

# 11.1: Παραδείγματα assembly του MicroCPU

Vasileios Tenentes

University of Ioannina

# Outline

- Οδηγίες για το MicroCPU στο Modelsim
- Επεξήγηση του Προγράμματος Παραγωγής της ακολουθίας Fibonacci
- Επεκτασιμότητα του MicroCPU
  - Επέκταση του πλήθους καταχωρητών
  - Επέκταση του συνόλου των εντολών (παράδειγμα sub)
  - Παράδειγμα testbench για εκτέλεση αφαίρεσης με τους 16 νέους καταχωρητές
- Παραδείγματα βρόγχων
  - Βρόγχος if ( $R1 \neq \alpha$ )
  - Ειδική περίπτωση if ( $R1 \neq 1$ )
  - Έλεγχος μονά-ζυγά

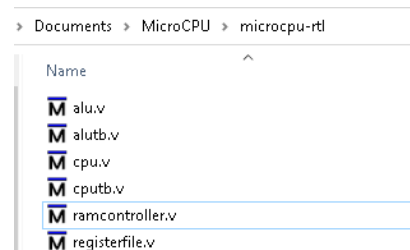
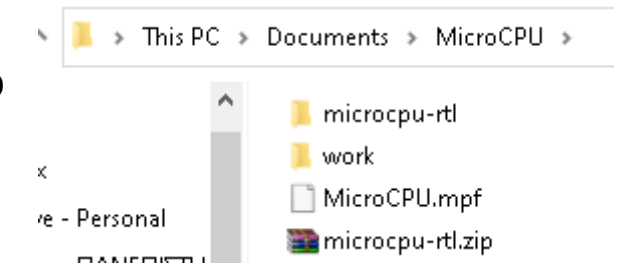
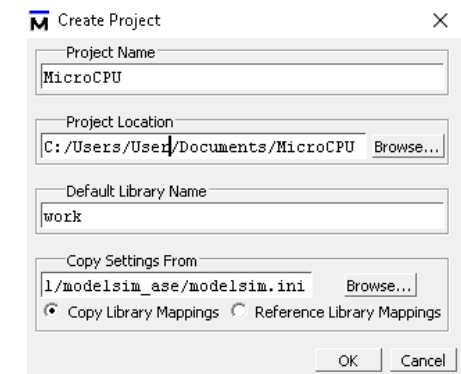
# Οδηγίες για το MicroCPU στο Modelsim

# Πως να ανοίξετε το MicroCPU στο Modelsim

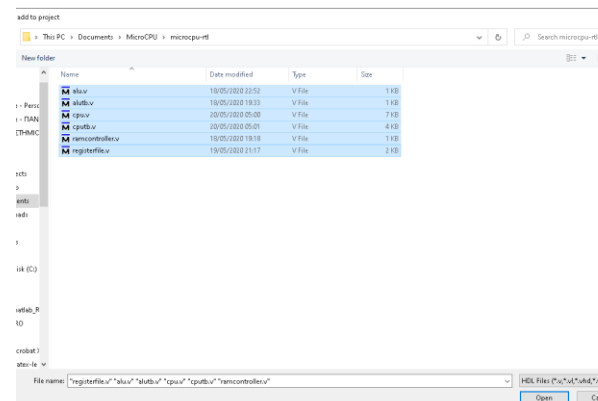
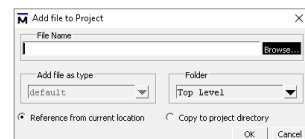
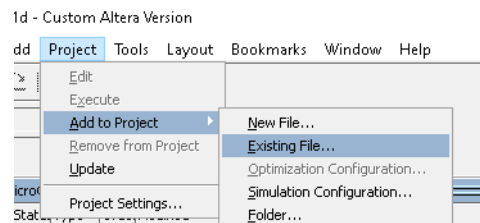
Αρχικά δημιουργείτε ένα νέο project στο Modelsim που θα ονομάσετε MicroCPU, όπως φαίνεται στην διπλανή εικόνα.

