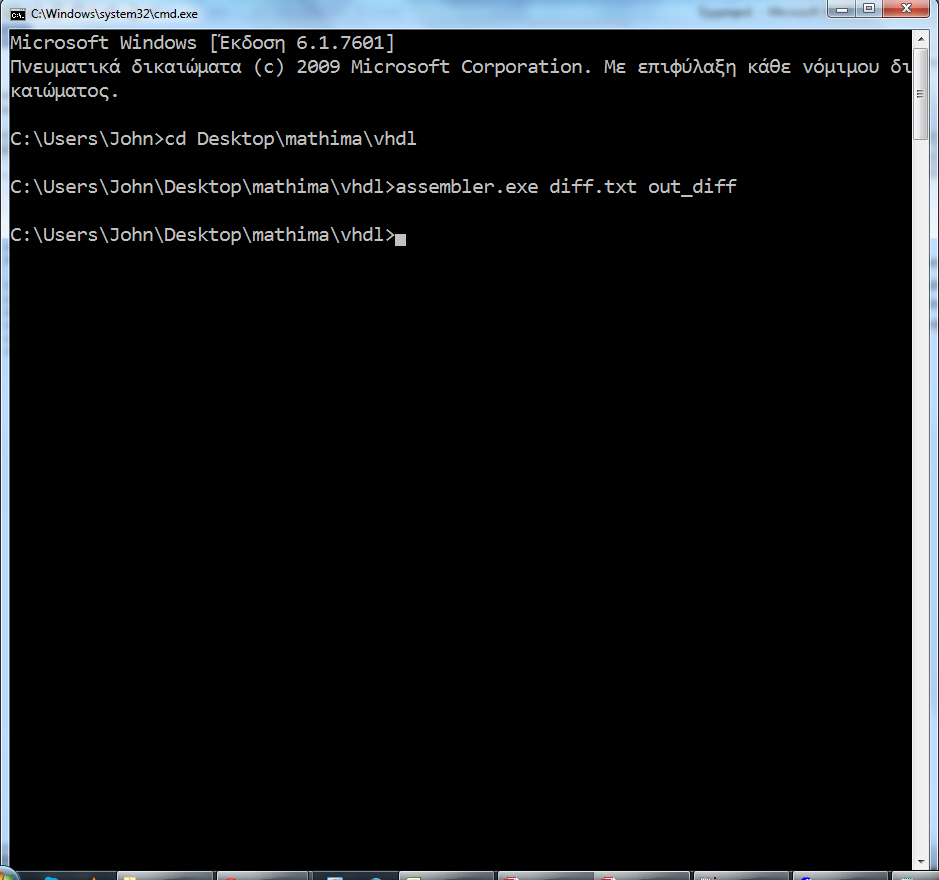
**Οδηγός Χρήσης Assembler και Modelsim/Quartus**  
O assembler δίνεται τόσο σε κώδικα C, ώστε να γίνει compile εάν χρειαστεί σε κάποιον υπολογιστή τοπικά, όσο και σε εκτελέσιμο exe για πιο γρήγορη χρήση. Για να τρέξει ο assembler μέσω command line πηγαίνουμε και εκτελείτε ως:  
**assembler.exe**<whitespace>**assembly\_file**<whitespace>**hex\_output**

Πχ, αν το αρχείο μας με τον κώδικα ήταν το diff.txt η εντολή θα είχε την παρακάτω μορφή.

