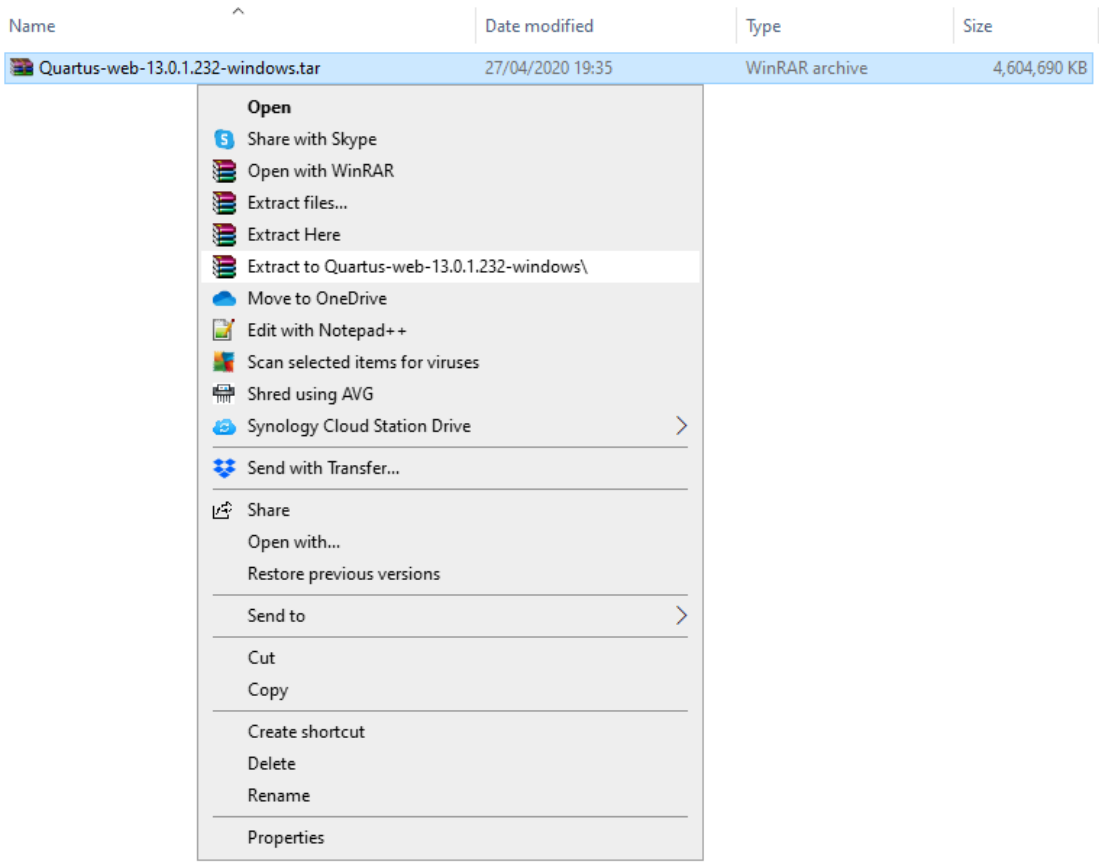
Παρακαλώ προχωρήστε στην εγκατάσταση για να είναι έτοιμο για το μάθημα, επειδή είναι μεγάλο το αρχείο.

Το λογισμικό υπάρχει ήδη στα εργαστήρια εγκατεστημένο, επομένως αν μπορείτε να συνδεθείτε απομακρυσμένα τότε μπορείτε να κάνετε και τις εργασίας με απομακρυσμένη σύνδεση, αν δεν θέλετε να το εγκαταστήσετε.

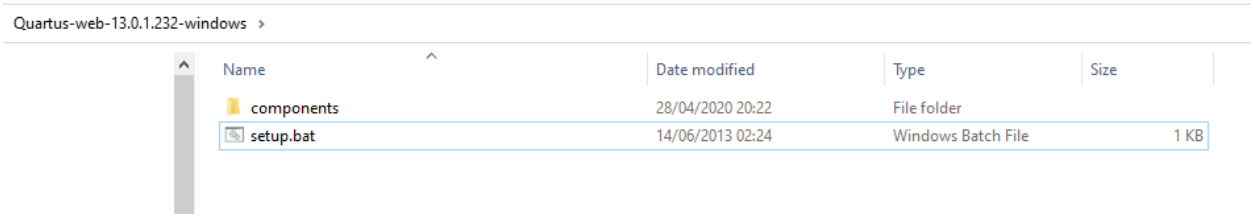
Ακολουθεί ένα βασικό tutorial για να ξεκινήσετε με το Modelsim.

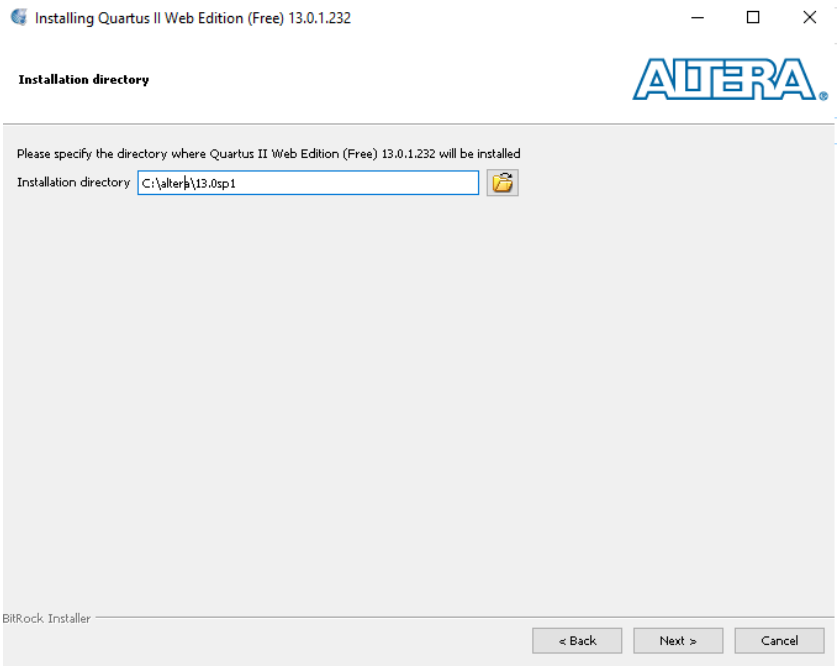
Εγκατάσταση Quartus

Κατεβάζουμε το Quartus και αποσυμπιέζουμε το αρχείο που κατεβάσαμε. Μπορείτε να το κάνετε με το winrar, όπως φαίνεται στην παρακάτω εικόνα:



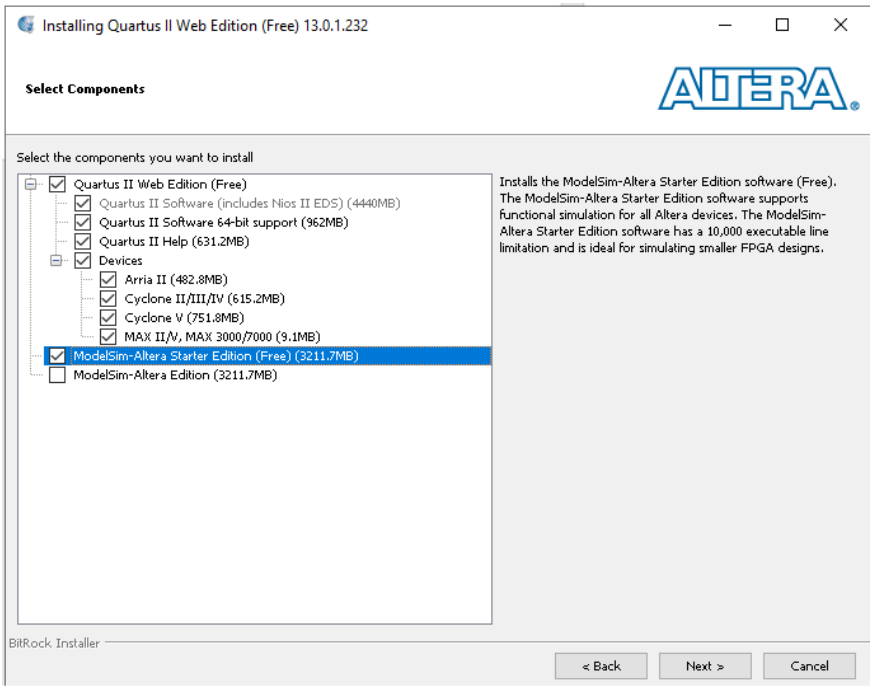
Έπειτα **εκτελούμε** το αρχείο setup.bat, όπως φαίνεται παρακάτω:

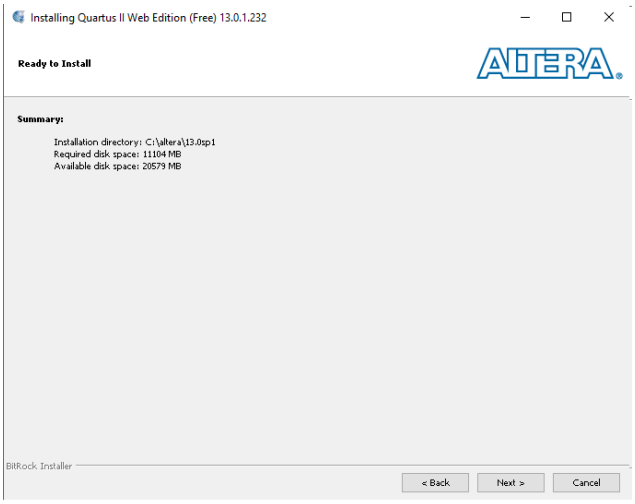




Επιλέγουμε που θα γίνει η εγκατάσταση και προχωράμε στην επιλογή πακέτων.

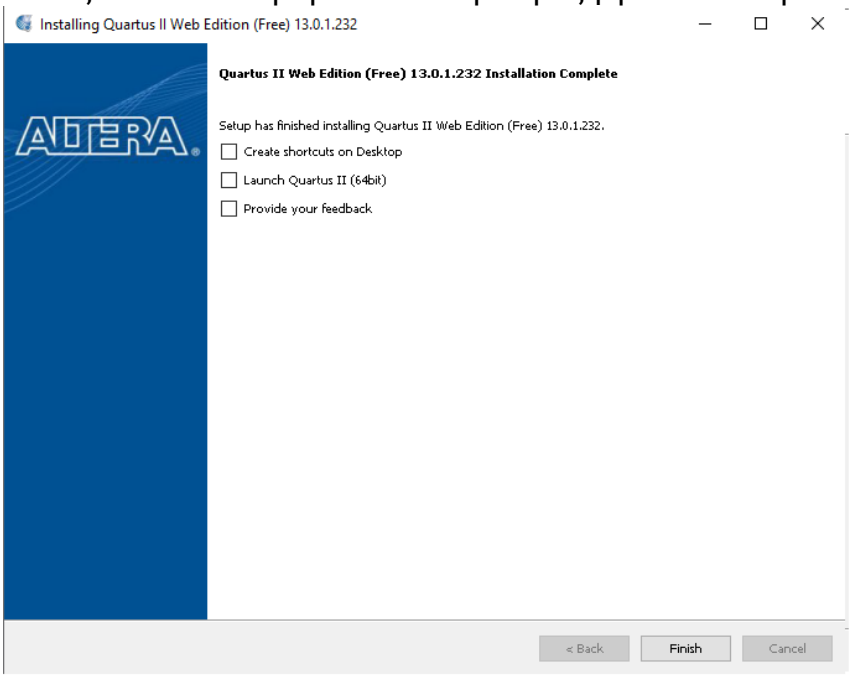
Θέλουμε **το Modelsim Altera Started Edition (Free),** που είναι προεπιλεγμένο. Ωστόσο βάλτε και το Quartus που είναι και αυτό προεπιλεγμένο. ΜΗΝ επιλέξετε την άλλη έκδοση modelsim που έχει κάτω-κάτω γιατί δεν είναι free και θα σας ζητάει άδειες.





Ξεκινάμε την εγκατάσταση με τις επιλογές μας. Θα πάρει αρκετή ώρα, επομένως περιμένουμε.

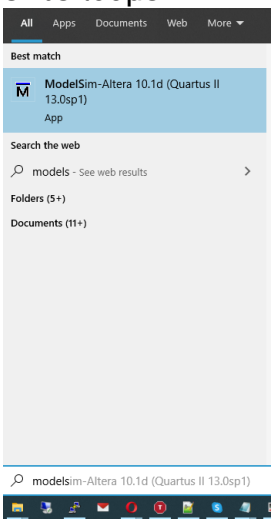
Μόλις τελειώσει η εγκατάσταση θα μας βγάλει το παρακάτω παράθυρο:



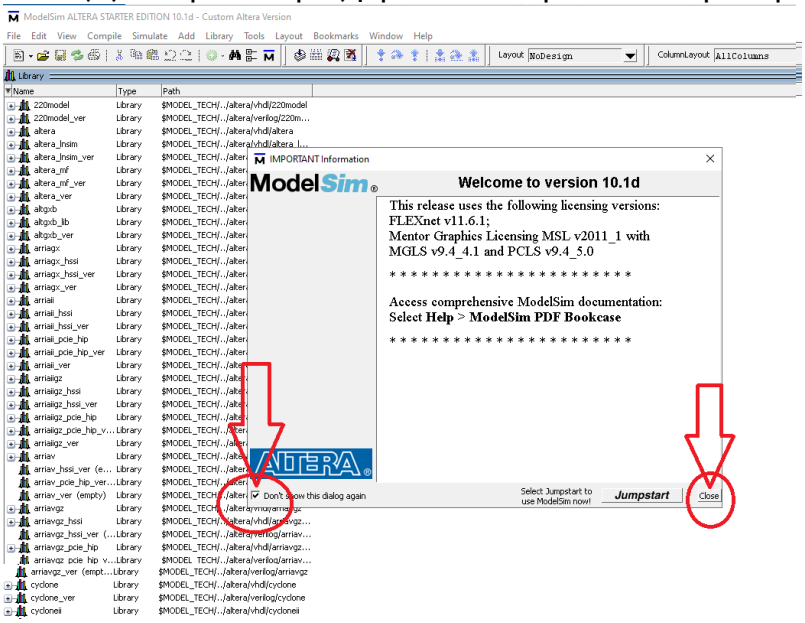
Απενεργοποιήστε τις επιλογές. Δηλαδή δεν θέλουμε μην φτιάξει shortcut ούτε θέλουμε να εκτελέσει το Quartus ούτε θέλουμε να δώσουμε feedback στην εταιρία. Αφού απενεργοποιήσουμε τις επιλογές πατάμε finish.
Συγχαρητήρια, η εγκατάσταση ολοκληρώθηκε και θα προχωρήσουμε στην εκτέλεση του προσομοιωτή Modelsim.

Εκτέλεση του Modelsim

Από το start βρίσκουμε το “Modelsim-Altera 10.1d (Quartus II 13.0sp1)” και το εκτελούμε.

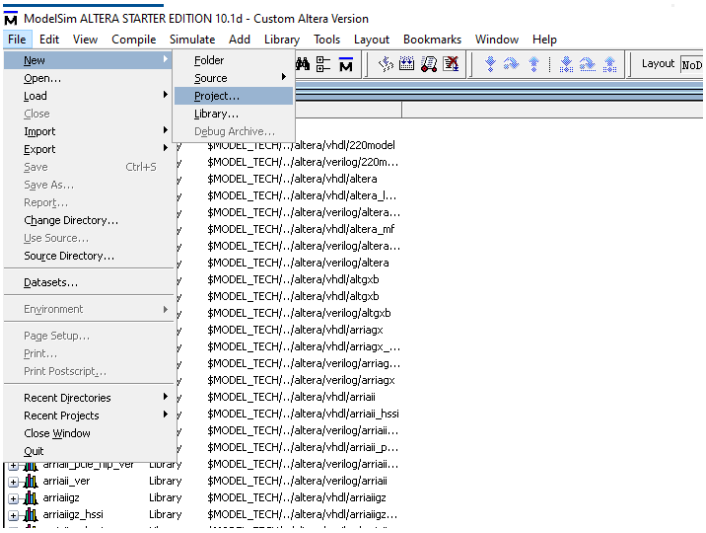


Θα πάρει λίγη ώρα να ξεκινήσει.
Μόλις ξεκινήσει θα μας βγάλει το παρακάτω παράθυρο:

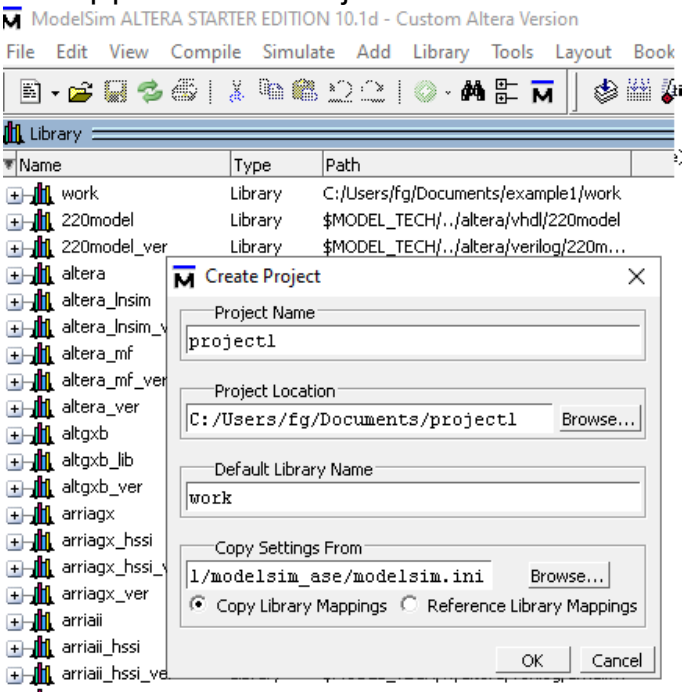


Επιλέγουμε “Don’t show this dialog again” και μετά πατάμε την επιλογή “close”.

Παράδειγμα Νέου Project στο Modelsim



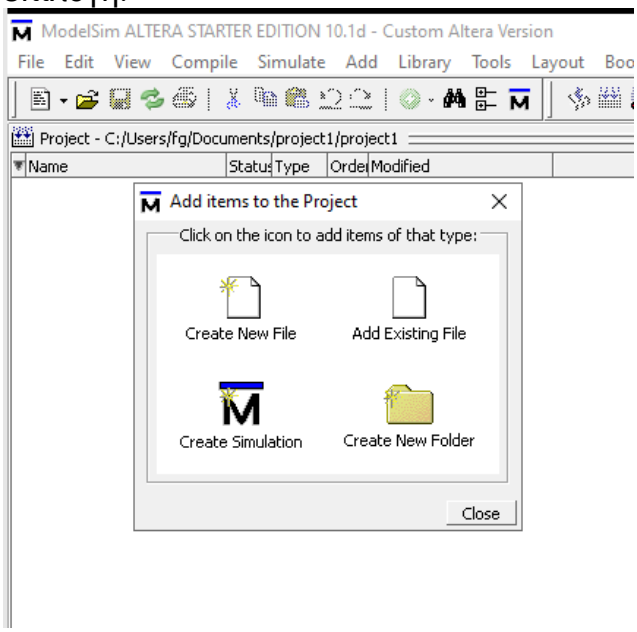
Όπως φαίνεται και από την παραπάνω εικόνα, φτιάχνουμε ένα νέο project με την επιλογή File->New->Project.