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 δίπλα



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

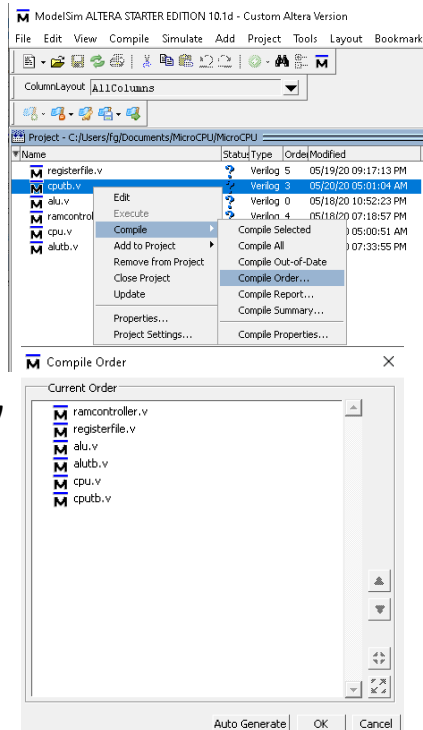


# Μεταγλώττιση του MicroCPU στο Modelsim

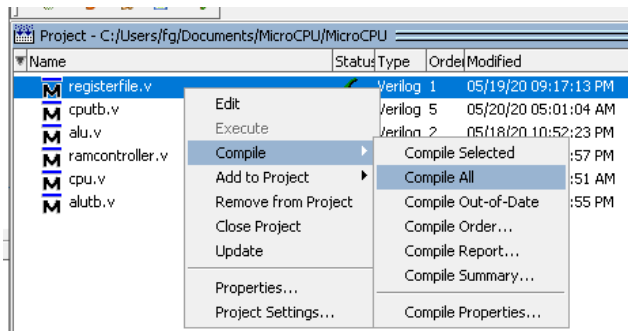
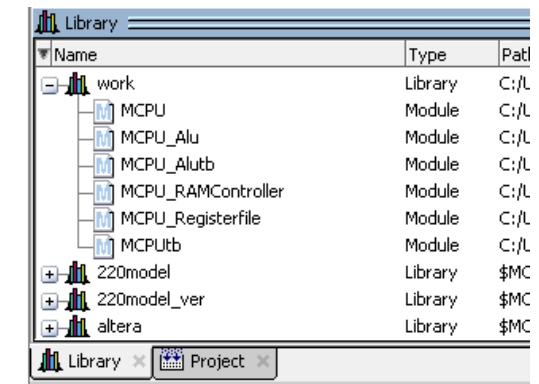
Στην συνέχεια με δεξί κλικ πάνω στα αρχεία του project μέσα από το περιβάλλον Modelsim θα επιλέξετε “compile order”

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 modules του MicroCPU είναι:

**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

# 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.

Ως παράδειγμα ακολουθεί ένα πρόγραμμα που υπολογίζει τους αριθμούς Fibonacci

# Εκτέλεση Προγράμματος στον Επεξεργαστή MicroCPU

```
module MCPUt看();  
reg reset, clk;  
MCPU cpuinst (clk, reset);
```

```
initial begin  
    reset=1;  
    #10 reset=0;  
end  
always begin  
    #5 clk=0;  
    #5 clk=1;  
End
```

```
/******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
```

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.

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.

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

# Εκτέλεση Προγράμματος στον Επεξεργαστή 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.



# Εκτέλεση Προγράμματος στον Επεξεργαστή MicroCPU

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

Στο μπλοκ αυτό αρχικά μηδενίζουμε όλες τις λέξεις της μνήμης και όλους τους καταχωρητές του MicroCPU. Θέλουμε έτσι να αποφύγουμε τα undefined Xes στην προσομοίωσή μας.

# Εκτέλεση Προγράμματος στον Επεξεργαστή MicroCPU

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

*i*=0; *cpuinst.raminst.mem*[*i*]={*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*, 8'b00010100, R2};

*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};

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)

10:...