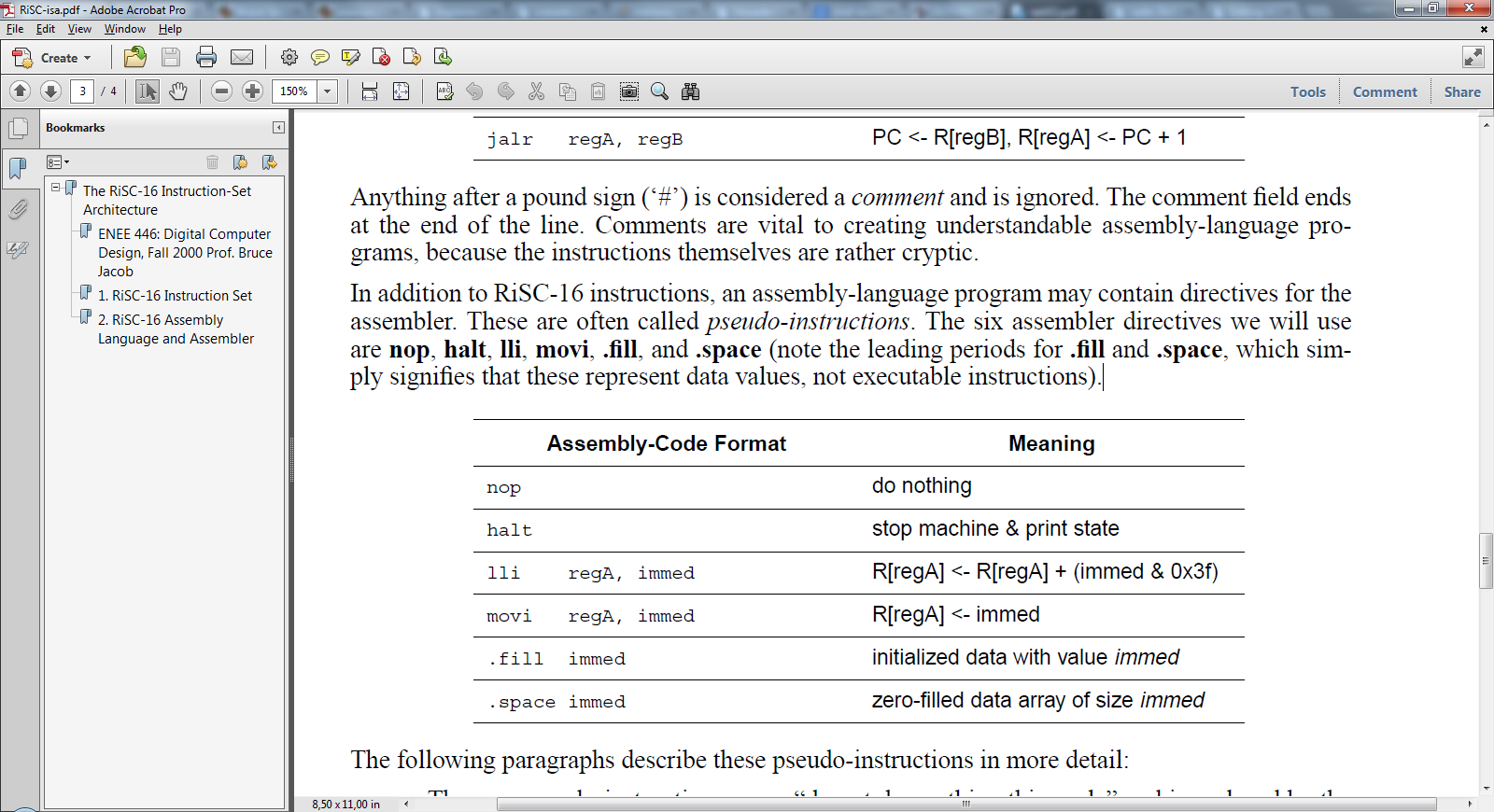


Η εντολές στο αρχείο με τον κώδικα assembly θα έχουν την παρακάτω μορφή:  
**label:**<whitespace>**opcode**<whitespace>**field0**, **field1**, **field2**<whitespace># **comments**

The leftmost field on a line is the label field. Valid labels are any combination of letters and numbers followed by a colon. The colon at the end of the label is not optional – a label without a colon is interpreted as an opcode. After the optional label is whitespace (space/s or tab/s). Then follows the opcode field, where the opcode can be any of the assembly-language instruction mnemonics listed in the above table. After more whitespace comes a series of fields separated by commas and possibly whitespace (you need to have either whitespace or a comma or both in between each field). All register-value fields are given as **decimal** numbers, optionally preceded by the letter ‘r’ ... as in r0, r1, r2, etc. Immediate-value fields are given in either decimal, octal, or hexadecimal form. Octal numbers are preceded by the character ‘0’ (zero). For example, 032 is interpreted as the octal number ‘oh-three-two’ which corresponds to the decimal number 26. It is *not* interpreted as the decimal number 32. Hexadecimal numbers are preceded by the string ‘0x’ (oh-x). For example, 0x12 is ‘hex-one-two’ and corresponds to the decimal number 18, not decimal 12.