Έπειτα επιλέγουμε όνομα για το project και κατάλογο στον οποίο θα το αποθηκεύσουμε. Στο παράδειγμα αυτό έχω δώσει το όνομα project1 στον κατάλογο project1.

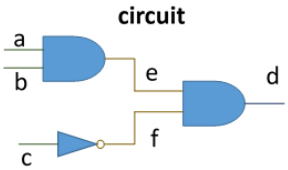
ΠΡΟΣΟΧΗ: το default library name πρέπει να είναι “work”.
Κάθε άσκηση που θα λύνουμε θα έχει το δικό της project.

Στο επόμενο στάδιο πρέπει να επιλέξουμε αρχεία για το project μας με την παρακάτω επιλογή:

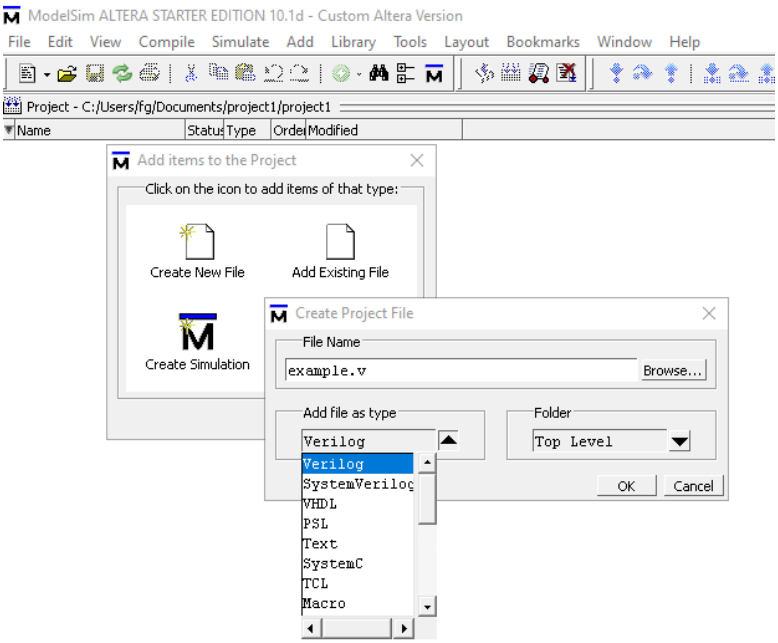


Μπορείτε είτε να φτιάξετε ένα νέο αρχείο είτε να βάλετε αρχεία που υπάρχουν ήδη. Τα αρχεία που θα δίνουμε θα είναι της μορφής Verilog και θα έχουν κατάληξη ".v".

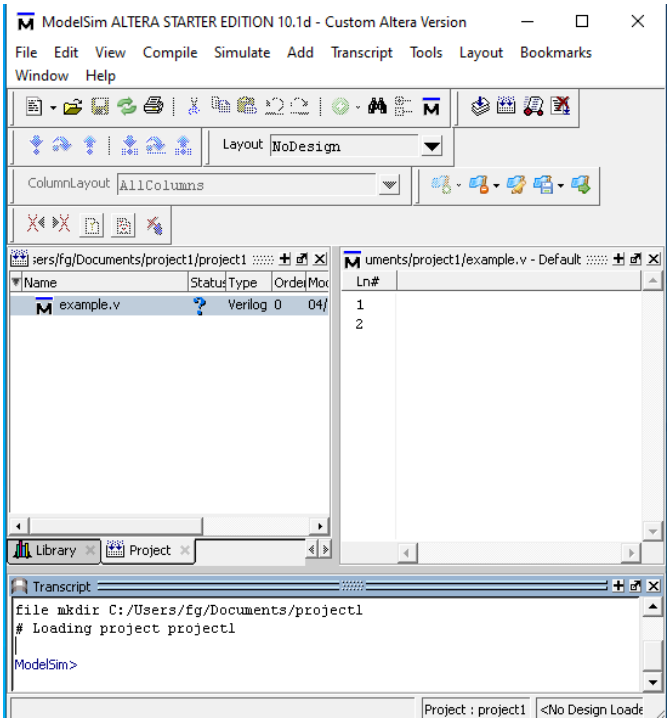
Στο project που φτιάξαμε θα σχεδιάσουμε με Verilog το γνωστό μας πια κύκλωμα:



Αρχικά, θα φτιάξουμε ένα νέο αρχείο στο project μας και θα το ονομάσουμε *example.v*.

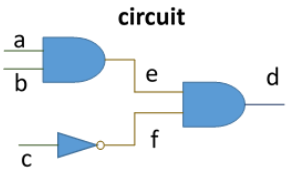


Προσέξτε ότι πρέπει να πείτε στο Modelsim ότι το αρχείο είναι Verilog αρχείο επιλέγοντας το filetype.

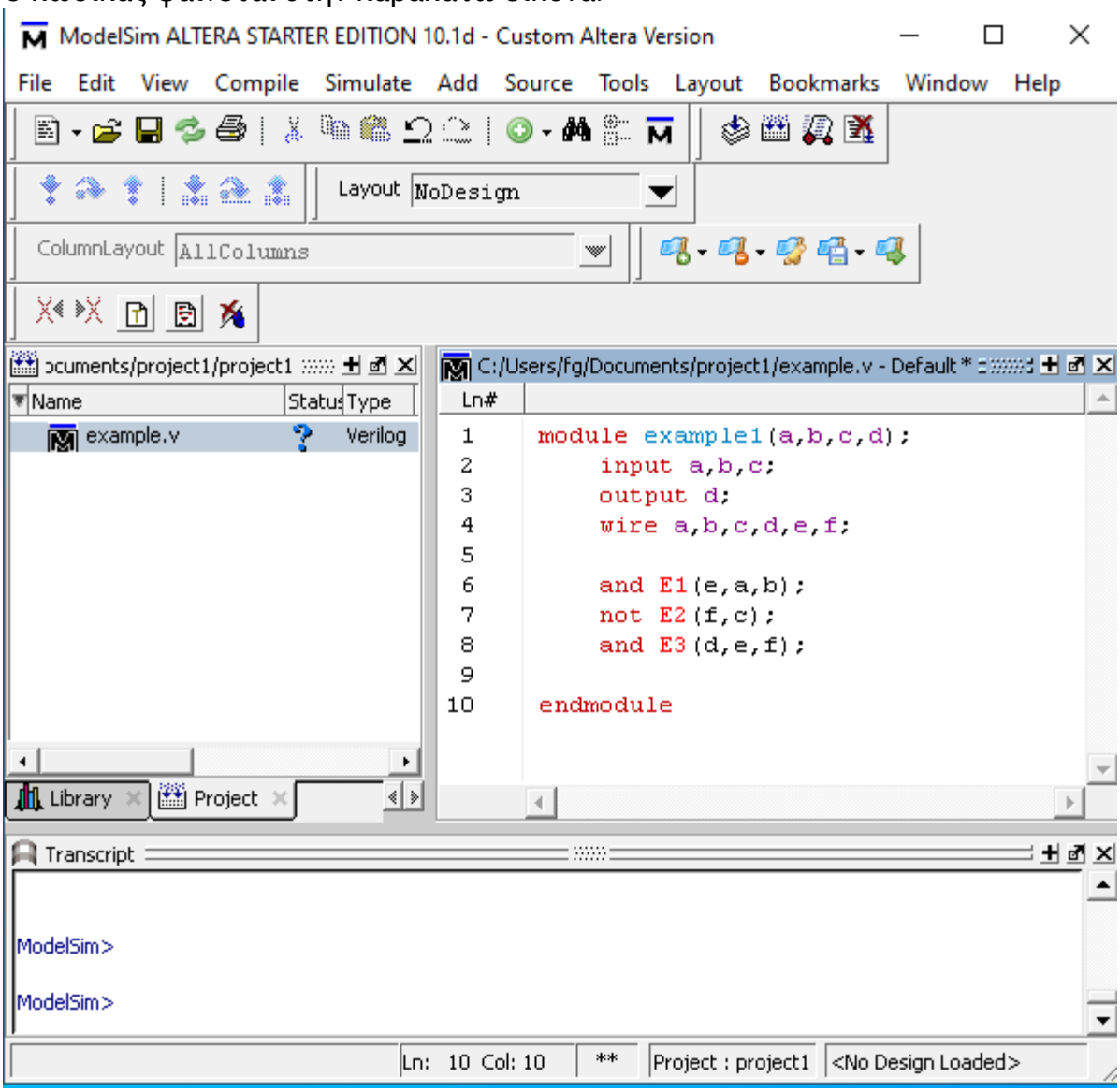


Στην παραπάνω εικόνα φαίνεται το project αριστερά και δεξιά τα περιεχόμενα του αρχείου example.v, το οποίο είναι άδειο. Αν δεν φαίνονται τα περιεχόμενα του αρχείου, κάντε διπλό κλικ πάνω στο αρχείο από το παράθυρο του project και θα το ανοίξει. Πρέπει να είναι άδειο, γιατί μόλις το φτιάξαμε.

Μέσα στο αρχείο example.v γράφουμε κώδικα Verilog με το γνωστό μας κύκλωμα/παράδειγμα:

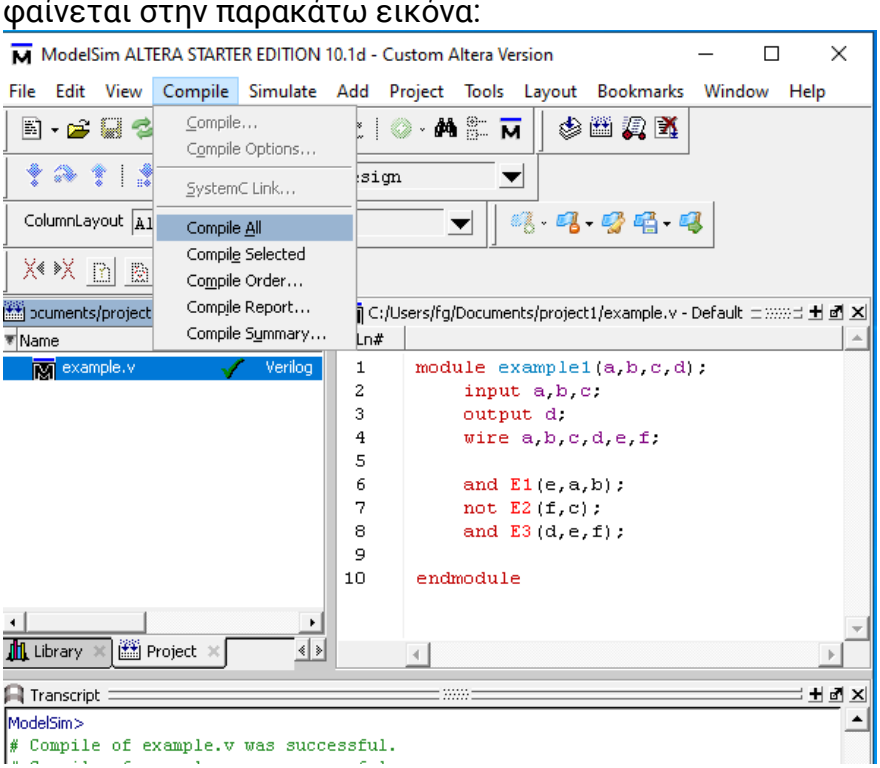


Ο κώδικας φαίνεται στην παρακάτω εικόνα:



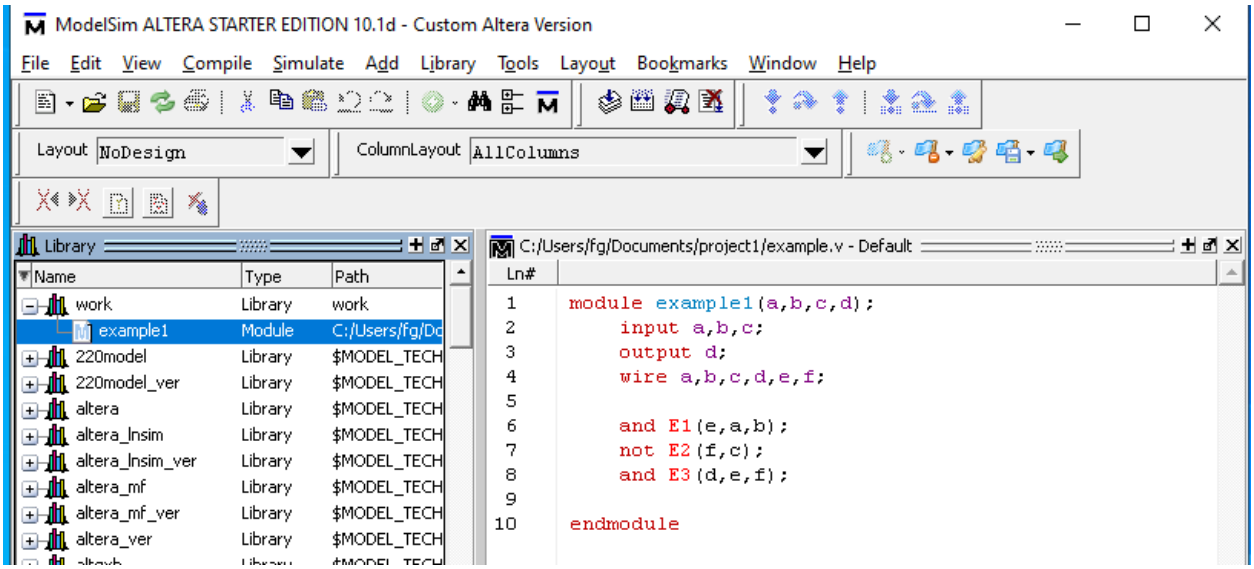
Παράδειγμα Compilation

Επόμενο βήμα είναι να δούμε αν έχουμε κάνει κάποιο λάθος στον κώδικα κάνοντας compilation, εκτελώντας την εντολή Compile->”compile all”, από το menu, όπως φαίνεται στην παρακάτω εικόνα:



Αν όλα είναι καλά θα πάρουμε μήνυμα επιτυχούς ολοκλήρωσης της διαδικασίας και θα υπάρχει ένα check με πράσινο χρώμα δίπλα από το αρχείο μας στο υποπαράθυρο του project. Αν όχι τότε θα πάρουμε μηνύματα λαθών στο παράθυρο της κονσόλας (Transcript) και με διπλό κλικ πάνω τους μπορούμε να δούμε παραπάνω πληροφορίες που θα μας βοηθήσουν να τα κατανοήσουμε και να τα διορθώσουμε.

Το module που μόλις σχεδιάσαμε σε Verilog έχει τώρα δημιουργηθεί και έχει τοποθετηθεί μέσα στην βιβλιοθήκη work, όπως φαίνεται στην παρακάτω εικόνα:



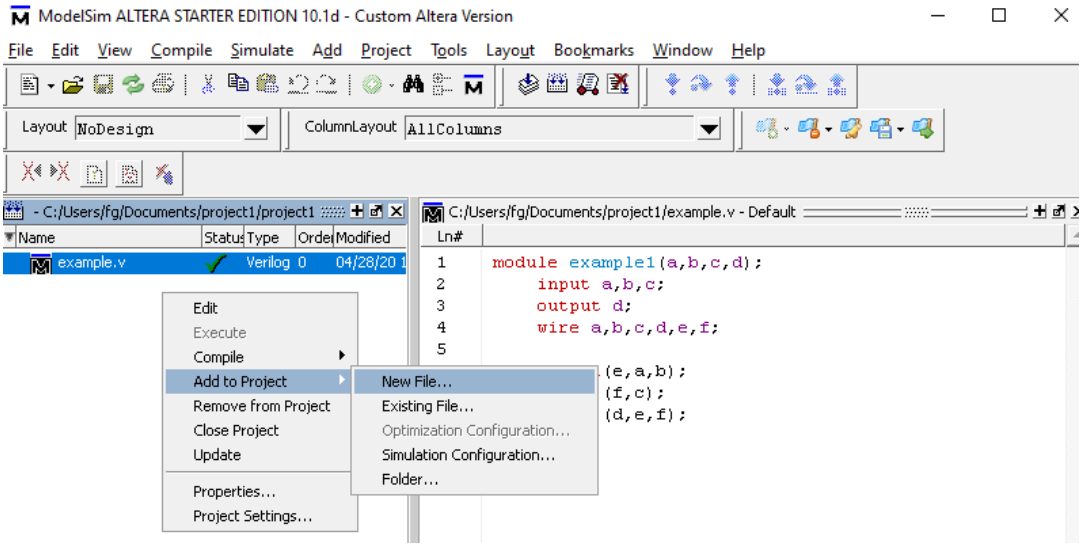
Κονσόλα του Modelsim

Στο κάτω-κάτω παράθυρο φαίνεται η κονσόλα του Modelsim με την προτροπή/prompt:
Modelsim>
Modelsim>
Εκεί θα τυπώνονται μηνύματα από το Modelsim. Επίσης με την κονσόλα μπορούμε να δώσουμε και εντολές. Μάλιστα, οτιδήποτε μπορούμε να κάνουμε με το γραφικό περιβάλλον μέσα στο Modelsim, μπορούμε να το κάνουμε και με την κονσόλα αυτή. Γράψτε για παράδειγμα
Modelsim> help commands
Και θα πάρετε μια λίστα με τις εντολές αυτής τη κονσόλας.
Μπορείτε επίσης να γράψετε
Modelsim> help write
Και να δείτε τι κάνει η write και ποια είναι η σύνταξή της.
Επίσης όταν γράφετε μια εντολή και πατάτε tab, τότε βγάζει σε παραθυράκι (κάτω από το modelsim) πληροφορίες για την εντολή.