# Εκτέλεση Προγράμματος στον Επεξεργαστή 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
```

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

# Το παράδειγμα με την ακολουθία Fibonacci

```
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};
```

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)

# Επεξήγηση του Προγράμματος Παραγωγής της ακολουθίας Fibonacci

# Το πρόγραμμα από ψευδοκώδικα σε γλώσσα μηχανής

## Ψευδοκώδικας

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)

## βοηθητική γλώσσα μηχανής (ψευδο-assembly)

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};

## γλώσσα μηχανής

0111\_0000\_0000\_0000

0111\_0001\_0000\_0001

0111\_0010\_0000\_0010

0100\_0000\_0001\_0000

0100\_0001\_0010\_0000

0011\_0010\_0000\_0001

0110\_0010\_0001\_0100

0101\_0011\_0001\_0100

0011\_0000\_0000\_0000

1000\_0010\_0000\_0011

Εμείς προγραμματίζουμε σε αυτό το στάδιο  
μέσα στο testbench

# Υπενθύμιση των opcodes

//opcodes for the supported instruction set

//Opcodes για ALU

parameter [OPCODE\_SIZE-1:0] OP\_AND = 0; //4'b0000

parameter [OPCODE\_SIZE-1:0] OP\_OR = 1; //4'b0001

parameter [OPCODE\_SIZE-1:0] OP\_XOR = 2; //4'b0010

parameter [OPCODE\_SIZE-1:0] OP\_ADD = 3; //4'b0011

//Opcodes για registers και μνήμη

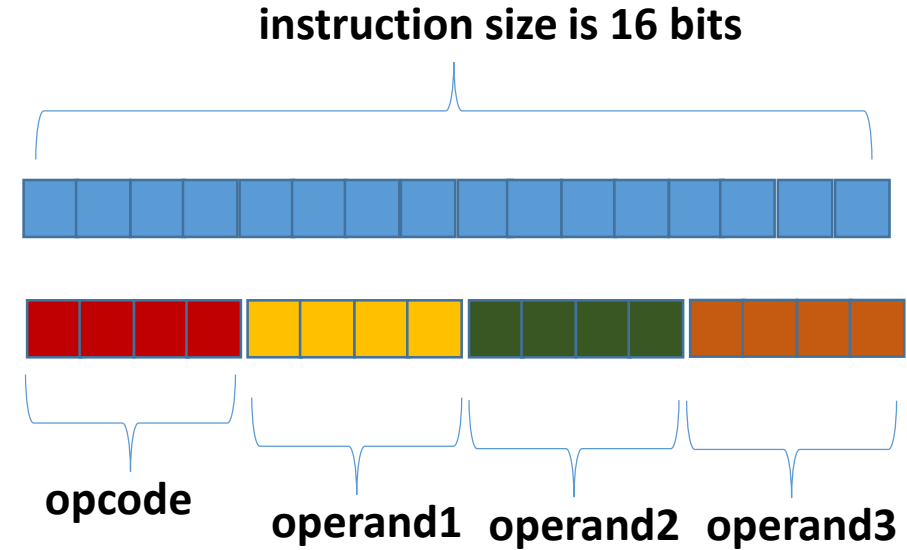
parameter [OPCODE\_SIZE-1:0] OP\_MOV = 4; //4'b0100

parameter [OPCODE\_SIZE-1:0] OP\_LOAD\_FROM\_MEM = 5; //4'b0101

parameter [OPCODE\_SIZE-1:0] OP\_STORE\_TO\_MEM = 6; //4'b0110

parameter [OPCODE\_SIZE-1:0] OP\_SHORT\_TO\_REG = 7; //4'b0111

parameter [OPCODE\_SIZE-1:0] OP\_BNZ = 8; //4'b1000



# Επεξήγηση του προγράμματος με την ακολουθία Fibonacci

Κάθε γραμμή είναι μια εντολή.

Μεταφράζω:

Θέση μνήμης: Εντολή

0: R0=0;