Anything after a pound sign (‘#’) is considered a *comment* and is ignored. The comment field ends at the end of the line. Comments are vital to creating understandable assembly-language programs, because the instructions themselves are rather cryptic.

In addition to instructions, an assembly-language program may contain directives for the assembler. These are often called *pseudo-instructions*. The six assembler directives we will use are **nop**, **halt**, **lli**, **movi**, **.fill**, and **.space** (note the leading periods for **.fill** and **.space**, which simply signifies that these represent data values, not executable instructions).



The following paragraphs describe these pseudo-instructions in more detail:

**•** The **nop** pseudo-instruction means “do not do anything this cycle” and is replaced by the instruction **add 0,0,0** (which clearly does nothing).

**•** The **halt** pseudo-instruction means “stop executing instructions and print current machine state” and is replaced by **jalr 0, 0** with a non-zero immediate field.

**•** The **lli** pseudo-instruction (*load-lower-immediate*) means “OR the bottom six bits of this number into the indicated register” and is replaced by **addi X,X,imm6**, where **X** is the register specified, and **imm6** is equal to **imm & 0x3f**. This instruction can be used in conjunction with **lui**: the **lui** first moves the top ten bits of a given number (or address, if a label is specified) into the register, setting the bottom six bits to zero; the **lli** moves the bottom six bits in. The six-bit number is guaranteed to be interpreted as positive and thus avoids signextension; therefore, the resulting **addi** is essentially a concatenation of the two bitfields.

**•** The **movi** pseudo-instruction is just shorthand for the **lui+lli** combination. Note, however, that the **movi** instruction *looks* like it only represents a single instruction, whereas in fact it represents two. This can throw off your counting if you are expecting a certain distance between instructions. Thus, it is always a good idea to use labels wherever possible.

**•** The **.fill** directive tells the assembler to put a number into the place where the instruction would normally be stored. The **.fill** directive uses one field, which can be either a numeric value or a symbolic address. For example, “.fill 32” puts the value 32 where the instruction would normally be stored. Using **.fill** with a symbolic address will store the address of the label. In the example below, the line “.fill start” will store the value 2, because the label “start” refers to address 2.

**•** The **.space** directive takes one integer **n** as an argument and is replaced by **n** copies of “.fill 0” in the code; i.e., it results in the creation of **n** 16-bit words all initialized to zero.

In general, acceptable assembly code is one-instruction-per-line. It **is** okay to have a line that is blank, whether it is commented out (i.e., the line begins with a pound sign) or not (i.e., just a blank line). However, a label **cannot** appear on a line by itself; it must be followed by a valid instruction on the same line (a **.fill** directive or **halt**/**nop**/**etc** counts as an instruction).

Note that the 8 basic instructions of the RiSC-16 architecture form a complete ISA that can perform arbitrary computation. For example:

**• Moving constant values into registers.** The number 0 can be moved into any register in one cycle (add rX r0 r0). Any number between -64 and 63 can be placed into a register in one operation using the ADDI instruction (addi rX r0 number). And, as mentioned, any 16-bit number can be moved into a register in two operations (**lui**+**lli**).

**• Subtracting numbers.** Subtracting is simply adding the negative value. Any number can be made negative in two instructions by flipping its bits and adding 1. Bit-flipping can be done by NANDing the value with itself; adding 1 is done with the ADDI instruction. Therefore, subtraction is a three-instruction process. Note that without an extra register, it is a destructive process.

**• Multiplying numbers.** Multiplication is easily done by repeated addition, bit-testing, and left-shifting a bitmask by one bit (which is the same as an addition with itself).