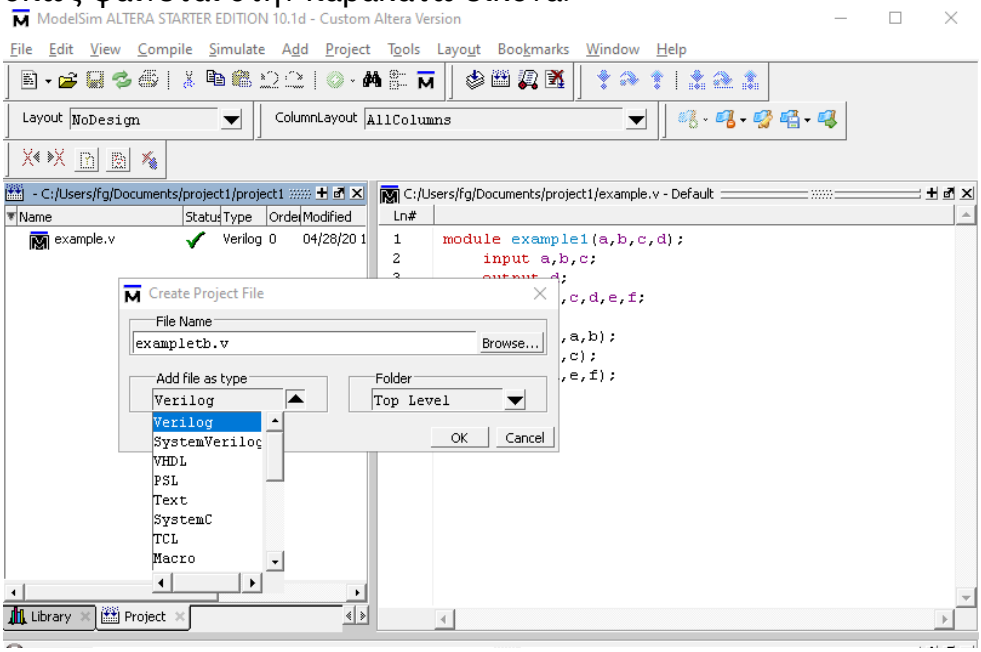
Παράδειγμα Testbench

Χωρίς testbench δεν μπορούμε να προσομοιώσουμε το module *example1* που μόλις φτιάξαμε.

Για να φτιάξουμε testbench, φτιάχνουμε ένα ακόμα module το οποίο το ονομάζουμε exampletb και το γράφουμε σε ένα αρχείο που ονομάζουμε exampletb.v. Το αρχείο αυτό το κατασκευάζουμε από το menu που εμφανίζεται με δεξί κλικ στο ποντίκι μέσα στο παράθυρο του project, όπως φαίνεται στην παρακάτω εικόνα:



Δώστε το όνομα στο αρχείο που θέλουμε. Προσοχή να επιλέξετε την επιλογή Verilog, όπως φαίνεται στην παρακάτω εικόνα:



Έπειτα γράφουμε τον κώδικα του testbench, ο οποίος φαίνεται παρακάτω:

C:/Users/fg/Documents/project1/exampletb.v - Default *

```
Ln#
 1  module example1tb();
 2
 3  //input registers to our instantiated module
 4  reg tb_a;
 5  reg tb_b;
 6  reg tb_c;
 7
 8  //bus for writing data to the inputs
 9  wire [2:0] tb_dut_inputs;
10
11  //wire for reading the output of the instantiated module
12  wire tb_d;
13
14  //this is the instantiated module example1.
15  //the name of the instance is dut
16  example1 dut(tb_a, tb_b, tb_c, tb_d);
17
18  //now we create the bus that consists of three input registers values
19  assign tb_dut_inputs={tb_a, tb_b, tb_c};
20
21    //this block is running only at the beginning of the simulation
22  initial begin
23    {tb_a, tb_b, tb_c}=3'b000;
24
25    //the following line runs forever every 5 time units
26    forever #5 {tb_a, tb_b, tb_c}=tb_dut_inputs+1;
27  end //initial begin
28
29  endmodule
```

Το testbench αποτελείται από 3 καταχωρητές τους οποίους τους χρησιμοποιούμε ως είσοδο για το module example1, είναι τα tb_a, tb_b και tb_c. Τις εξόδους αυτών των τριών καταχωρητών τις ομαδοποιούμε με το bus tb_dut_inputs:
//input registers to our instantiated module
reg tb_a;
reg tb_b;
reg tb_c;
//bus for writing data to the inputs
wire [2:0] tb_dut_inputs;
//now we create the bus that consists of three input registers values
assign tb_dut_inputs={tb_a, tb_b, tb_c};

Φτιάχνουμε και ένα wire tb_d για να διαβάσουμε την έξοδο του module που τεστάρουμε με το testbench:
//wire for reading the output of the instantiated module
wire tb_d;

Στην πορεία φτιάχνουμε ένα instance του module example που σχεδιάσαμε προηγουμένως:
//this is the instantiated module example1.
//the name of the instance is dut

example1 dut(tb_a, tb_b, tb_c, tb_d);
το ονομάσαμε dut (από το design-under-test).

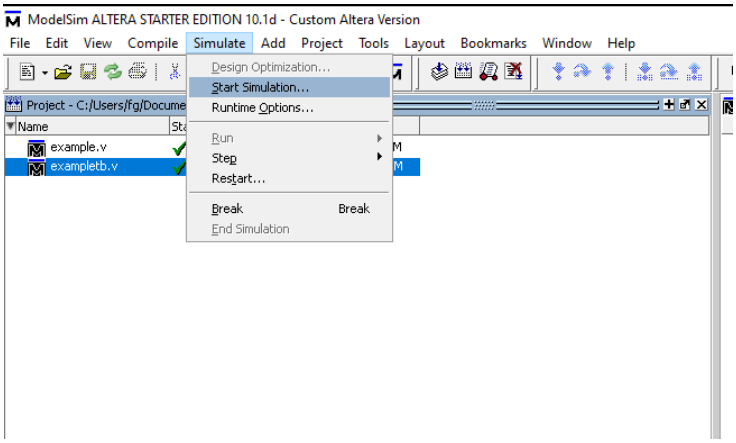
Η καρδιά του testbench είναι το παρακάτω μπλοκ, το οποίο εκτελείτε στην έναρξη της προσομοίωσης:

```
  //this block is running only at the beginning of the simulation
initial begin
  {tb_a, tb_b, tb_c}=3'b000;
  //the following line runs forever every 5 time units
  forever #5 {tb_a, tb_b, tb_c}=tb_dut_inputs+1;
end //initial begin
```

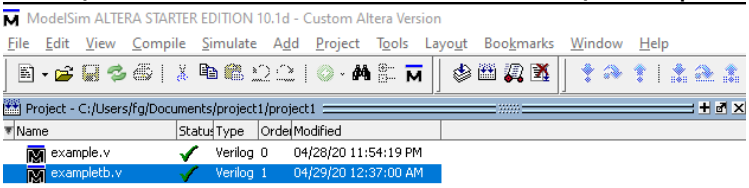
Μηδενίζει τους καταχωρητές:
{tb_a, tb_b, tb_c}=3'b000;
Και στην πορεία για πάντα και κάθε 5 μονάδες χρόνου (αν δεν δηλώσουμε μονάδα χρόνου τότε είναι σε picoseconds), διαβάζει τους καταχωρητές σε μορφή bus, ως ένα δυαδικό αριθμό τριών bits, από το tb_dut_inputs και την τιμή που διάβασε την αυξάνει κατά 1. Το αποτέλεσμα το γράφει πάλι στους καταχωρητές.

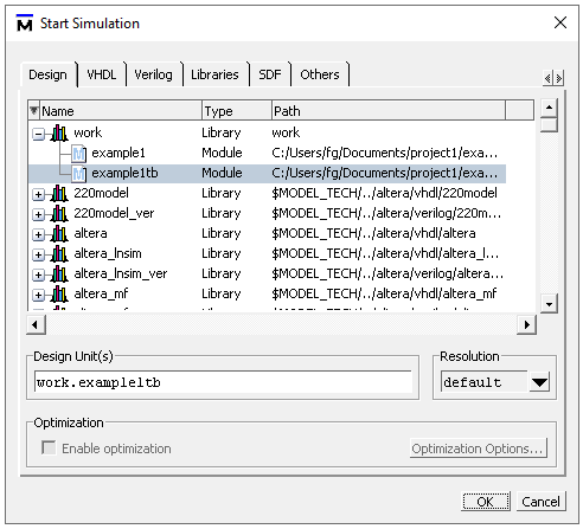
Προσέξτε πως στο testbench συνδέσαμε τους 3 καταχωρητές εισόδου σε ένα bus για να μπορούμε να τους διαβάζουμε σαν ένα δυαδικό αριθμό των $3^{ων}$ bits tb_dut_inputs. Χρησιμοποιήσαμε την εντολή assign:
assign tb_dut_inputs={tb_a, tb_b, tb_c};

Παράδειγμα Προσομοίωσης



Για να ξεκινήσουμε την προσομοίωση εκτελούμε Simulate->"Start Simulation". Θα ανοίξει το παρακάτω παράθυρο, στο οποίο πρέπει να επιλέξουμε το module example1tb, που είναι το testbench που μόλις φτιάξαμε:



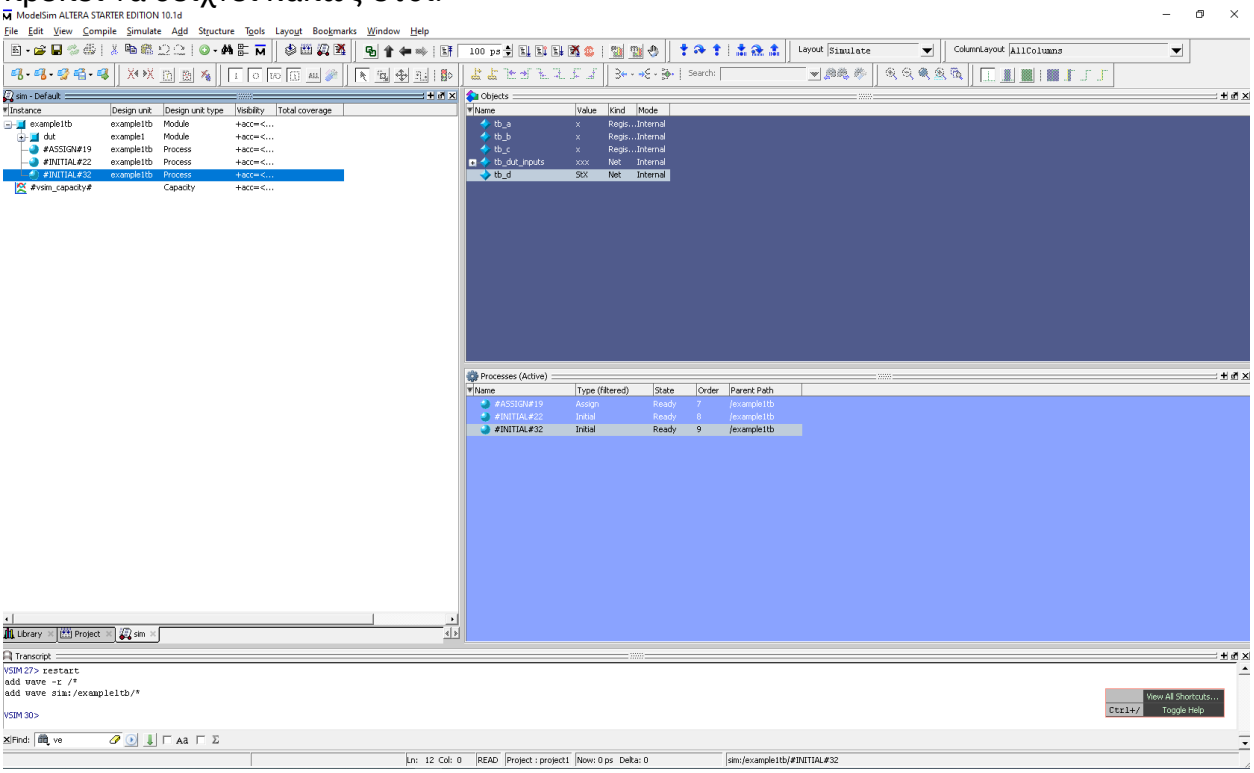


ΠΡΟΣΟΧΗ: Πάντα προσομοιώνουμε ένα testbench, μέσα στο οποίο είναι ως instance module το βασικό μας design.

Μετά επιλέγουμε «OK» και το Modelsim θα μπει σε κατάσταση προσομοίωσης. Περιμένουμε λίγο μέχρι το Modelsim να αλλάξει μορφή, μπορεί να χρειαστούν κάποια δεκάδες δευτερόλεπτα.

Στο στάδιο αυτό μπορείτε να μεγαλώσετε το παράθυρο Transcript και να δείτε την εντολή που εκτελέστηκε στην κονσόλα του Modelsim.
Modelsim> vsim –gui work.example1.tb

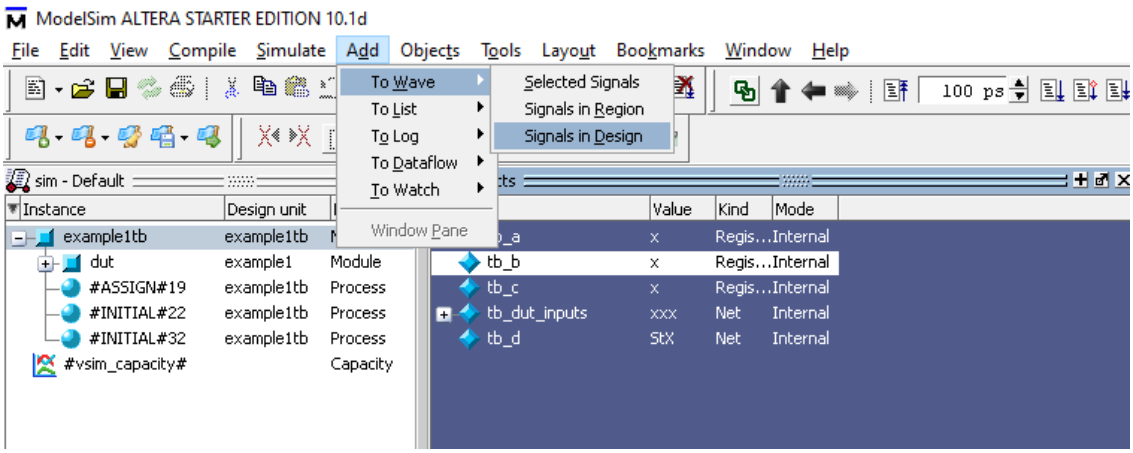
Στο στάδιο που είμαστε το Modelsim μπήκε σε κατάσταση προσομοίωσης και θα πρέπει να δείχνει κάπως έτσι:



Η προσομοίωση έχει ξεκινήσει, αλλά έχει μείνει «παγωμένη» στην χρονική στιγμή t=0.

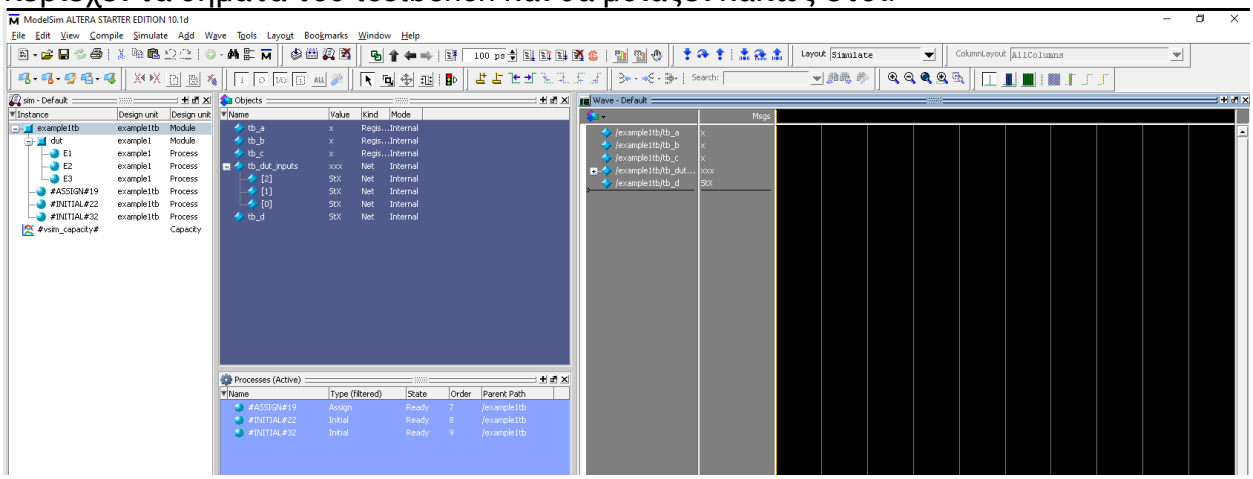
Παράθυρο Sim: Προσέξτε ότι το αριστερά παράθυρο είναι νέο και ονομάζεται Sim – Default. Έχει τα αντικείμενα και τα processes του κώδικά μας ιεραρχικά. Φροντίζουμε να έχουμε επιλεγμένο το testbench εκεί (το example1tb), γιατί θέλουμε να παρατηρήσουμε τι γίνεται σε αυτό.
Παράθυρο Objects: Προσέξτε το κεντρικό μπλε σκούρο παράθυρο που λέγεται Objects, αυτά είναι αντικείμενα του Simulator την ώρα που τρέχει η προσομοίωσή μας.
Παράθυρο Processes: Επίσης προσέξτε το ανοιχτόχρωμο μπλε παράθυρο, που λέγεται Processes, αυτά είναι τα Processes/blocks που είχε το testbench μας.
Σε αυτά τα παράθυρα αν πατήσουμε σε κάποια επιλογή διπλό κλικ, μας πηγαίνει στον κώδικα τους.

Για να προχωρήσουμε την προσομοίωση (και να καταλάβουμε ότι έχει ξεκινήσει), πρέπει να φτιάξουμε ένα παράθυρο κυμματομορφών (wave window). Αυτό το κάνουμε επιλέγοντας από το Menu Add→"To Wave"→"All items in in Region", όπως φαίνεται στην παρακάτω εικόνα.

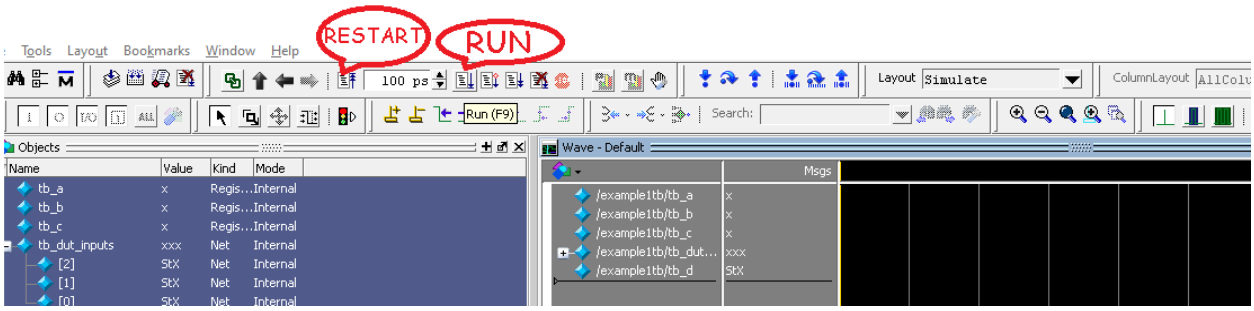


Προσοχή για να δουλέψει σωστά αυτή η επιλογή, πρέπει το στο παράθυρο Sim να είναι επιλεγμένο το testbench, αλλιώς θα μας δείξει τα σήματα του region που είναι επιλεγμένο. Το region επομένως είναι σαν το scope στον προγραμματισμό λογισμικού.