```
i=0; cpuinst.raminst.mem[i]={cpuinst.OP_SHORT_TO_REG, R0, 8'b0000_0000};
```

0111\_0000\_0000\_0000

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

0111\_0001\_0000\_0001

1: R1=1;

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

0111\_0010\_0000\_0010

2: R2=2;

OP\_AND = 0; //4'b0000  
OP\_OR = 1; //4'b0001  
OP\_XOR = 2; //4'b0010  
OP\_ADD = 3; //4'b0011  
//Opcodes για registers και μνήμη  
OP\_MOV = 4; //4'b0100  
OP\_LOAD\_FROM\_MEM = 5; //4'b0101  
OP\_STORE\_TO\_MEM = 6; //4'b0110  
OP\_SHORT\_TO\_REG = 7; //4'b0111  
OP\_BNZ = 8; //4'b1000



# Επεξήγηση του προγράμματος με την ακολουθία Fibonacci

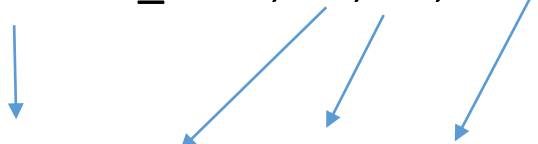
`i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_MOV, R0, R1, 4'b0000};`      **3: R0=R1;**

**0100**\_0000\_0001\_0000



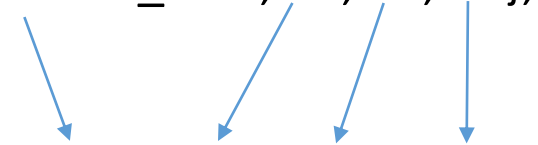
`i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_MOV, R1, R2, 4'b0000};`      **4: R1=R2;**

**0100**\_0001\_0010\_0000



`i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_ADD, R2, R0, R1};`      **5: R2=R0+R1;**

**0011**\_0010\_0000\_0001



**OP\_AND = 0; //4'b0000**  
**OP\_OR = 1; //4'b0001**  
**OP\_XOR = 2; //4'b0010**  
**OP\_ADD = 3; //4'b0011**  
**//Opcodes για registers και μνήμη**  
**OP\_MOV = 4; //4'b0100**  
**OP\_LOAD\_FROM\_MEM = 5; //4'b0101**  
**OP\_STORE\_TO\_MEM = 6; //4'b0110**  
**OP\_SHORT\_TO\_REG = 7; //4'b0111**  
**OP\_BNZ = 8; //4'b1000**

# Επεξήγηση του προγράμματος με την ακολουθία Fibonacci

`i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_STORE_TO_MEM, R2, 8'b0001_0100};`     **6:mem[20]=R2;**

0110\_0010\_0001\_0100

`i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_LOAD_FROM_MEM, R3, 8'b00010100};`     **7:R3=mem[20];**

0101\_0011\_0001\_0100

`i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_ADD, R0, R0, R0};`     **8:R0=R0+R0**

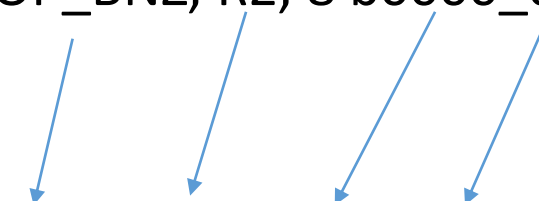
0011\_0000\_0000\_0000

**OP\_AND = 0; //4'b0000**  
**OP\_OR = 1; //4'b0001**  
**OP\_XOR = 2; //4'b0010**  
**OP\_ADD = 3; //4'b0011**  
**//Opcodes για registers και μνήμη**  
**OP\_MOV = 4; //4'b0100**  
**OP\_LOAD\_FROM\_MEM = 5; //4'b0101**  
**OP\_STORE\_TO\_MEM = 6; //4'b0110**  
**OP\_SHORT\_TO\_REG = 7; //4'b0111**  
**OP\_BNZ = 8; //4'b1000**

# Επεξήγηση του προγράμματος με την ακολουθία Fibonacci

```
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_BNZ, R2, 8'b0000_0011};
```

1000\_0010\_0000\_0011



Αυτό είναι και παράδειγμα while

```
do  
{  
3: R0=R1;  
.  
.  
.
```

```
}9:while(R2!=0)
```

```
OP_AND = 0; //4'b0000  
OP_OR  = 1; //4'b0001  
OP_XOR = 2; //4'b0010  
OP_ADD = 3; //4'b0011  
//Opcodes για registers και μνήμη  
OP_MOV = 4; //4'b0100  
OP_LOAD_FROM_MEM = 5; //4'b0101  
OP_STORE_TO_MEM = 6; //4'b0110  
OP_SHORT_TO_REG = 7; //4'b0111  
OP_BNZ = 8; //4'b1000
```

Επεκτασιμότητα του MicroCPU και  
παραδείγματα βρόγχων

# Επέκταση του πλήθους καταχωρητών

Στην δήλωση του καταχωρητή αρχείων μπορούμε να παραμετροποιήσουμε το REGS\_NUMBER\_WIDTH.