The following is an assembly-language program that counts down from 5, stopping when it hits 0.

lw 1,0,count # load reg1 with 5 (uses symbolic address)

lw 2,1,2 # load reg2 with -1 (uses numeric address)

start: add 1,1,2 # decrement reg1 -- could have been addi 1,1,-1

beq 0,1,1 # goto end of program when reg1==0

beq 0,0,start # go back to the beginning of the loop

done: halt # end of program

count: .fill 5

neg1: .fill -1

startAddr: .fill start # will contain the address of start (2)

Modelsim και φόρτωση της μνήμης  
  
Για το Modelsim θα χρησιμοποιήσουμε το κατάλληλο κομμάτι του κώδικα από το αρχείο για τη μνήμ. Για να φορτώσουμε τις εντολές χρησιμοποιούμε το αρχείο που προέκυψε από τον assembler (αλλάζουμε το όρισμα στον κώδικα εφόσον χρειάζεται). Το μέγεθος του block της μνήμης είναι με 256 γραμμές, οπότε μπορεί να δεχθεί συνολικά 256 διαφορετικές εντολές.

Για την προσομοίωση θα διαβάζουμε μόνο από τη μνήμη (δε θα χρησιμοποιήσουμε τη μέθοδο εγγραφής στη μνήμη). Οπότε, αλλάζουμε σε διαδοχικούς κύκλους τη διεύθυνση μνήμης που διαβάζουμε (αύξηση του program counter) και μελετάμε τα δεδομένα εξόδου.

Ο κώδικας της μνήμης τοποθετεί τις εντολές ξεκινώντας από τη διεύθυνση 0.

Δίνεται ο κώδικας σε HEX μορφή από το παραπάνω assembly πρόγραμμα ως παράδειγμα.

Στο Modelsim για να δούμε το block της μνήμης, επιλέγουμε από το μενού View --> Memory List (w). Με δεξί κλικ στο παράθυρο της μνήμης επιλέγουμε το μενού **Properties** για να αλλάξουμε τo Radix για τις διευθύνσεις και τα δεδομένα.  
Για να αποθηκεύσουμε τη μνήμη επιλέγουμε από το μενού File --> Export --> Memory Data.

Μπορούμε να επιλέξουμε τη μορφή για τις διευθύνσεις και τα δεδομένα, καθώς και ποιες διευθύνσεις θέλουμε να επιλέξουμε. Ορίζουμε ένα όνομα και αποθηκεύεται σε αρχείο μορφής \*.mem.

Quartus και δημιουργία Memory Initialization File (.mif)

Στο Quartus για να δημιουργήσουμε ένα αρχείο που θα περιέχει τις εντολές σε HEX μορφή από τον assembler ακολουθούμε την παρακάτω διαδικασία:  
Αρχικά δημιουργούμε ένα project και σε αυτό βάζουμε τα αρχεία με τον κώδικα vhdl.

Στη συνέχεια επιλέγουμε από το μενού File --> New --> Memory Initialization File

Στη συνέχεια από το μενού View, αλλάζουμε τις επιλογές:  
Cells Per Row: πόσες εντολές ανά γραμμή θα βλέπουμε στο παράθυρο στο Quartus

Address Radix: ορίζουμε το radix για τις διευθύνσεις μνήμης (καλό είναι να επιλέξουμε decimal).

Memory Radix: ορίζουμε το radix των εντολών (το θέτουμε ίσο με hexadecimal, καθώς ο assembler μας δίνει τιμές σε HEX μορφή).

Οι εντολές θα γεμίσουν από το αρχείο που προέκυψε από τον assembler. Σε κάθε κελί θα μπει μία γραμμή του αρχείου.

Περισσότερες πληροφορίες μπορούν να βρεθούν στις παρακάτω διευθύνσεις:

http://quartushelp.altera.com/13.0/mergedProjects/design/med/med\_pro\_med\_files.htm

http://quartushelp.altera.com/13.0/mergedProjects/design/med/med\_pro\_editing\_memory\_contents.htm

Στον κώδικα vhdl για τη μνήμη, χρησιμοποιούμε το κομμάτι που αφορά το Quartus. Προσέχουμε το όνομα του αρχείου mif που χρησιμοποιούμε (αλλάζουμε το όρισμα της παραμέτρου) και να βρίσκεται στον ίδιο φάκελο με τα αρχεία του κώδικα.