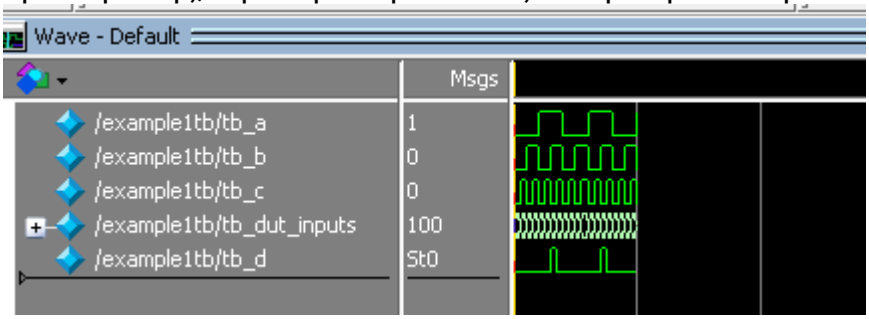
Θα ανοίξει ένα νέο παράθυρο δεξιά που θα ονομάζεται Wave- Default. Εκεί πρέπει να περιέχει τα σήματα του testbench και θα μοιάζει κάπως έτσι:



Προσέξτε ότι οι κυμματομορφές είναι κενές γιατί ακόμα είμαστε στην αρχή της προσομοίωσης, ο χρόνος είναι t=0.
Επόμενο βήμα είναι να πούμε στον προσομοιωτή να προχωρήσει το χρόνο. Το κάνουμε αυτό με την επιλογή run που φαίνεται παρακάτω:



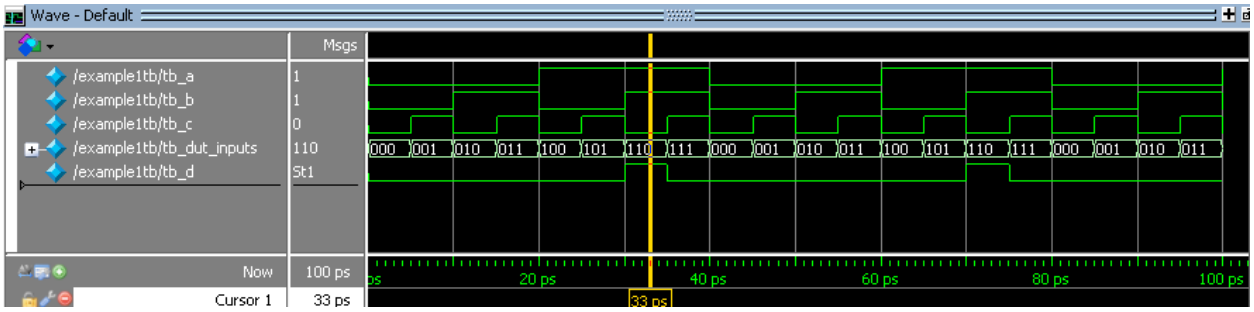
Η επιλογή αυτή θα προχωρήσει την προσομοίωση κατά όσο χρόνο λέει το κουτάκι αριστερά της, δηλαδή 100 ps. Μόλις πατήσουμε run προκύπτει το παρακάτω:



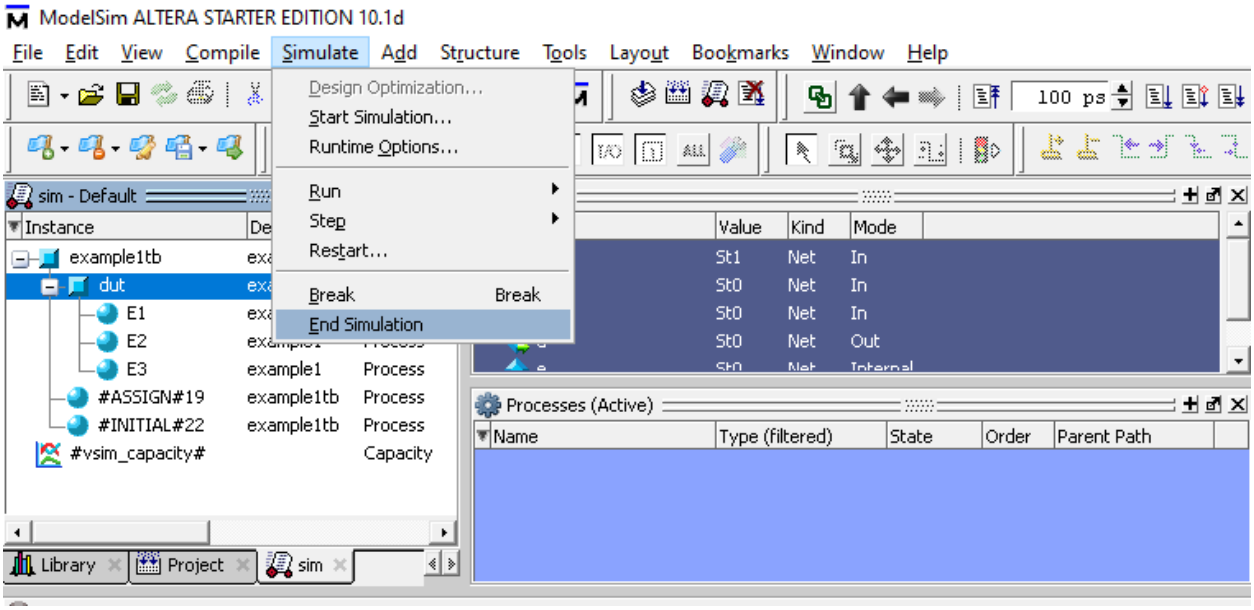
Στο παράθυρο αυτό μπορούμε να περιηγηθούμε και αν βρούμε τις τιμές των σημάτων σε κάθε χρονική στιγμή και να ελέγξουμε οπτικά αν όλα τρέχουν σωστά.

Η περιήγηση γίνεται με το πλήκτρο ctrl από το πληκτρολόγιο πατημένο και την ρόδα από το ποντίκι για zoom-in και zoom-out. Επίσης μπορούμε να πατήσουμε περισσότερες από μία φορές το run και να προχωρήσουμε τον χρόνο παρακάτω.

Παρατηρούμε ότι το σήμα tb_d που είναι η έξοδος του κυκλώματος example1 είναι συνήθως στο λογικό ‘0’ και γίνεται λογικό ‘1’ μόνο όταν οι είσοδοι είναι tb_a=1, tb_b=1 και tb_c=0, που είναι και η σωστή συμπεριφορά που περιμέναμε από τον πίνακα αληθείας του κυκλώματός μας.



Τερματισμός Προσομοίωσης



Για να τερματίσουμε την προσομοίωση επιλέγουμε Simulate→"End Simulation".

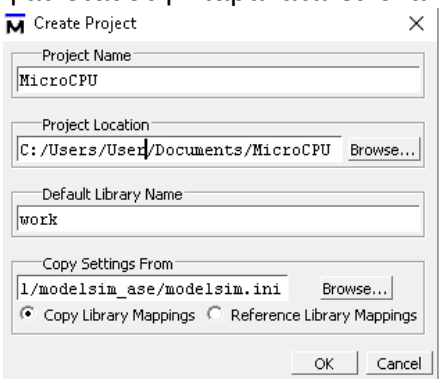
Εγχειρίδιο χρήσης του Microcpu

MicroCPU

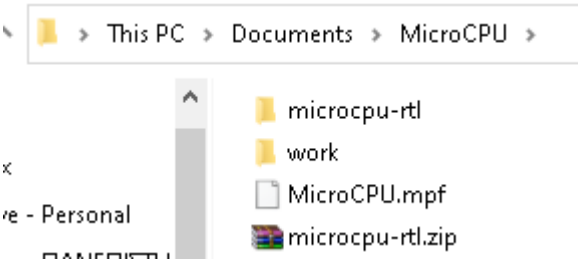
1 Οδηγίες για τον MicroCPU στο Modelsim

1.1 Δημιουργία project και import της Verilog του MicroCPU

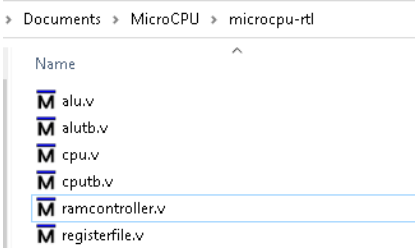
Αρχικά δημιουργείστε ένα νέο project στο Modelsim που θα ονομάσετε MicroCPU, όπως φαίνεται στην παρακάτω εικόνα.



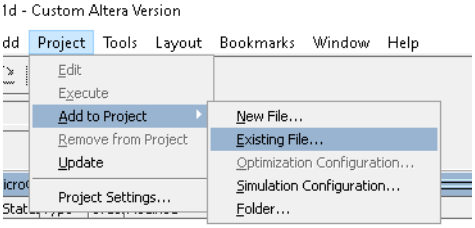
Στην συνέχεια θα βάλετε μέσα στον κατάλογο "MicroCPU" που μόλις δημιουργήθηκε τα αρχεία που περιέχει το συμπιεσμένο αρχείο "microcpu-rtl.zip". Το αρχείο αυτό θα το βρείτε στο site του μαθήματος μαζί με την εκφώνηση της άσκησης.



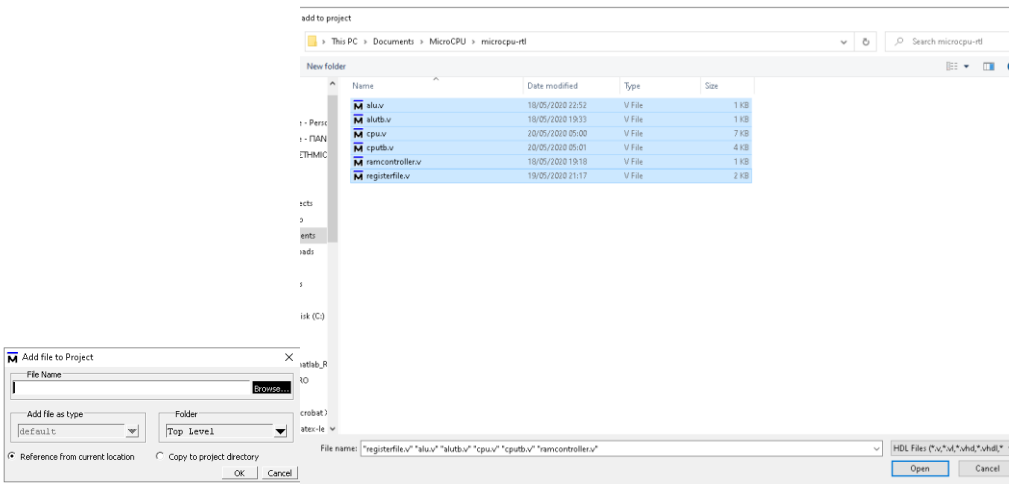
Τα αρχεία αυτά είναι ο RTL σχεδιασμός του MicroCPU γραμμένος στην γλώσσα περιγραφής υλικού Verilog και φαίνονται παρακάτω:



Έπειτα θα ξαναγυρίσετε στο Modelsim όπου και θα συμπεριλάβετε στο νέο project τα αρχεία με τον σχεδιασμό RTL σε Verilog του MicroCPU.

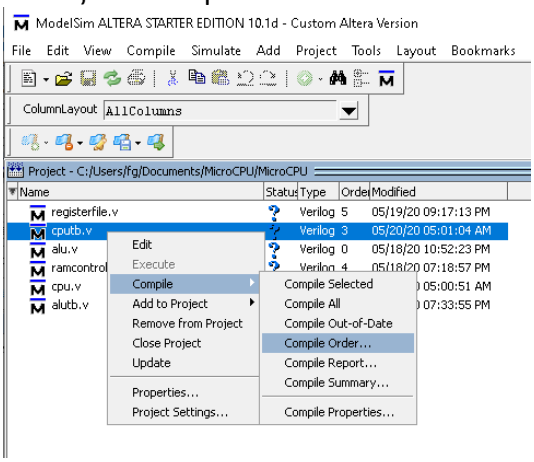


Έπειτα:

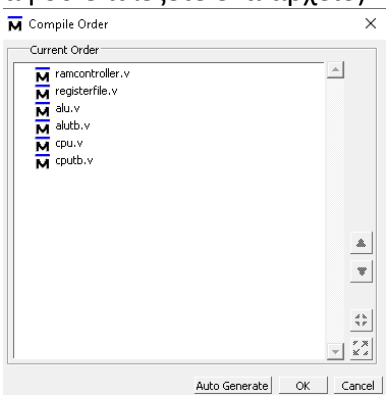


1.2 Μεταγλώττιση

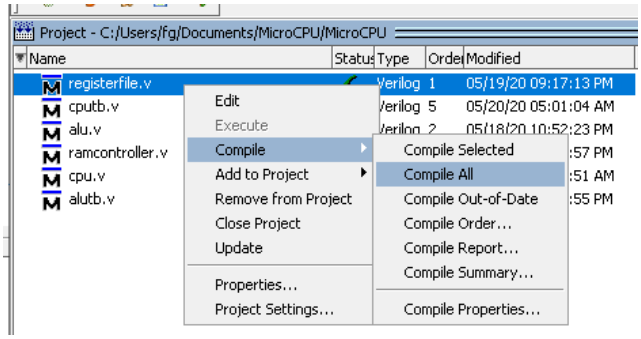
Στην συνέχεια με δεξί κλικ πάνω στα αρχεία του project μέσα από το περιβάλλον Modelsim θα επιλέξετε “compile order”



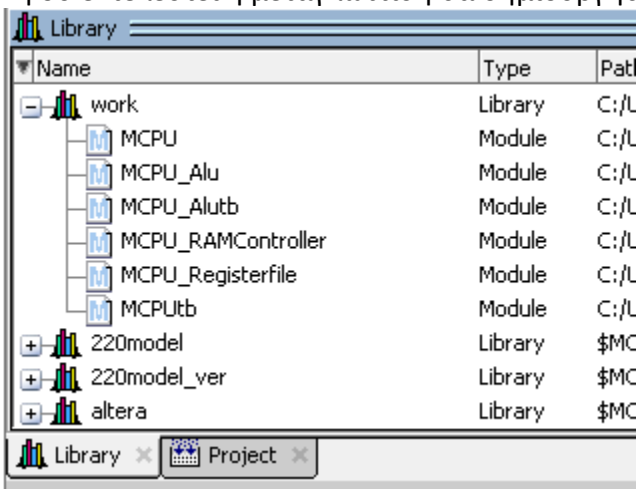
Θα εμφανιστούν οι παρακάτω επιλογές. Δώστε, αν δεν είναι ήδη έτσι, τη σειρά μεταγλώττισής που φαίνεται στην εικόνα (μπορείτε να αλλάξετε τη σειρά χρησιμοποιώντας τα βελάκια δεξιά αφού επιλέξετε ένα αρχείο):



Στη συνέχεια εκτελέστε μεταγλώττιση με την επιλογή “compile”→”compile all”.



Η μεταγλώττιση χρειάζεται να εκτελείτε κάθε φορά που αλλάζετε τα αρχεία κώδικα.
Αφού εκτελεστεί η μεταγλώττιση θα δημιουργηθούν τα παρακάτω modules στην βιβλιοθήκη:



Τα οποία είναι τα εξής:
MCPU→είναι η μονάδα ελέγχου (Control Unit) του MicroPro
MCPU_Alu→είναι η Αριθμητική και Λογική Μονάδα (Arithmetic Logic Unit - ALU) του MicroPro
MCPU_Alutb→είναι testbench για την επιβεβαίωση της ALU του MicroPro
MCPU_RAMController→είναι ο ελεγκτής μνήμης και η μνήμη του MicroPro
MCPU_Registerfile→είναι το αρχείο καταχωρητών (Register File) του MicroPro
MCPUtb →είναι το testbench του MicroPro

1.3 Εκτέλεση προγράμματος στον επεξεργαστή MicroCPU

Για να εκτελέσουμε ένα πρόγραμμα στον MicroCPU πρέπει να φτιάξουμε ένα testbench στο οποίο θα δημιουργήσουμε ένα instance/αντικείμενο του module του MicroCPU. Έπειτα πρέπει να αποθηκεύσουμε το πρόγραμμα που θέλουμε να εκτελέσουμε στην μνήμη του MicroCPU και μετά με κατάλληλη σηματοδότηση των σημάτων reset και clk μπορεί να γίνει η εκτέλεση του προγράμματος.
Φυσικά το πρόγραμμα πρέπει να είναι σε γλώσσα μηχανής, επομένως πρέπει αρχικά να μελετήσουμε την αρχιτεκτονική του συνόλου εντολών του MicroCPU που βρίσκετε στο επόμενο κεφάλαιο.
Για να διευκολύνουμε τη συγγραφή προγραμμάτων στο MicroCPU υπάρχει ένα έτοιμο testbench στο αρχείο *cputb.v,* το οποίο εκτελεί ένα πρόγραμμα που υπολογίζει την ακολουθία

των αριθμών Fibonacci. Υπενθυμίζουμε πως στην ακολουθία Fibonacci το X_i παράγεται ως το άθροισμα των X_{i-2} και X_{i-2}. Δηλαδή: $X_i=X_{i-1}+X_{i-2}$

```
module MCPUtb();

reg reset, clk;
MCPU cpuinst (clk, reset);
```

Αρχικά φτιάχνουμε ένα instance του module MCPU, το οποίο είναι το top-level module του επεξεργαστή MicroCPU. Το τροφοδοτούμε με τα σήματα clk και το reset.

```
initial begin
  reset=1;
  #10  reset=0;
end
```

Το μπλοκ αυτό εκτελείτε στην αρχή της προσομοίωσης μόνο μια φορά και θέτει το σήμα reset του MicroCPU στο λογικό-1 για 10 ps. Μετά από 10ps το θέτει στην τιμή λογικό-0.

```
always begin
  #5 clk=0;
  #5 clk=1;
end
```

Το μπλοκ αυτό εκτελείτε για πάντα και είναι η γεννήτρια του ρολογιού clk του MicroCPU

```
/********ASSEMBLER*****/

integer file, i;
reg[cpuinst.WORD_SIZE-1:0] memi;
parameter  [cpuinst.OPERAND_SIZE-1:0]  R0  = 0; //4'b0000
parameter  [cpuinst.OPERAND_SIZE-1:0]  R1  = 1; //4'b0001
parameter  [cpuinst.OPERAND_SIZE-1:0]  R2  = 2; //4'b0010
parameter  [cpuinst.OPERAND_SIZE-1:0]  R3  = 3; //4'b0011
```