```
module MCPU_Registerfile(op1, op2, op3, RegOp1, alu1, alu2, datatoload, regsetwb, regsetcmd);  
parameter WORD_SIZE=8;  
parameter OPERAND_SIZE=4;  
parameter REGS_NUMBER_WIDTH=4;  
parameter REGISTERS_NUMBER=1<<REGS_NUMBER_WIDTH;  
.  
.  
.
```

Το regs\_number\_width μας δίνει έμμεσα τον αριθμό των καταχωρητών REGISTERS\_NUMBER, σύμφωνα με τη σχέση:

$\text{REGISTERS\_NUMBER} = 2^{\text{REGS\_NUMBER\_WIDTH}}$

Αυτό ακριβώς σημαίνει το:

```
parameter REGISTERS_NUMBER=1<<REGS_NUMBER_WIDTH;
```

Επομένως αν θέλουμε να έχουμε 16 καταχωρητές πόσο πρέπει να είναι το REGS\_NUMBER\_WIDTH? Επίσης, μην ξεχνάτε ότι οι παράμετροι γίνονται overload κατά την δημιουργία των instances από την MCPU, επομένως οι αλλαγές πρέπει να γίνουν εκεί!!!

# Εντολές μηχανής (instruction set) του MicroCPU

- **Κλασσικές εντολές επεξεργασίας** που εμπλέκουν την ALU για αριθμητικές και λογικές πράξεις, π.χ.:

*AND operand1 operand2 operand3*

*OR operand1 operand2 operand3*

*XOR operand1 operand2 operand3*

*ADD operand1 operand2 operand3*

Το αποτέλεσμα γράφεται στον operand1. Οι πράξεις εμπλέκουν δύο operands τους operand2 και operand3.

Operands για τον MicroCPU είναι μόνο οι καταχωρητές του.

- **Εντολές μετακίνησης δεδομένων από καταχωρητή σε καταχωρητή:**

*MOV Rd R*

τα δεδομένα αντιγράφονται από τον καταχωρητή R στον καταχωρητή Rd (το Rd προκύπτει από το register destination).

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

*Load\_FROM\_MEM Rd address*, για λόγους ευκολίας θα την λέω LFM

αντιγράφει τα δεδομένα από την διεύθυνση μνήμης address στον καταχωρητή Rd.

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

*STORE\_TO\_MEM address R*, για λόγους ευκολίας θα την λέω STM

αποθηκεύει στη διεύθυνση μνήμης address τα δεδομένα του καταχωρητή R.

- **Εντολές Αρχικοποίησης τιμών:**

*OP\_SHORT\_TO\_REG Rd value*, για λόγους ευκολίας θα την λέω STR

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)

**//OPCODES**

**OP\_AND = 0; //4'b0000**

**OP\_OR = 1; //4'b0001**

**OP\_XOR = 2; //4'b0010**

**OP\_ADD = 3; //4'b0011**

**//Opcodes για registers και μνήμη**

**OP\_MOV = 4; //4'b0100**

**OP\_LOAD\_FROM\_MEM = 5; //4'b0101**

**OP\_STORE\_TO\_MEM = 6; //4'b0110**

**OP\_SHORT\_TO\_REG = 7; //4'b0111**

**OP\_BNZ = 8; //4'b1000**

# Επέκταση του συνόλου των εντολών

Αρχικά πρέπει να προσθέσουμε το νέο opcode της νέας εντολής (**cpu.v**).

Πρόκειται για εντολή αριθμητικών ή λογικών πράξεων;  
Θα εκτελεστεί από την ALU.

Πρόκειται για εντολή μετακίνησης δεδομένων;  
Θα εκτελεστεί από το RegisterFile

Πρόκειται για εντολή διακλάδωση;  
Θα εκτελεστεί από την MCPU

# Επέκταση του συνόλου των εντολών (παράδειγμα sub)

Αρχικά πρέπει να προσθέσουμε το νέο opcode της νέας εντολής (**cpu.v**).

Πρόκειται για εντολή αριθμητικών ή λογικών πράξεων;  
Θα εκτελεστεί από την ALU.

Πρόκειται για εντολή μετακίνησης δεδομένων;  
Θα εκτελεστεί από το RegisterFile

Παράδειγμα:

Ας υποθέσουμε ότι θέλουμε να βάλουμε την εντολή που εκτελεί την αριθμητική πράξη της **αφαίρεση SUB**.

**Θα είναι της μορφής:** SUB Rd Ra Rb και θα εκτελεί:  $Rd = Ra - Rb$

Θα εκτελείται από την ALU. Άρα επεκτείνουμε τα opcodes που σχετίζονται με την ALU:

//Opcodes για ALU