Τώρα ξεκινάει η δήλωση μεταβλητών και καταχωρητών που θα μας είναι χρήσιμες για την προσομοίωση.
file: θα αποθηκεύσουμε το πρόγραμμα σε γλώσσα μηχανής σε ένα αρχείο
Δηλώνουμε τους integers R0,R1,R2,R3 και R4 στους κωδικούς των αντίστοιχων καταχωρητών του MicroCPU για να μπορούμε να χρησιμοποιήσουμε τα σύμβολα Rx κατά την δημιουργία του προγράμματός μας. Αυτό μας δίνει την δυνατότητα να έχουμε μια άμεση μεταγλώττιση των συμβόλων στους κωδικούς. Είναι δηλαδή μια απλοποιημένη μορφή assembler.

```
initial
begin
   for(i=0;i<256;i=i+1)
   begin
     cpuinst.raminst.mem[i]=0;
   end
   cpuinst.regfileinst.R[0]=0;
   cpuinst.regfileinst.R[1]=0;
```

Στο μπλοκ αυτό αρχικά μηδενίζουμε όλες τις λέξεις της μνήμης και όλους τους καταχωρητές του MicroCPU. Θέλουμε έτσι να αποφύγουμε

```
  cpuinst.regfileinst.R[2]=0;
  cpuinst.regfileinst.R[3]=0;
```

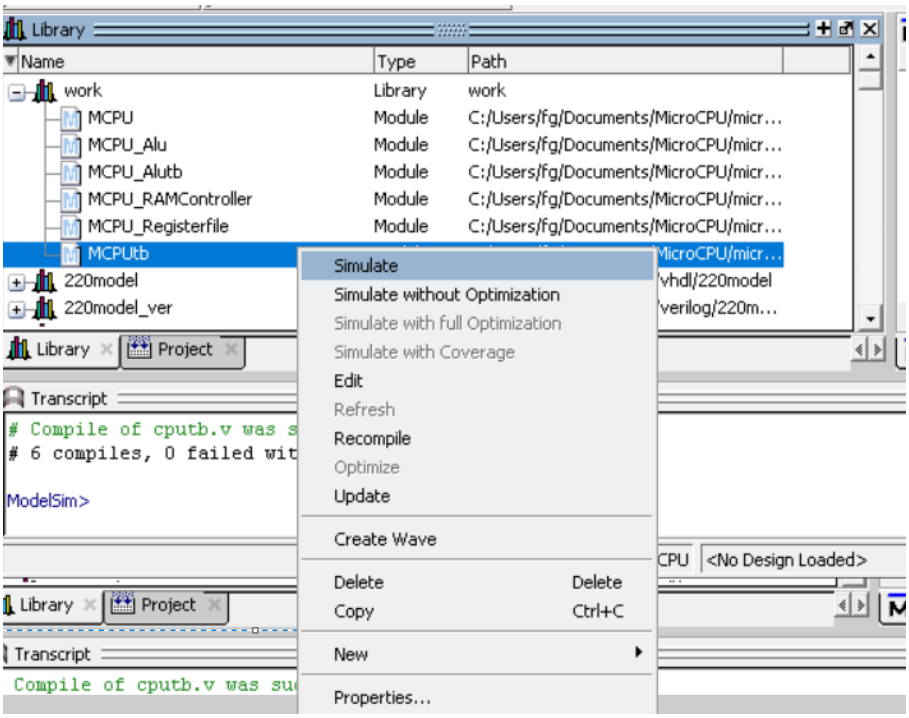
τα undefined Xes στην προσομοίωσή μας.
Ο κώδικας που ακολουθεί, γράφει στην μνήμη του MCPU το **πρόγραμμα/benchmark** που θέλουμε να εκτελέσει.
Κάθε γραμμή είναι μια εντολή.
Μεταφράζω:
Θέση μνήμης: Εντολή
0: R0=0;
1: R1=1;
2: R2=2;
do{
 3: R0=R1;
 4: R1=R2;
 5: R2=R0+R1;
 6:mem[20]=R2;
 7:R3=mem[20];
 8:R0=R0+R0
}
9:while(R2!=0)

```
  i=0;  cpuinst.raminst.mem[0]={cpuinst.OP_SHORT_TO_REG, R0, 8'b00000000};
  i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_SHORT_TO_REG, R1, 8'b00000001};
  i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_SHORT_TO_REG, R2, 8'b00000010};

  i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_MOV, R0, R1, 4'b0000};
  i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_MOV, R1, R2, 4'b0000};
  i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_ADD, R2, R0, R1};
  i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_STORE_TO_MEM, R2, 8'b00010100};
  i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_LOAD_FROM_MEM, R3, 8'b00010100};
  i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_ADD, R0, R0, R0};

  i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_BNZ, R2, 8'b00000011};
```

Στην συνέχεια ανοίγει το αρχείο “program.list” και γράφει στο αρχείο όλη την μνήμη του MicroCPU.

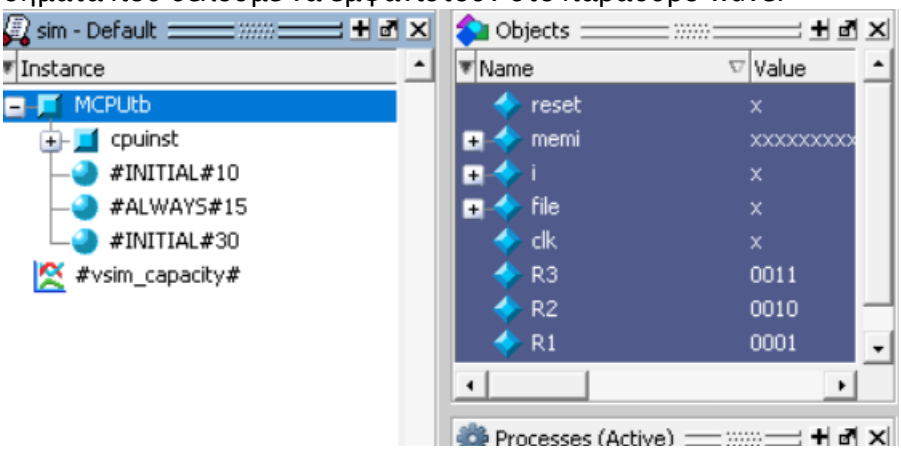
```
  file = $fopen("program.list","w");
  for(i=0;i<cpuinst.raminst.RAM_SIZE;i=i+1)
  begin
   memi=cpuinst.raminst.mem[i];

   $fwrite(file, "%b_%b_%b_%b\n",
     memi[cpuinst.INSTRUCTION_SIZE-1:cpuinst.INSTRUCTION_SIZE-cpuinst.OPCODE_SIZE],
     memi[cpuinst.OPCODE_SIZE*3-1:2*cpuinst.OPCODE_SIZE],
     memi[cpuinst.OPCODE_SIZE*2-1:cpuinst.OPCODE_SIZE],
     memi[cpuinst.OPCODE_SIZE-1:0]);
  end
  $fclose(file);
end
endmodule
```

Αφού έχουμε τελειώσει μη την μεταγλώττιση και του cputb.v, ήρθε η ώρα να εκτελέσουμε το πρόγραμμα μας. Για να το κάνουμε αυτό, πρέπει να ξεκινήσουμε την προσομοίωση του module που περιέχει το πρόγραμμα, το οποίο είναι το MCPUtb.



Αυτό το κάνουμε από το παράθυρο Library με δεξί κλικ πάνω στο module και επιλογή του "Simulate".

Θα εμφανιστεί το παράθυρο αντικειμένων, που γνωρίζουμε, από το οποίο θα επιλέξουμε τα σήματα που θέλουμε να εμφανιστούν στο παράθυρο wave.



Από το module cpuinst (που πρόκειται για το instance της μονάδας ελέγχου) θέλουμε τα σήματα STATE_AS_STR, pc, opcode, operand1, operand2, operand3, regset_cmd, regset_wb, regdatatoload, RegOp1.

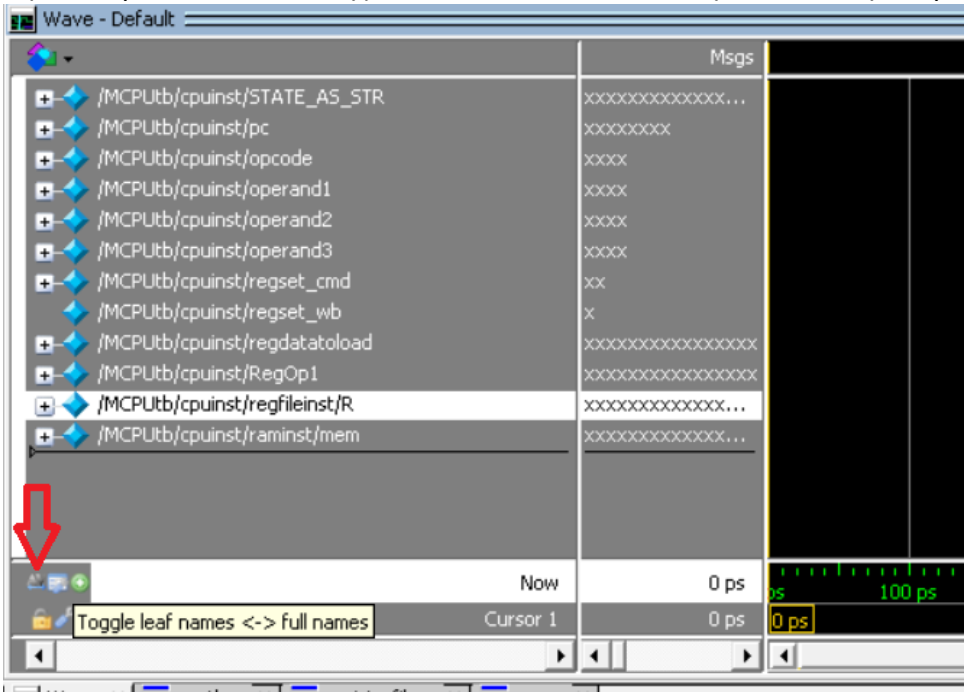
Από το module cpuinst.regfileinst θέλουμε το σήμα R, αυτό έχει τους καταχωρητές.

Από το module cpuinst.raminst θέλουμε το mem, που είναι η μνήμη.

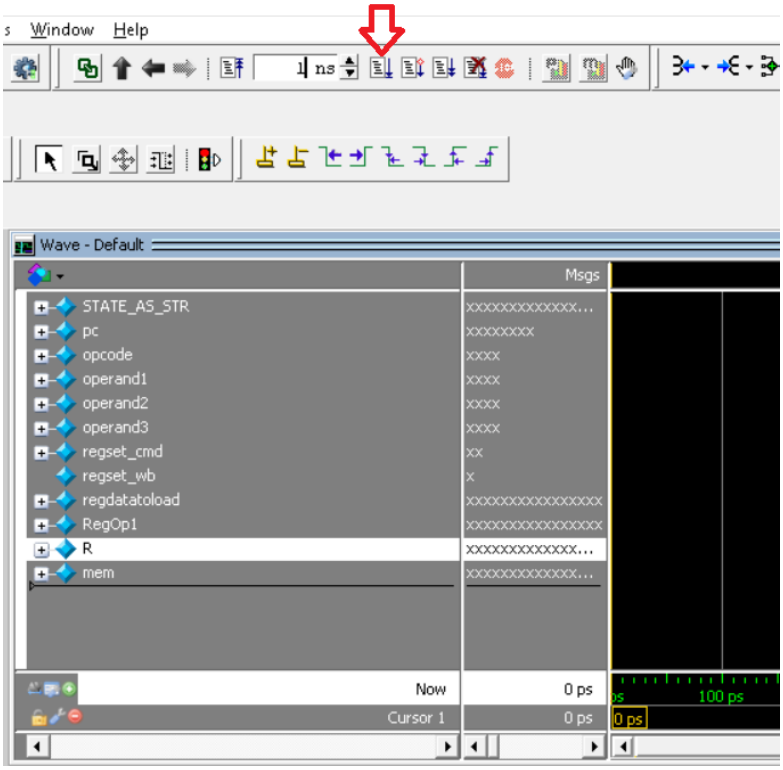
Καλό θα ήταν να πάρετε ένα αντίγραφο του σήματος mem[20] μέσα στο παράθυρο waves, ξεχωριστά από την συνολική μνήμη γιατί εκεί αποθηκεύουμε το αποτέλεσμα.

Φυσικά ανάλογα με το πρόγραμμα που θα εκτελείτε ή ανάλογα με τον σχεδιασμό υλικού (π.χ. κάποιας νέας εντολής) που θα κάνετε, θα χρειάζεται να βάζετε τα σήματα που εμπλέκονται.

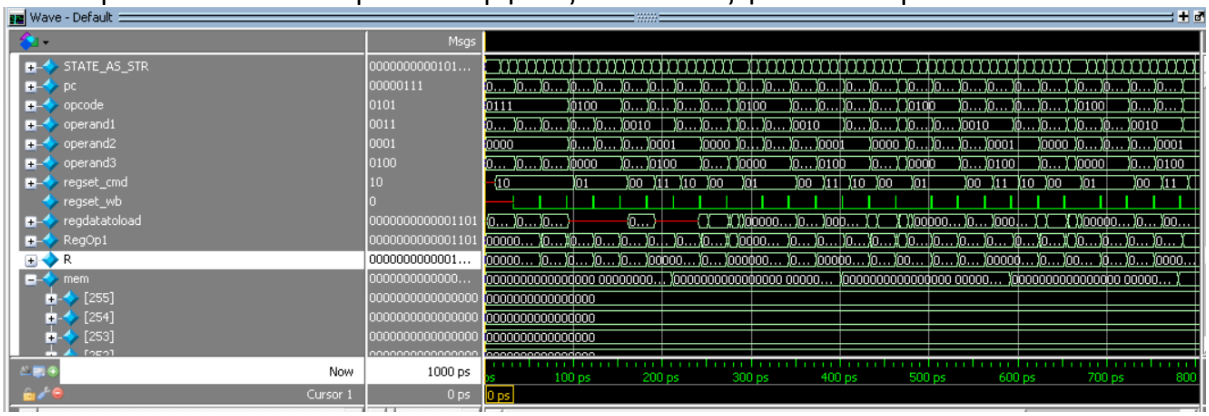
Αφού τα βάλετε θα καταλήξετε σε κάτι σαν αυτό που φαίνεται στην παρακάτω εικόνα:



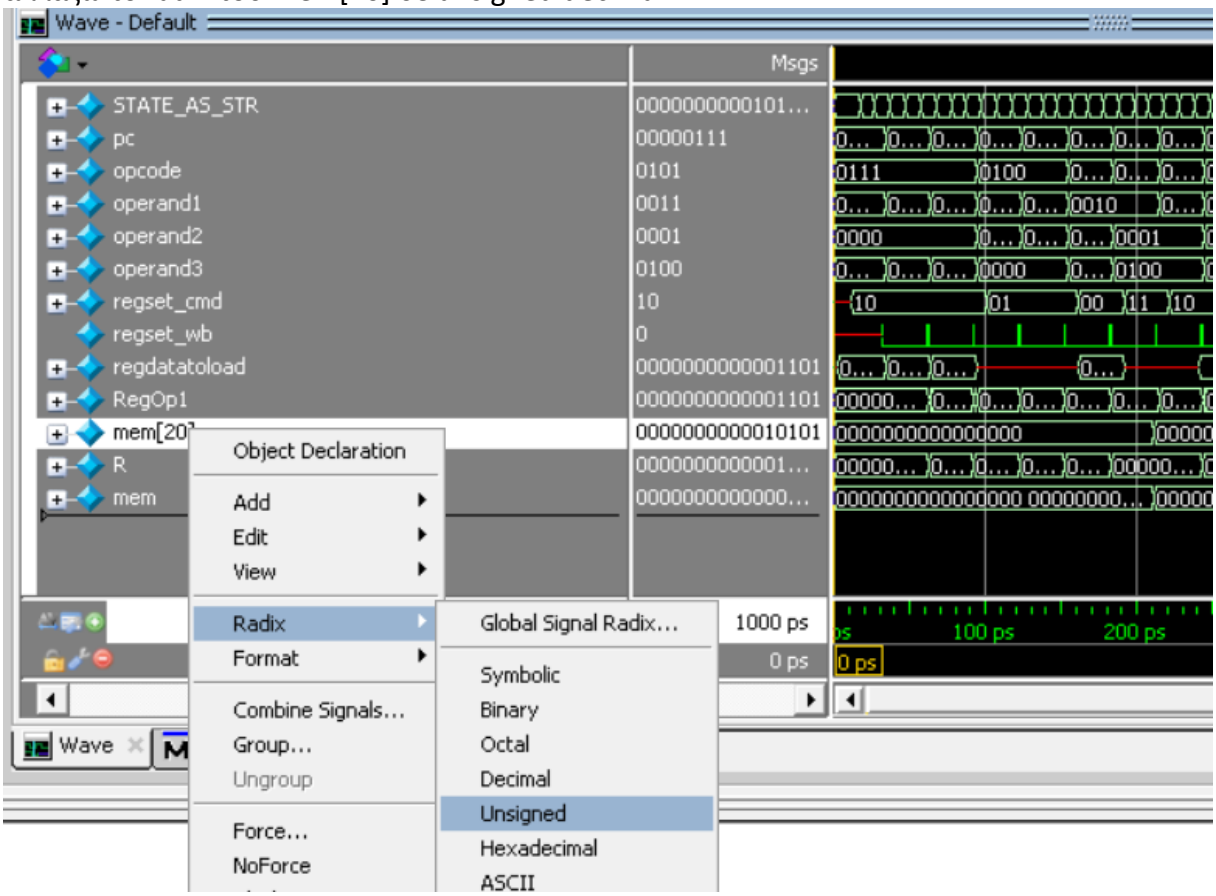
(όπως βλέπετε στην εικόνα ξέχασα το mem[20] και το έχω προσθέσει αργότερα)
Προσέξτε το κουμπάκι που σας δείχνω με κόκκινο βελάκι στην παραπάνω εικόνα. Πατώντας το μπορείτε να απενεργοποιήσετε/ενεργοποιήσετε το πλήρες όνομα των σημάτων. Έτσι μπορείτε να τα κάνετε να φαίνονται πιο σύντομα, όπως φαίνεται παρακάτω.



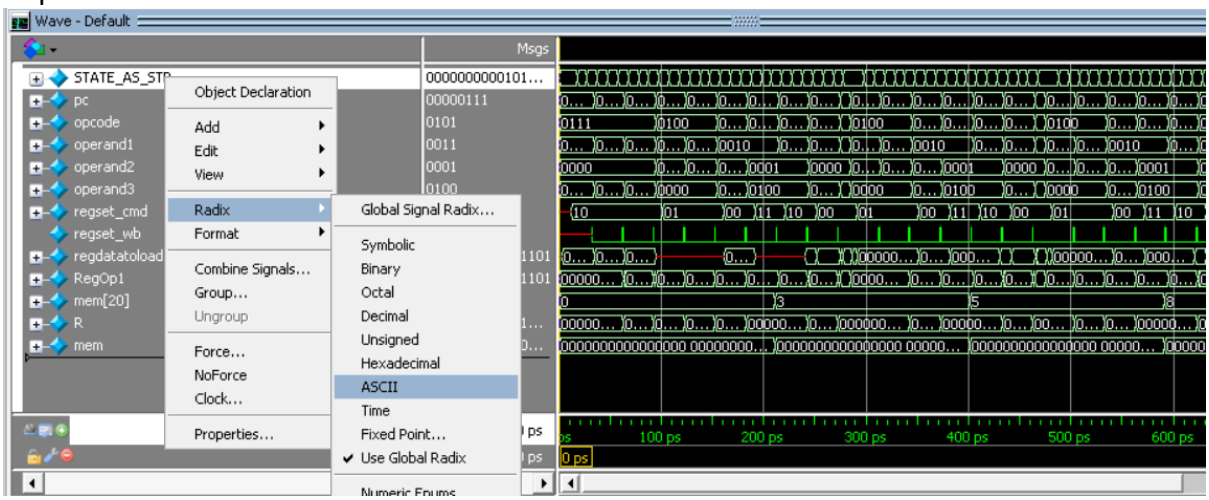
Τώρα ήρθε η ώρα να βάλουμε 1 ns εκτέλεση στην προσομοίωση μας και να πατήσουμε εκτέλεση/run.
Θα δούμε πολλά αποτελέσματα να εμφανίζονται. Όπως φαίνεται παρακάτω.



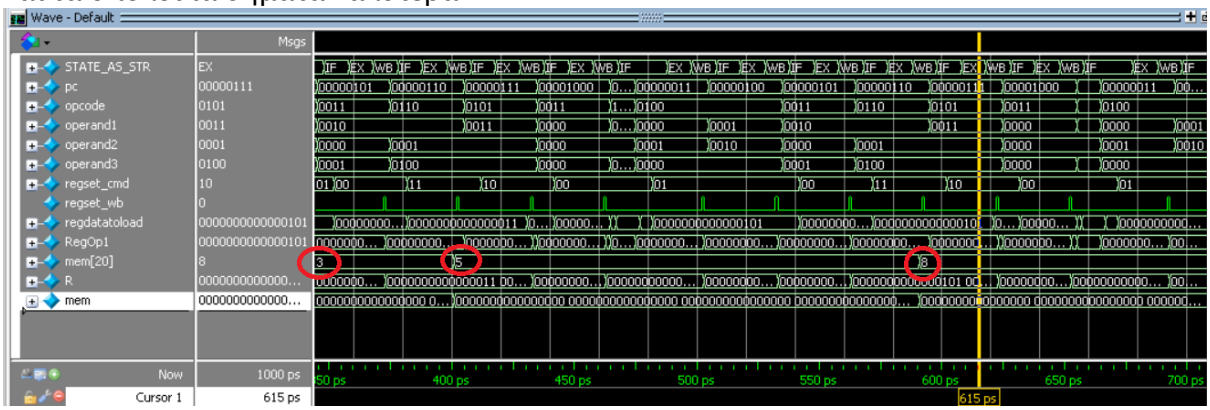
Όμως, τα αποτελέσματα από την ακολουθία Fibonacci θα βρίσκονται αποθηκευμένα στην διεύθυνση μνήμης 20 και στον καταχωρητή R2. Μπορούμε να αλλάξουμε το radix αυτών σε unsigned για να τα βλέπουμε σε 10δικό σύστημα ώστε να αντιλαμβανόμαστε πιο εύκολα αν λειτουργεί σωστά το πρόγραμμα.
Καλό θα ήταν να πάρετε ένα αντίγραφο του σήματος mem[20] μέσα στο παράθυρο waves, ξεχωριστά από την συνολική μνήμη γιατί εκεί αποθηκεύουμε το αποτέλεσμα. Εδώ βλέπετε πως αλλάζω το radix του mem[20] σε unsigned decimal.



Ομοίως καλό θα ήταν να αλλάξετε το radix του STATE_AS_STR σε ASCII όπως φαίνεται στην παρακάτω εικόνα:

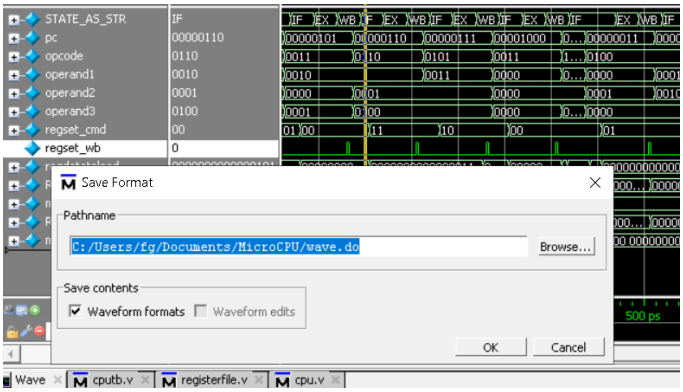


Προσέξτε πως ήδη φαίνεται η ακολουθία Fibonacci στο mem[20] (3,5,8,…)
Αν τώρα εστιάσουμε πιο κοντά στην κυμματομορφή θα δούμε και τις καταστάσεις διοχέτευσης και τα υπόλοιπα σήματα καλύτερα:

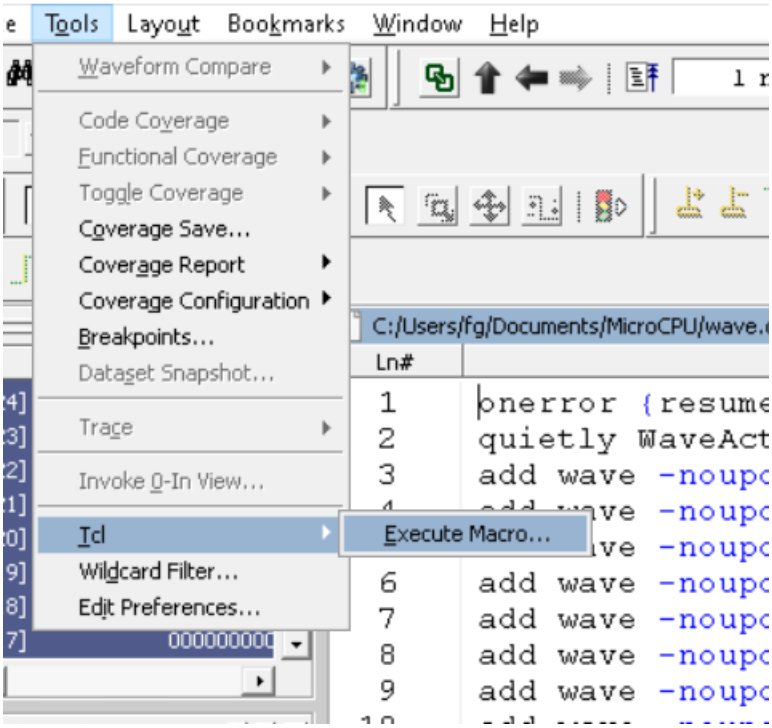


Μην το χάσετε!

Τέλος, επειδή η διαδικασία επιλογής σημάτων και επιλογής της μορφοποίησής τους είναι επίπονη, μπορείτε να την σώσετε ώστε να μην χρειάζεται να γίνεται κάθε φορά που εκτελείτε μια προσομοίωση. Αυτό γίνεται ως εξής: όσο είναι το focus στο παράθυρο wave, επιλέξτε file->save format και επιλέξτε να το σώσετε σε ένα αρχείο με κατάληξη ".do", όπως wave.do.



Αν για κάποιο λόγο χάσατε τα σήματα σας από το παράθυρο waves, επειδή για παράδειγμα σταματήσατε τελείως το simulation χωρίς να κάνετε restart (αν πατάτε restart στο simulation δεν χάνονται τα σήματα, αλλά αυτό που ακολουθεί είναι χρήσιμο για την περίπτωση που κλείσατε το modelsim), τότε μπορείτε να φορτώσετε το αρχείο waves.do από την επιλογή tools->tcl->execute macro. Από εκεί επιλέξτε το αρχείο waves.do:

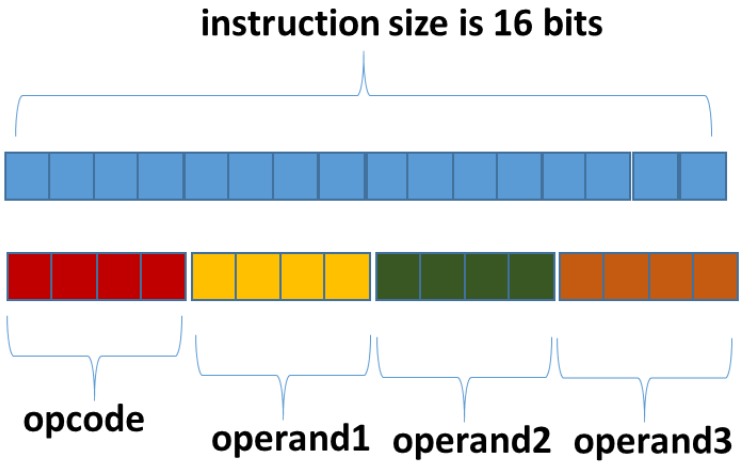


Και θα εκτελεστεί ο κώδικάς του στην κονσόλα του Modelsim, οποίος φορτώνει τα σήματα στο παράθυρο wave (ακόμα και να μην υπάρχει wave το ανοίγει, φτάνει να έχει ξεκινήσει το simulation).

Στην συνέχεια θα δούμε λεπτομέρειες της αρχιτεκτονικής (τα διάφορα modules και τη διασύνδεσή τους) του MicroCPU.

2 Αρχιτεκτονική του MicroCPU

2.1 Σύνολο Εντολών μηχανής (instruction set) του MicroCPU



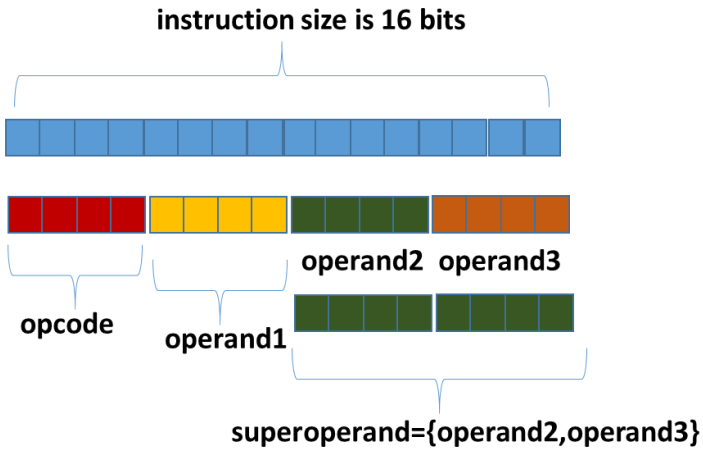
Π.χ.
ADD operand1 operand2 operand3
Ένας operand μπορεί να είναι η κωδική ονομασία ενός καταχωρητή.
Π.χ.
ADD 3 2 1
Προσθέτει τα περιεχόμενα των καταχωρητών R2 και R1 και γράφει το αποτέλεσμα τον καταχωρητή R3. Προσέξτε ότι στην ουσία γράφονται οι κωδικές ονομασίες των καταχωρητών ως operands π.χ. 0→R0, 1→R1, 2→R2, 3→R3. Έπειτα κατά το instruction decoding αυτές οι κωδικές ονομασίας αντικαθίστανται από τους καταχωρητές.
Για λόγους απλούστευσης, η σύνταξη μιας εντολής στην οποία τα operands είναι οι κωδικοί των καταχωρητών, θα περιγράφεται ως εξής:
Εντολή Rd Ra Rb
Όπου Rd ο καταχωρητής destination (ο καταχωρητής δηλαδή στον οποίο καταλήγουν τα δεδομένα μετά το τέλος της εκτέλεσης του instruction) και Ra, Rb τα ορίσματα που εμπλέκονται. Ο MicroCPU έχει τις εξής κατηγορίες εντολών:

- **Εντολές επεξεργασίας** που εμπλέκουν την ALU για αριθμητικές και λογικές πράξεις, π.χ.:
  Bitwise λογικό ΚΑΙ: *AND Rd Ra Rb*
  Bitwise λογικό Ή: *OR Rd Ra Rb*
  Bitwise λογικό αποκλειστικό Ή: *XOR Rd Ra Rb*
  Πρόσθεση: *ADD Rd Ra Rb*

  Το αποτέλεσμα γράφεται στον καταχωρητή Rd. Οι πράξεις εμπλέκουν τους Ra και Rb.
- **Εντολές μετακίνησης δεδομένων από καταχωρητή σε καταχωρητή**:
  MOV Rd Ra
  τα δεδομένα αντιγράφονται από τον καταχωρητή Ra στον καταχωρητή Rd

- **Εντολές φόρτωσης δεδομένων από μνήμη σε καταχωρητή**:

  Load_FROM_MEM Rd address
  αντιγράφει τα δεδομένα από την διεύθυνση μνήμης address στον καταχωρητή Rd. Προσέξτε ότι η μνήμη του MicroCPU είναι 256 λέξεων, άρα απαιτείται διεύθυνση των 8 bits για αν αναφερθεί κάποιος σε όλες τις λέξεις της μνήμης. Για τον λόγο αυτό το address της Load_FROM_MEM είναι ένας superoperand και χρησιμοποιεί τόσο τα bits του operand2 όσο και τα bits του operand3. Άρα συνολικά έχει διαθέσιμα 8 bits, όπως φαίνεται στην πατακάτω εικόνα:



- **Εντολές αποθήκευσης δεδομένων από καταχωρητή στην μνήμη**:

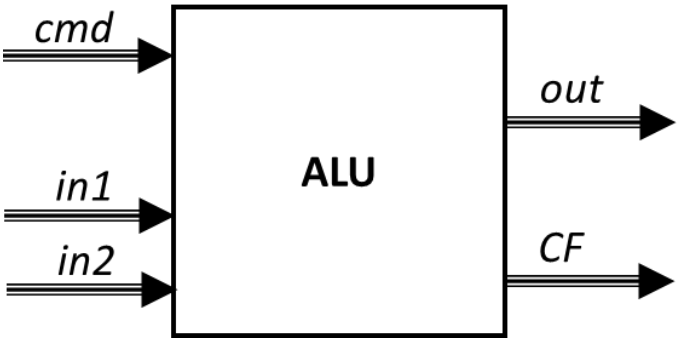
  STORE_TO_MEM R address
  αποθηκεύει στη διεύθυνση μνήμης address τα δεδομένα του καταχωρητή R. Η address είναι ένας superoperand των 8 bits.
- **Εντολές Αρχικοποίησης τιμών**:

  OP_SHORT_TO_REG Rd value
  αντιγράφει την τιμή value των 8 bits στον καταχωρητή Rd.
- **Εντολές διακλάδωσης:**

  BNZ Rc address
  μετακινεί τον program counter στην τιμή address αν ο καταχωρητής Rc δεν είναι 0. (Branch if Not Zero). Η address είναι ένας superoperand των 8 bits.

2.2 Η Αριθμητική και λογική μονάδα (ALU) του MicroCPU

Η αριθμητική λογική μονάδα είναι το module MCPU_Alu και θα το βρείτε στο αρχείο *alu.v*.
Η αριθμητική/λογική μονάδα είναι υπεύθυνη για την επεξεργασία των δεδομένων στον επεξεργαστή. Εκτελεί αριθμητικές (πρόσθεση, αφαίρεση, πολλαπλασιασμό κτλ) και λογικές πράξεις (bitwise not, or, xor κτλ.) με τους καταχωρητές. Επιστρέφει τα δεδομένα της σε κάποιον καταχωρητή και τα flags από τις αριθμητικές πράξεις (π.χ. κρατούμενα υπολογισμών carry flag, διαίρεσης με το μηδέν (zero flag) κτλ.)



Το διάγραμμα εισόδων/εξόδων της ALU του MicroCPU φαίνεται παραπάνω.

- cmd: Ο τύπος της πράξης που πρόκειται να εκτελεστεί καθορίζεται από την είσοδο cmd. Μπορεί να πάρει τις εξής τιμές:
  parameter [CMD_SIZE-1:0] CMD_AND = 0; //2'b00
  parameter [CMD_SIZE-1:0] CMD_OR = 1; //2'b01
  parameter [CMD_SIZE-1:0] CMD_XOR = 2; //2'b10
  parameter [CMD_SIZE-1:0] CMD_ADD = 3; //2'b11
  προσέξτε ότι μόνο 4 τύποι πράξεων μπορούν να εκτελεστούν από την ALU του MicroCPU (bitwise AND, bitwise OR, bitwise XOR και ADD). Για τον λόγο αυτό χρειάζονται CMD_SIZE=2 bits για την κωδικοποίηση του τύπου της εντολής (parameter [CMD_SIZE-1:0]).
- in1, in2: τιμές καταχωρητών εισόδου πάνω οι οποίες θα χρησιμοποιηθούν από την πράξη που πρόκειται να εκτελεστεί. Το μήκος των καταχωρητών του MicroCPU είναι 16 bits.
  parameter WORD_SIZE=16;
  Παρόλαυτά έχετε υπόψιν σας ότι μια παράμετρο μπορεί να αλλάξει από το σημείο που δημιουργείται ένα αντικείμενο και άρα να χρησιμοποιηθεί η ίδια ALU με διαφορετικό μήκος καταχωρητών. Επίσης προσέξτε ότι στον σχεδιασμό του MicroCPU έχουμε θεωρήσει ότι το μέγεθος μιας λέξης μνήμης (WORD_SIZE) είναι ίσο με το μέγεθος των καταχωρητών. Αυτό δεν ισχύει για όλους τους μικροεπεξεργαστές.
- out, CF: Ο out είναι καταχωρητής στον οποίο εκχωρείται το αποτέλεσμα της πράξης που εκτελέστηκε από την ALU. Το CF είναι ψηφίο κρατουμένου της πρόσθεσης.

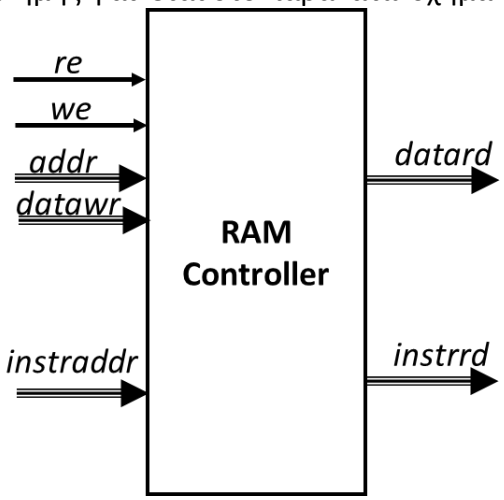
Το testbench της ALU

Στο αρχείο *alutb.v* θα βρείτε ένα testbench της ALU που μεταγλωττίζεται στο αντικείμενο MCPU_Alutb το οποίο δοκιμάζει τυχαίες τιμές και εντολές στις εισόδους της ALU.
Προσθέστε στο testbench ένα always block το οποίο θα ελέγχει τις αποκρίσεις της ALU και θα ενημερώνει έναν καταχωρητή iscorrect για το αν ο υπολογισμός έγινε σωστά από την Alu.

2.3 Η μνήμη τυχαίας προσπέλασης (RAM) του MicroCPU

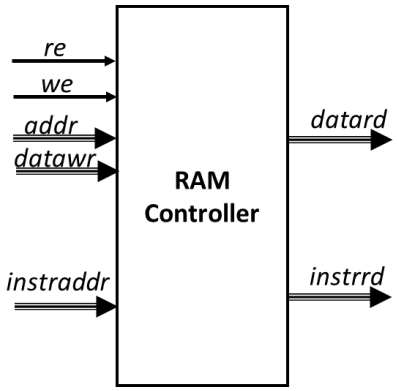
Η μνήμη τυχαίας προσπέλασης είναι το module MCPU_RAMController και θα το βρείτε στο αρχείο *ramcontroller.v*.

Ο MicroCPU έχει μία κοινή μνήμη για εντολές και δεδομένα. Η μνήμη έχει μία δυνατότητα εγγραφή και δύο ταυτόχρονες δυνατότητες ανάγνωσης. Η αρχιτεκτονική του ελεγκτή της μνήμης φαίνεται στο παρακάτω σχήμα:



Εγγραφή δεδομένων: Η εγγραφή θα χρησιμοποιείτε από τα προγράμματα που εκτελούνται στον MicroCPU για να αποθηκεύουν δεδομένα στην μνήμη. Η εγγραφή ελέγχεται από το σήμα *we* (write enable). Τα δεδομένα από το bus *datawr* γράφονται στην θέση μνήμης *addr* όταν το *we* γίνεται λογικό-1.

```
module MCPU_RAMController(we, datawr, re, addr, datard, instraddr, instrrd);
parameter WORD_SIZE=16;
parameter ADDR_WIDTH=8;
parameter RAM_SIZE=1<<ADDR_WIDTH;

input we, re;

input [WORD_SIZE-1:0] datawr;

input [ADDR_WIDTH-1:0] addr;
input [ADDR_WIDTH-1:0] instraddr;

output [WORD_SIZE-1:0] datard;
output [WORD_SIZE-1:0] instrrd;

reg [WORD_SIZE-1:0] mem[RAM_SIZE-1:0];
```




```
reg [WORD_SIZE-1:0] datard;
reg [WORD_SIZE-1:0] instrrd;
```

```
always @ (addr or we or re or datawr)
begin
  if(we)begin
    mem[addr]=datawr;
  end
  if(re) begin
    datard=mem[addr];
  end
end

always @ (instraddr)
begin
   instrrd=mem[instraddr];
end
endmodule
```

Ανάγνωση δεδομένων και εντολών

Η μνήμη του MicroCPU έχει δύο δυνατότητες ανάγνωσης. Η μία χρησιμοποιείται από τα προγράμματα που εκτελούνται στον MicroCPU για να διαβάζουν δεδομένα και η άλλη από την μονάδα ελέγχου του MicroCPU για να διαβάζει τις εντολές που πρόκειται να εκτελέσει. Όπως καταλαβαίνεται ο επεξεργαστής MicroCPU έχει κοινή μνήμη τόσο για δεδομένα όσο και για εντολές. Δεν συμβαίνει όμως αυτό για όλους τους επεξεργαστές. Υπάρχουν αρχιτεκτονικές με διαφορετική μνήμη για κάθε δραστηριότητα.

Ανάγνωση δεδομένων: Η ανάγνωση δεδομένων γίνεται από τον καταχωρητή *darard*. Η μνήμη ανακτά στον καταχωρητή *datard* τα δεδομένα που βρίσκονται στην διεύθυνση *addr* όταν το σήμα *re* (read enable) γίνεται λογικό-1.

Ανάγνωση εντολών: Η ανάγνωση εντολών γίνεται από τον καταχωρητή *instrrd*. Η μνήμη ανακτά στον καταχωρητή *instrrd* την εντολή που βρίσκεται στην διεύθυνση *instraddr*. Η ανάγνωση δεδομένων δεν απαιτεί enable σήμα.

Θυμίζουμε ότι *λέξη μνήμης* είναι το μέγεθος κάθε γραμμής στον πίνακα της μνήμης. Το μέγεθος λέξης της μνήμης του MicroCPU είναι 16 bits ή αλλιώς 2 bytes.

Το testbench της Μνήμης

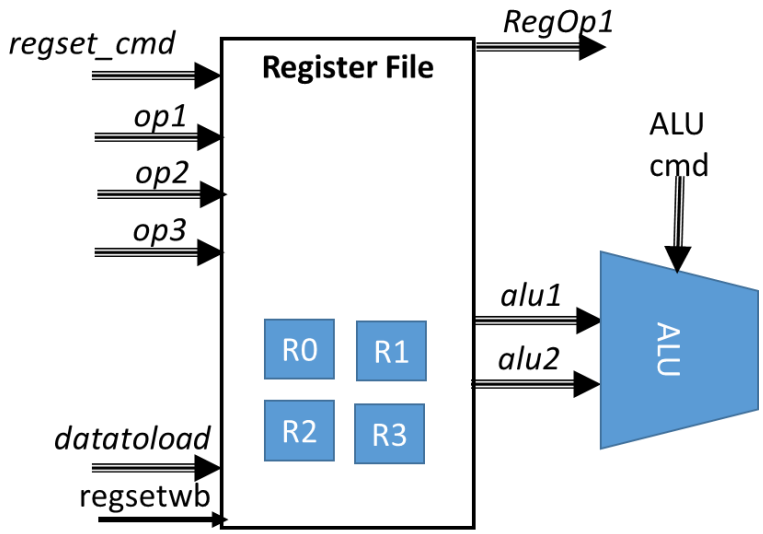
Δεν υπάρχει testbench της μνήμης. Γράψτε ένα testbench για την μνήμη το οποίο θα επιβεβαιώνει την σωστή λειτουργία των τριών διαδικασιών της μνήμης (εγγραφή δεδομένων, ανάγνωση δεδομένων και ανάγνωση εντολών). Αρχικά θα χρησιμοποιήσετε την εγγραφή δεδομένων για να γεμίσετε με τυχαίες λέξεις την μνήμη. Τα δεδομένα αυτά θα τα κρατήσετε και σε ένα τοπικό αντίγραφο στο testbench (για να ελέγξετε αργότερα να τα διαβάσατε σωστά από την μνήμη). Στην πορεία θα διαβάσετε με τις διαδικασίες ανάγνωσης δεδομένων και ανάγνωσης εντολών τις λέξεις που έχετε εγγράψει στην μνήμη και θα επιβεβαιώσετε ότι ταυτίζονται με τις τυχαίες λέξεις που γράψατε.

2.4 Αρχείο Καταχωρητών (Register File) του MicroCPU

Το αρχείο καταχωρητών είναι το module MCPU_Registerfile και θα το βρείτε στο αρχείο *registerfile.v*.

Πρόκειται για ένα σύνολο από καταχωρητές, μαζί με λογική ελέγχου ανάγνωσης/εγγραφής τους, το οποίο υλοποιείτε από flip-flops για είναι ταχύτατοι στην λειτουργία τους. Χρησιμοποιούνται για να αποθηκεύονται προσωρινά τα δεδομένα των εντολών που εκτελούνται από τον MicroCPU.

Επειδή συνδέονται στενά με τη διαδικασία αποκωδικοποίησης (Decoding) της Μονάδας Ελέγχου, το αρχείο καταχωρητών θα μας απασχολήσει και στο σύνολο-εντολών (Instruction Set) του επεξεργαστή.



Ένα σχεδιάγραμμα εισόδων/εξόδων του αρχείου καταχωρητών φαίνεται στο παραπάνω σχήμα.

Καταχωρητές: έχει 4 καταχωρητές, τους R[0], R[1], R[2] και R[3]. Δεν έχουν όλοι οι μικροεπεξεργαστές μόνο 4 καταχωρητές. Συνήθως έχουν τουλάχιστον 16.

```
//this processor has 4 registers
reg [WORD_SIZE-1:0] R[REGISTERS_NUMBER-1:0];
//this processor has 4 registers //aliases
```

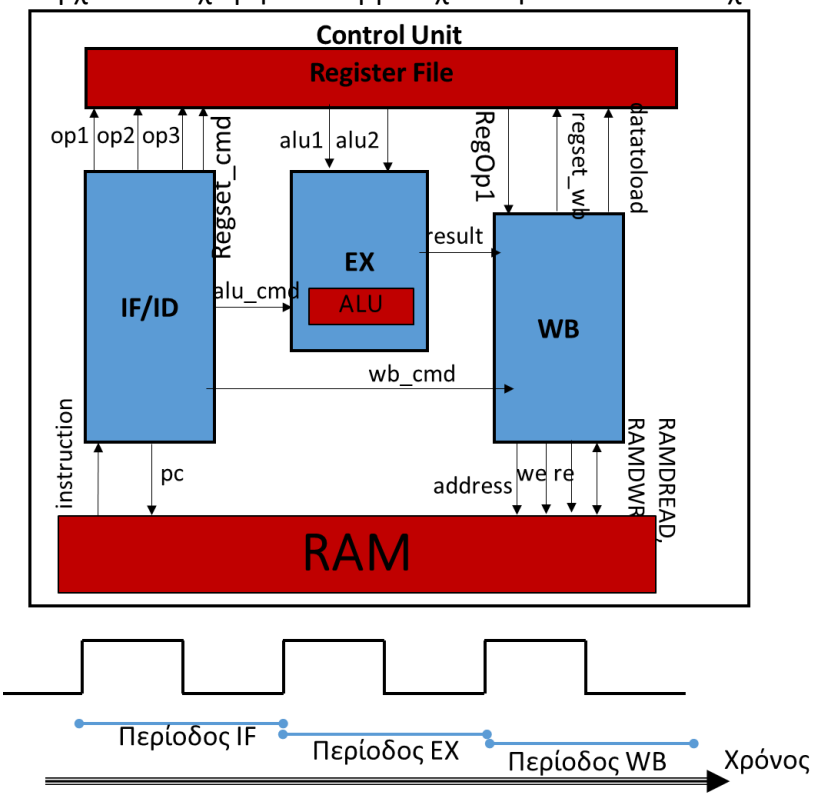
regset_cmd: στον καταχωρητή αυτό η control unit προγραμματίζει το register file κατά το instruction fetch (STATE_IF) γράφοντας την εντολή που πρέπει το register file να εκτελέσει κατά το writeback (STATE_WB). Το WB το αντιλαμβάνεται το register file γιατί τότε το σήμα *regset_wb* γίνετε λογικό-1. Λεπτομέρειες στην παράγραφο περιγραφής συμπεριφοράς (Behavioural) του ακολουθιακού τμήματος του αρχείου καταρητών.

op1, op2, op3: τελεστές που εμπλέκονται στις εντολές. Λεπτομέρειες στην παράγραφο τελεστών και στην δομική περιγραφή του αρχείου καταχωρητών.

RegOp1, alu1, alu2: τελεστέοι που εμπλέκονται στις εντολές. Λεπτομέρειες στην δομική περιγραφή του αρχείου καταχωρητών.

Τελεστέοι (Operands): οι εντολές που εκτελεί ο MicroCPU συνήθως εμπλέκουν τα δεδομένα που υπάρχουν σε κάποιον καταχωρητή. Μπορεί να εμπλέκουν και περισσότερους καταχωρητές. Συγκεκριμένα, μια εντολή χρησιμοποιεί το πολύ 3 καταχωρητές. Η κωδικοποίηση των

καταχωρητών που απαιτούνται από μια εντολή γίνεται με την χρήση των τριών operands: op1, op2 και op3. Για να καταλάβουμε όμως τους operands, πρέπει να δούμε τον τρόπο με τον οποίο το αρχείο καταχωρητών συμμετέχει στη διαδικασία διοχέτευσης.



Συμμετοχή του register file στη διοχέτευση του MicroCPU

Το αρχείο καταχωρητών συμμετέχει σε όλα τα στάδια της διοχέτευσης του MicroCPU. Αρχικά, κατά το Fetching αναλαμβάνει μέρος της αποκωδικοποίησης των εντολών (Instruction Decode - ID). Συγκεκριμένα, δέχεται τους operands των εντολών και αναλαμβάνει να αποκωδικοποιήσει σε ποιους καταχωρητές αναφέρονται.

Παράδειγμα συμμετοχής του αρχείου καταχωρητών στην αποκωδικοιποίηση μιας εντολής/assembly:

ADD 3 2 1

Η συγκεκριμένη εντολή assembly του MicroCPU έχει τρεις operands με τιμή 3, 2 και 1. Αυτό αποκωδικοποιείτε στο εξής: οι καταχωρητές R2 και R1 θα δοθούν στην ALU και το αποτέλεσμα θα γραφτεί στον καταχωρητή R3.

Επίσης, το αρχείο καταχωρητών είναι αυτό που προωθεί καταχωρητές στην ALU για την εκτέλεση των εντολών (Execute - EX) της μορφής:

Εντολή op1 op2 op3

Το επιτυγχάνει αντιγράφοντας στους καταχωρητές alu1 και alu2 τους καταχωρητές που καταδεικνύουν οι operands op2 και op3.

Όμως, την δομική διασύνδεση των σημάτων alu1, alu2 με τις εισόδους της ALU θα την αναλάβει, όπως θα δούμε, η μονάδα ελέγχου του MicroCPU.

Δομική Περιγραφή (Structural) του Συνδυαστικού τμήματος του αρχείου καταχωρητών

Η περιγραφή αυτή αφορά κυρίως την ανάγνωση των καταχωρητών. Το αρχείο καταχωρητών λοιπόν αναλαμβάνει πάντα να αποκωδικοποιήσει τους operands ειδόδου σε καταχωρητές. Αυτό το πετυχαίνει το παρακάτω δομικό/structural κομμάτι του κώδικα περιγραφής:

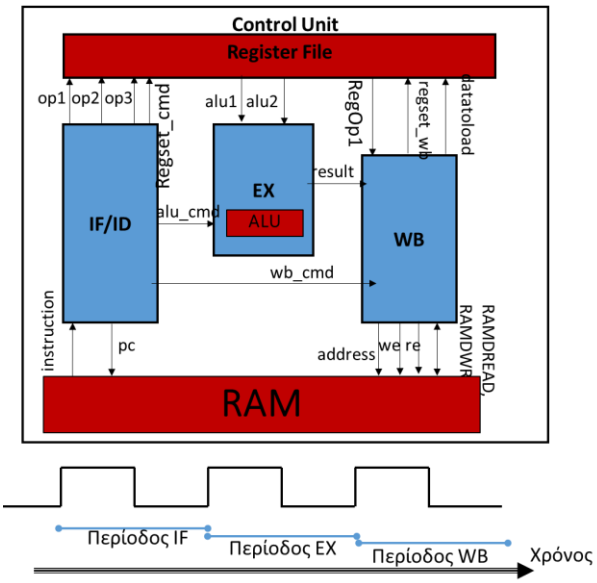
```
assign RegOp1=R[op1];
assign alu1=R[op2];
assign alu2=R[op3];
```

Άρα οι έξοδοι RegOp1, alu1 και alu2 είναι η αποκωδικοποίηση/μετάφραση των operands op1, op2 και op3, αντίστοιχα.

Περιγραφή συμπεριφοράς (Behavioural) του ακολουθιακού τμήματος του αρχείου καταχωρητών

Η περιγραφή αυτή αφορά την εγγραφή των καταχωρητών και το μπλοκ κώδικα είναι το παρακάτω:

```
input [1:0] regsetcmd;
input regsetwb;
//REGISTER FILE COMMAND (regsetcmd control bits)
parameter  [1:0]  NORMAL_EX  = 0; //2'b00
parameter  [1:0]  MOV_INTERNAL   = 1; //2'b01
parameter  [1:0]  LOAD_FROM_DATA   = 2; //2'b10
parameter  [1:0]  DO_NOTHING   = 3; //2'b11
//whenever this unit needs to act - this signal
//is asserted at WB from the control unit
always @(posedge regsetwb)
begin
 #1   //some delay
 case(regsetcmd)
 NORMAL_EX, LOAD_FROM_DATA:
 begin
  R[op1[REGS_NUMBER_WIDTH-1:0]]<=datatoload;
 end
 MOV_INTERNAL:
 begin
  R[op1[REGS_NUMBER_WIDTH-1:0]]<=R[op2[REGS_NUMBER_WIDTH-1:0]];
 end
 default:
 begin
 end
 endcase
end
endmodule
```



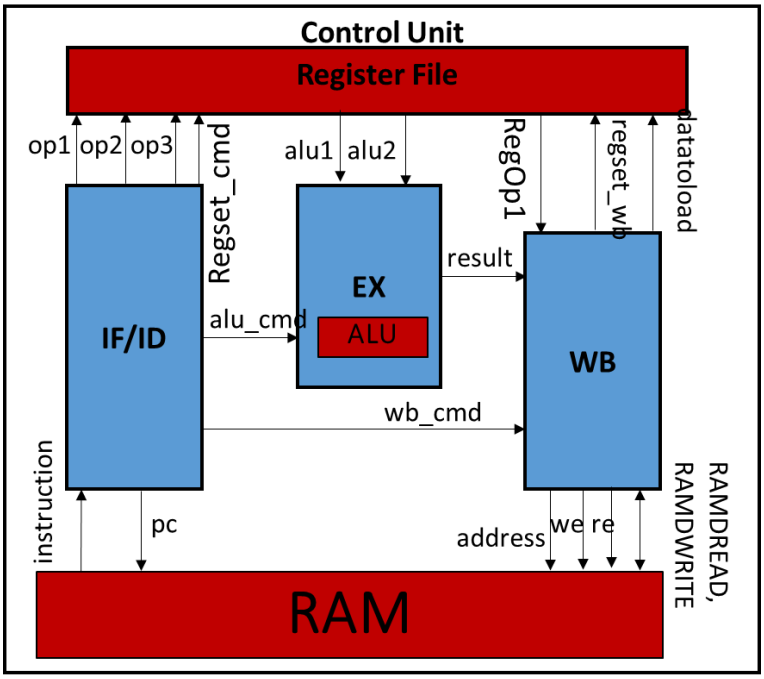
Όπως και οι υπόλοιπες μονάδες, το αρχείο καταχωρητών δέχεται μια εντολή με το σήμα regset_cmd. Μπορεί να πάρει 4 ειδικές περιπτώσεις: NORMAL_EX, MOV_INTERNAL, LOAD_FROM_DATA και DO_NOTHING. Κάθε μια από τις περιπτώσεις λέει στο αρχείο καταχωρητών τι διαδικασία που πρέπει να εκτελέσει όταν δεχτεί λογικό-1 στο enable σήμα *regsetwb*. Οι εντολές για το αρχείο καταχωρητών είναι επί της ουσίας 3:

DO_NOTHING: το προφανές, δεν κάνει καμία διαδικασία εγγραφής.

NORMAL_EX, LOAD_FROM_DATA: γράφει στον καταχωρητή που αναφέρεται ο operand1 τα δεδομένα από εξωτερικά δεδομένα που δέχεται με την είσοδο datatoload. Υπάρχουν 2 περιπτώσεις γιατί στην μια περίπτωση:
NORMAL_EX: παίρνει δεδομένα (το datatoload) από την ALU.
LOAD_FROM_DATA: Ενώ στην άλλη στην άλλη δέχεται δεδομένα από την μνήμη.
MOV_INTERNAL: στην περίπτωση αυτή μετακινεί δεδομένα ανάμεσα σε καταχωρητές εσωτερικά στο αρχείο καταχωρητών, δηλαδή αγνοεί το σήμα datatoload που φέρνει εξωτερικά δεδομένα.

2.5 Μονάδα Ελέγχου (Control Unit) του MicroCPU

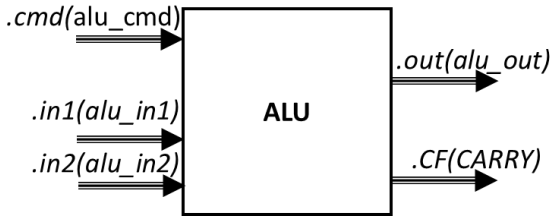
Η μονάδα ελέγχου (control unit) είναι το module MCPU και θα το βρείτε στο αρχείο *cpu.v*.
Η μονάδα ελέγχου είναι υπεύθυνη για την αλλαγή των καταστάσεων της διοχέτευσης και την κατάλληλη σηματοδότηση των μονάδων (ALU, registerfile, RAM) την κατάλληλη χρονική στιγμή. Εμπεριέχει ως αντικείμενα όλα τα δομικά modules του MicroCPU. Είναι επίσης το top module του MicroCPU.



2.5.1 Δημιουργία instances των modules από την control unit

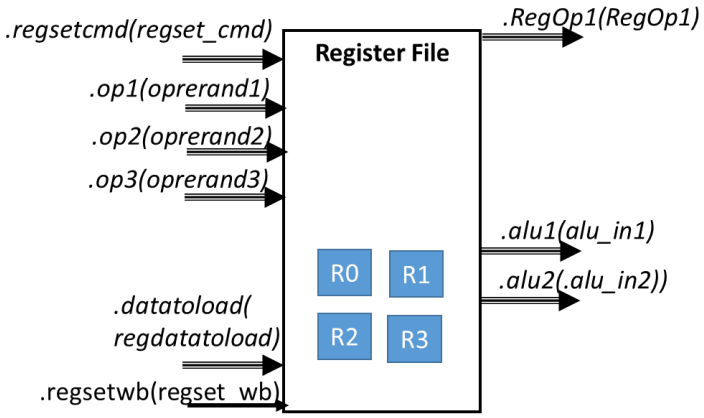
Δημιουργία instance της ALU

```
//control signals for ALU
wire [WORD_SIZE-1:0] alu_in1;
wire [WORD_SIZE-1:0] alu_in2;
wire [WORD_SIZE-1:0] alu_out;
wire CARRY;
wire [ALU_CMD_SIZE-1:0] alu_cmd;
MCPU_Alu    #(.CMD_SIZE(ALU_CMD_SIZE),
.WORD_SIZE(WORD_SIZE))
aluinst (.cmd(alu_cmd),
         .in1(alu_in1),
         .in2(alu_in2),
         .out(alu_out),
         .CF(CARRY));
```



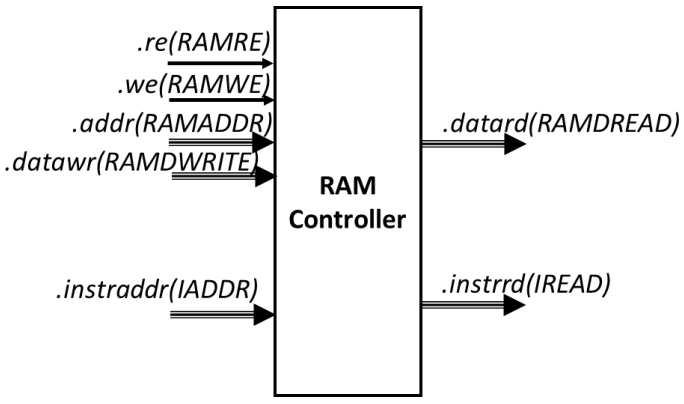
Δημιουργία instance του register file

```
//The Program Counter
reg [ADDR_WIDTH-1:0] pc;
//Control Signals for Register file
reg [1:0] regset_cmd;
reg regset_wb;
wire                [WORD_SIZE-1:0]
regdatatoload;
wire [WORD_SIZE-1:0] RegOp1;

MCPU_Registerfile
#(.WORD_SIZE(WORD_SIZE),
.OPERAND_SIZE(OPERAND_SIZE))
regfileinst (.op1(operand1),
             .op2(operand2),
             .op3(operand3),
             .RegOp1(RegOp1),
             .alu1(alu_in1),
             .alu2(alu_in2),

.datatoload(regdatatoload),
             .regsetwb(regset_wb),

.regsetcmd(regset_cmd));
```



Δημιουργία instance της μνήμης

```
//Control signals for RAM
reg RAMWE, RAMRE;
reg [ADDR_WIDTH-1:0] RAMADDR;
wire [WORD_SIZE-1:0] RAMDWRITE;
wire [WORD_SIZE-1:0] RAMDREAD;

wire [ADDR_WIDTH-1:0] IADDR;
wire [WORD_SIZE-1:0] IREAD;

MCPU_RAMController
#(.WORD_SIZE(WORD_SIZE),
.ADDR_WIDTH(ADDR_WIDTH))
raminst (.we(RAMWE),
         .datawr(RAMDWRITE),
         .re(RAMRE),
         .addr(RAMADDR),
         .datard(RAMDREAD),
         .instraddr(IADDR),
         .instrrd(IREAD));
```



2.5.2 Δομική Σχεδίαση του Συνδυαστικού τμήματος της μονάδας ελέγχου

```
assign IADDR=pc;
//instruction is always read from the
//IREAD channel pf memory
wire [INSTRUCTION_SIZE-1:0] instruction;
assign instruction=IREAD;
```

Η βασική λειτουργία αυτού του τμήματος είναι η αποκωδικοποίηση των εντολών που διαβάζονται από την μνήμη.
Αρχικά η διεύθυνση ανάγνωση εντολής από την μνήμη συνδέεται με τον program counter pc. Επομένως, το τρέχων instruction θα είναι πάντα στα IREAD, το οποίο συνδέεται με το instruction.

```
//structural code for instruction decoding
assign opcode=
instruction[INSTRUCTION_SIZE-1:INSTRUCTION_SIZE-
OPCODE_SIZE];
assign alu_cmd=opcode[ALU_CMD_SIZE-1:0];
assign operand1=instruction[OPCODE_SIZE*3-1:2*OPCODE_SIZE];
assign operand2=instruction[OPCODE_SIZE*2-1:OPCODE_SIZE];
assign operand3=instruction[OPCODE_SIZE-1:0];
```

Toinstruction είναι η εντολή που διαβάζεται από την μνήμη, η οποία σπάει σε 4 στοιχεία των 4ρων bits, τα οποία είναι: το opcode και τα 3 operands,(operand1, operand2, operand3).

```
wire [WORD_SIZE-1:0] MemOrConstant;
assign MemOrConstant=
(opcode==OP_SHORT_TO_REG)?
{8'b00000000, operand2, operand3}:RAMDREAD;
assign regdatatoload=
(regset_cmd==regfileinst.NORMAL_EX)?alu_out:MemOrConstant;
```

Εδώ σχεδιάζεται το κύκλωμα που γράφει τα δεδομένα στο σήμα regdatatoload το οποίο έχει τα δεδομένα που γράφονται στο registerfile. Στο registerfile γράφονται είτε όταν εκτελούνται εντολές επεξεργασίας όπως είναι οι ADD, OR, XOR κτλ (το καταλαβαίνει από το NORMAL_Ex ότι πρόκειται για τέτοιες εντολές) είτε όταν γράφονται σταθερές των 8 bits ως **superoperand** από το instruction προς τους καταχωρητές (το καταλαβαίνει από το opcode== OP_SHORT_TO_REG).

```
 //only data from operand 1 decoding to register are ever written
into memory
 assign RAMDWRITE=RegOp1;
```

Δεδομένα στην μνήμη μέσω του κανονικού καναλιού εγγραφής δεδομένων γράφονται κατά τη λειτουργία του επεξεργαστή, γράφονται μόνο από τον καταχωρητή που βάζουμε στη θέση operand1. Π.χ.
STORE_TO_MEM Rd address
Όπως είναι τώρα ο σχεδιασμός αυτή είναι και η μόνο εντολή που γράφει δεδομένα από τον operand1 προς την μνήμη

2.5.3 Περιγραφική σχεδίαση του Συνδυαστικού τμήματος της μονάδας ελέγχου

```
//parameter CPU_STATES_BITS=2;
//Instruction Fetch State
parameter  [1:0]  IF_STATE  = 2'b00;
//Execute Fetch State
parameter  [1:0]  EX_STATE  = 2'b01;
//WriteBack State
parameter  [1:0]  WB_STATE  = 2'b10;
//HALTED State
parameter  [1:0]  HLT_STATE  = 2'b11;
```

Αυτές είναι οι καταστάσεις λειτουργίας της μηχανής καταστάσεων της μονάδας ελέγχο. Ταυτίζονται στο σχεδιασμό μας με τα στάδια της διοχέτευσης.

```
reg [1:0] state;
reg [1:0] next_state;
```

Καταχωρητές που καταχωρούν την τρέχουσα και την επόμενη κατάσταση

```
always @ (state, opcode)
begin : MAIN_FSM_ASYCHRONOUS
 next_state=0;
 case(state)
   IF_STATE:
   begin
    //this is a 3 stages pipelining so
    //IF and DECODE occur on this step
    case(opcode)
      OP_BNZ:
      begin
         next_state = IF_STATE;
      end
      default:
      begin
         next_state = EX_STATE;
      end
    endcase
   end
   EX_STATE:
   begin
    next_state = WB_STATE;
   end
   WB_STATE:
   begin
    next_state = IF_STATE;
   end
 endcase
end
```

Το μπλοκ αυτό έχει ως αρμοδιότητα να υπολογίσει την επόμενη κατάσταση. Συνήθως η σειρά είναι IF→EX→WB→IF→EX→WB→IF… κοκ.
Όμως στην περίπτωση της εντολής BNZ η επόμενη κατάσταση κατά το Instruction Fetch είναι πάλι το Instruction Fetch.

2.5.4 Περιγραφική σχεδίαση του Ακολουθιακού Τμήματος της μονάδας ελέγχου

```
always @ (posedge clk, reset)
begin : MAIN_FSM
 if (reset == 1'b1) begin
  //get the CPU into IF state
  state <= #1 IF_STATE;
  //reset the Program Counter PC
  pc <= #1 0;

 end else begin
  case(state)
   IF_STATE:
   begin
    //this is a 3 stages pipelining so
    //IF and DECODE occur on this step
    case(opcode)
     OP_AND,OP_OR,OP_XOR,OP_ADD:
     begin
      regset_cmd <= #2 regfileinst.NORMAL_EX;
     end
     OP_MOV:
     begin
      regset_cmd <= #2 regfileinst.MOV_INTERNAL;
     end
     OP_LOAD_FROM_MEM:
     begin
      regset_cmd<= #2 regfileinst.LOAD_FROM_DATA;
      wb_cmd[ADDR_WIDTH-1:0]<= #2 {operand2,operand3};
      wb_cmd[ADDR_WIDTH]<= #2 1'b0; //RAMWE
      wb_cmd[ADDR_WIDTH+1]<= #2 1'b1; //RAMRE
     end

     OP_STORE_TO_MEM:
     begin
      regset_cmd <= #2 regfileinst.DO_NOTHING;
      //whatever there is in RAMDWRITE,
      //it is going to be written at WB
     //to address {operand2,operand3}
      wb_cmd[ADDR_WIDTH-1:0] <= #2 {operand2,operand3};
       //RAMWE - RAM WRITE ENABLE
      wb_cmd[ADDR_WIDTH] <= #2 1'b1;
       //RAMRE - RAM READ ENABLE
      wb_cmd[ADDR_WIDTH+1] <= #2  1'b0;
     end

    OP_SHORT_TO_REG:
     begin
      regset_cmd<=#2 regfileinst.LOAD_FROM_DATA;
     end

     OP_BNZ:
     begin
      if(RegOp1!=0)
      begin
       pc <= #5 {operand2, operand3};
      end else
      begin
```

Αυτό το μπλοκ είναι η καρδιά της μονάδας ελέγχου και είναι σύγχρονο στο reset και στο ρολόι clk του MicroCPU. Αρχικά διαχειρίζεται το σήμα reset μηδενίζοντας τον program counter pc και αρχικοποιόντας την μηχανή καταστάσεων στο IF_STATE όταν το reset ενεργοποιείται.

Όταν το reset είναι λογικό-0, τότε διαχειρίζεται σε κάθε κύκλο πως θα συμπεριφερθεί η μονάδα ελέγχου ανάλογα με την κατάστασή στην οποία βρίσκεται.

Έτσι:
όταν η τρέχουσα εντολή είναι κάποια εντολή επεξεργασίας τότε προγραμματίζει το αρχείο καταχωρητών κατάλληλα (ώστε οι καταχωρητές να περιμένουν να γράψουν το αποτέλεσμα που θα τους έρθει από την ALU κατά το WB)
όταν η τρέχουσα εντολή είναι MOV τότε προγραμματίζει το αρχείο καταχωρητών κατάλληλα (ώστε να κάνει εσωτερική μεταφορά δεδομένων κατά το WB)

όταν η τρέχουσα εντολή είναι LOAD_FROM_MEM τότε προγραμματίζει το αρχείο καταχωρητών να περιμένει δεδομένα από την μνήμη κατά το WB. Παράλληλα ετοιμάζει και την εντολή wb_cmd που θα εκτελέσει το wb. Στο wb_cmd τοποθετεί τη διεύθυνση μνήμης η οποία θα αναγνωστεί και θέτει το RAMRE σε λογικό-1.

όταν η τρέχουσα εντολή είναι STORE_TO_MEM τότε προγραμματίζει το αρχείο καταχωρητών ότι δεν πρόκειται να γίνει εγγραφή στους καταχωρητές κατά το WB. Παράλληλα ετοιμάζει και την εντολή wb_cmd που θα εκτελέσει το wb. Στο wb_cmd τοποθετεί τη διεύθυνση μνήμης στην οποία θα γίνει η εγγραφή και θέτει το RAMWE σε λογικό-1.

όταν η τρέχουσα εντολή είναι SHORT_TO_REG τότε προγραμματίζει το αρχείο καταχωρητών να περιμένει εξωτερικά δεδομένα κατά το WB. Το συνδυαστικό τμήμα (Παράγραφος 2.5.2) διαχειρίζεται αυτά τα σήματα για να τροφοδοτήσει με τον superoperand τα εξωτερικά δεδομένα του αρχείου καταχωρητών.

όταν η τρέχουσα εντολή είναι BNZ τότε τσεκάρει τον καταχωρητή που υποδεικνύεται από τον operand1 και αν η τιμή του δεν είναι 0 τότε τοποθετεί τον program counter pc στην τιμή των 8 bits του superoperand των {operand2, operand3}.

```
        pc <= #5 pc+1;
       end
      end
      default:
      begin
      end
     endcase
    end

    EX_STATE:
    begin
    end

    WB_STATE:
    begin
     RAMADDR<=#1 wb_cmd[ADDR_WIDTH-1:0];
     RAMWE<=#1 wb_cmd[ADDR_WIDTH];
     RAMRE<=#1 wb_cmd[ADDR_WIDTH+1];

     regset_wb <= #2 1'b1;
     regset_wb<= #3 1'b0;

     wb_cmd[ADDR_WIDTH]<= #3 1'b0;
     wb_cmd[ADDR_WIDTH+1]<=  #3 1'b0;
     RAMWE <= #3 1'b0;
     RAMRE <= #3 1'b0;


     pc <= #5 pc+1;

    end
    HLT_STATE:
    begin
     $display("processor HALTED\n");
     $stop;
    end
    default:
    begin
    end

   endcase
   state<=#8 next_state;
  end

end
```

Στο EX_STATE δεν κάνει κάτι στο ακολουθιακό κύκλωμα, γιατί ό,τι πράξεις είναι να συμβούν, θα συμβούν από την συνδυαστική λογική.

Στην ουσία αποκωδικοποιεί την εντολή wb_cmd που δέχτηκε από το IF_STATE και θέτει τα σήματα της μνήμης ανάλογα..

Στην πορεία θέτει το enable του καταχωρητή αρχείων (regset_wb) για να εκτελέσει και αυτό τις εντολές του.

Έπειτα σταματάει κάθε δραστηριότητα της μνήμης

Στο τέλος αυξάνει τον program counter κατά 1

Αν βρεθεί σε αυτήν την κατάσταση τότε ο MicroCPU έχει κολλήσει

Αφού εκτελέσει τις ακολουθιακές δραστηριότητές του ανάλογα με το state που βρίσκεται, τότε μεταβαίνει στην επόμενη κατάσταση. Θα μπορούσαμε να περιμένουμε μια ολόκληρη περίοδο πριν μεταβούμε στην επόμενη κατάσταση, ωστόσο μιας και το ακολουθιακό κομμάτι είναι ακμοπυροδότητο στο ρολόι, είναι θεμιτό να έχει ήδη μπει στο επόμενο state πριν έρθει η επόμενη ακμή του ρολογιού. Η επόμενη ακμή πρόκειται να έρθει σε 10ps. Έτσι επιλέγουμε να αλλάξουμε κατάσταση λίγο νωρίτερα, στα 8ps, για να είναι ήδη στην επόμενη κατάσταση η μηχανή καταστάσεων όταν έρθει η επόμενη ακμή του clk.