```
parameter [OPCODE_SIZE-1:0] OP_AND = 0; //4'b0000
parameter [OPCODE_SIZE-1:0] OP_OR  = 1; //4'b0001
parameter [OPCODE_SIZE-1:0] OP_XOR = 2; //4'b0010
parameter [OPCODE_SIZE-1:0] OP_ADD  = 3; //4'b0011
parameter [OPCODE_SIZE-1:0] OP_SUB  = 4; //4'b0100
```

//Opcodes για registers και μνήμη

```
parameter [OPCODE_SIZE-1:0] OP_MOV  = 4; //4'b0100
parameter [OPCODE_SIZE-1:0] OP_LOAD_FROM_MEM = 5; //4'b0101
parameter [OPCODE_SIZE-1:0] OP_STORE_TO_MEM  = 6; //4'b0110
parameter [OPCODE_SIZE-1:0] OP_SHORT_TO_REG  = 7; //4'b0111
parameter [OPCODE_SIZE-1:0] OP_BNZ   = 8; //4'b1000
```



//Opcodes για ALU

```
parameter [OPCODE_SIZE-1:0] OP_AND = 0; //4'b0000
parameter [OPCODE_SIZE-1:0] OP_OR  = 1; //4'b0001
parameter [OPCODE_SIZE-1:0] OP_XOR = 2; //4'b0010
parameter [OPCODE_SIZE-1:0] OP_ADD  = 3; //4'b0011
parameter [OPCODE_SIZE-1:0] OP_SUB  = 4; //4'b0100
```

**//Opcodes για registers και μνήμη**

```
parameter [OPCODE_SIZE-1:0] OP_MOV  = 5; //4'b0101
parameter [OPCODE_SIZE-1:0] OP_LOAD_FROM_MEM = 6; //4'b0110
parameter [OPCODE_SIZE-1:0] OP_STORE_TO_MEM  = 7; //4'b0111
parameter [OPCODE_SIZE-1:0] OP_SHORT_TO_REG  = 8; //4'b1000
parameter [OPCODE_SIZE-1:0] OP_BNZ   = 9; //4'b1001
```



# Επέκταση του συνόλου των εντολών (παράδειγμα sub)

Πρέπει να γίνουν αλλαγές και στην ALU (**alu.v**).

```
module MCPU_Alu(cmd,in1,in2,out,CF);  
parameter CMD_SIZE=2;  
parameter WORD_SIZE=16;  
  
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  
parameter [CMD_SIZE-1:0] CMD_SUB  = 3; //2'b11
```



```
module MCPU_Alu(cmd,in1,in2,out,CF);  
parameter CMD_SIZE=3;  
parameter WORD_SIZE=16;  
  
parameter [CMD_SIZE-1:0] CMD_AND = 0; //3'b000  
parameter [CMD_SIZE-1:0] CMD_OR  = 1; //3'b001  
parameter [CMD_SIZE-1:0] CMD_XOR  = 2; //3'b010  
parameter [CMD_SIZE-1:0] CMD_ADD  = 3; //3'b011  
parameter [CMD_SIZE-1:0] CMD_SUB  = 4; //3'b100
```

Προστέθηκε το νέο cmd για να υποδεικνύετε στην ALU πότε γίνεται αφαίρεση και αναγκαστικά άλλαξε το πλήθος των bits του cmd από 2 σε 3 bits γιατί αλλιώς δεν έφταναν τα bits για την κωδικοποίηση της αφαίρεσης.

Επίσης, μην ξεχνάτε ότι οι παράμετροι γίνονται overload κατά την δημιουργία των instances από την MCPU, επομένως οι αλλαγές που σχετίζονται με πλήθος bits από παραμετροποίηση πρέπει να γίνουν και εκεί!!!

Στην προκειμένη περίπτωση είναι το ALU\_CMD\_SIZE στο cru.v το οποίο πρέπει να γίνει 3 bits

# Επέκταση του συνόλου των εντολών (παράδειγμα sub)

Πρέπει να γίνουν επιπλέον αλλαγές στην ALU (**alu.v**).

Ήρθε βασικά η ώρα της υλοποίησης της πράξης που εκτελεί η εντολή:

```
always @ (cmd, in1, in2)
```

```
.
```

```
.
```

```
.
```

```
CMD_SUB : begin
```

```
    {CF,out} = in1-in2;
```

Προσθέτουμε την πράξη στο case των πράξεων της ALU

```
end
```

```
.
```

```
.
```

```
.
```

# Επέκταση του συνόλου των εντολών (παράδειγμα sub)

Πρέπει να γίνουν επιπλέον αλλαγές στην Control Unit (**cpu.v**).

Επίσης, μην ξεχνάτε ότι οι παράμετροι γίνονται overload κατά την δημιουργία των instances από την MCPU, επομένως οι αλλαγές που σχετίζονται με πλήθος bits από παραμετροποίηση πρέπει να γίνουν και εκεί!!!

Στην προκειμένη περίπτωση είναι το ALU\_CMD\_SIZE στο cpu.v το οποίο πρέπει να γίνει 3 bits

Επίσης πρέπει να πείτε στην control unit ότι πρόκειται για εντολή εκτέλεσης πράξης:

```
case(opcode)
```

```
    OP_AND,OP_OR,OP_XOR,OP_ADD, OP_SUB:  
    begin  
        regset_cmd <= #2 regfileinst.NORMAL_EX;  
    end
```

```
.
```

```
.
```

```
.
```

```
parameter ALU_CMD_SIZE=3;
```

```
.
```

```
.
```

```
.
```

Γιατί παρακάτω γίνεται:

```
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));
```

Άρα πρέπει να προστεθεί η εντολή στο case των normally executed από την Control Unit

# Εκτέλεση αφαίρεσης με τους 16 νέους καταχωρητές

Γράφουμε ένα testbench για να τεστάρουμε τους 16 καταχωρητές και την αφαίρεση:

```
///SUB test sequence
```

```
i=0; cpuinst.raminst.mem[0]={cpuinst.OP_SHORT_TO_REG, R0, 8'b00001010}; //0: R0=10;
```

```
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_SHORT_TO_REG, R1, 8'b00000011}; //1: R1=3;
```

```
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_SUB, R12, R0, R1};          //2: R12=R0-R1
```

```
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_SUB, R12, R1, R0};          //3: R12=R1-R0
```

Ίσως πάρουμε αυτό το σφάλμα κατά την μεταγλώττιση:

```
** Error: C:/Users/fg/Documents/MicroCPU-LSL/microcpu-rtl/sub-test.v(57): (vlog-2730) Undefined variable: 'R12'.
```

Αυτό σημαίνει πως στον assembler μέσα στο testbench δεν έχουμε δηλώσει τους νέους καταχωρητές. Δείτε πως δηλώνονται οι R1,...,R4 μέσα στο testbench και δηλώτε και τους υπόλοιπους. Δηλαδή:

```
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
```

```
parameter [cpuinst.OPERAND_SIZE-1:0] R4 = 4;
```

```
parameter [cpuinst.OPERAND_SIZE-1:0] R5 = 5;
```

```
parameter [cpuinst.OPERAND_SIZE-1:0] R6 = 6;
```

```
parameter [cpuinst.OPERAND_SIZE-1:0] R7 = 7;
```

```
parameter [cpuinst.OPERAND_SIZE-1:0] R8 = 8;
```

# Εκτέλεση αφαίρεσης με τους 16 νέους καταχωρητές

Γράφουμε ένα testbench για να τεστάρουμε τους 16 καταχωρητές και την αφαίρεση:

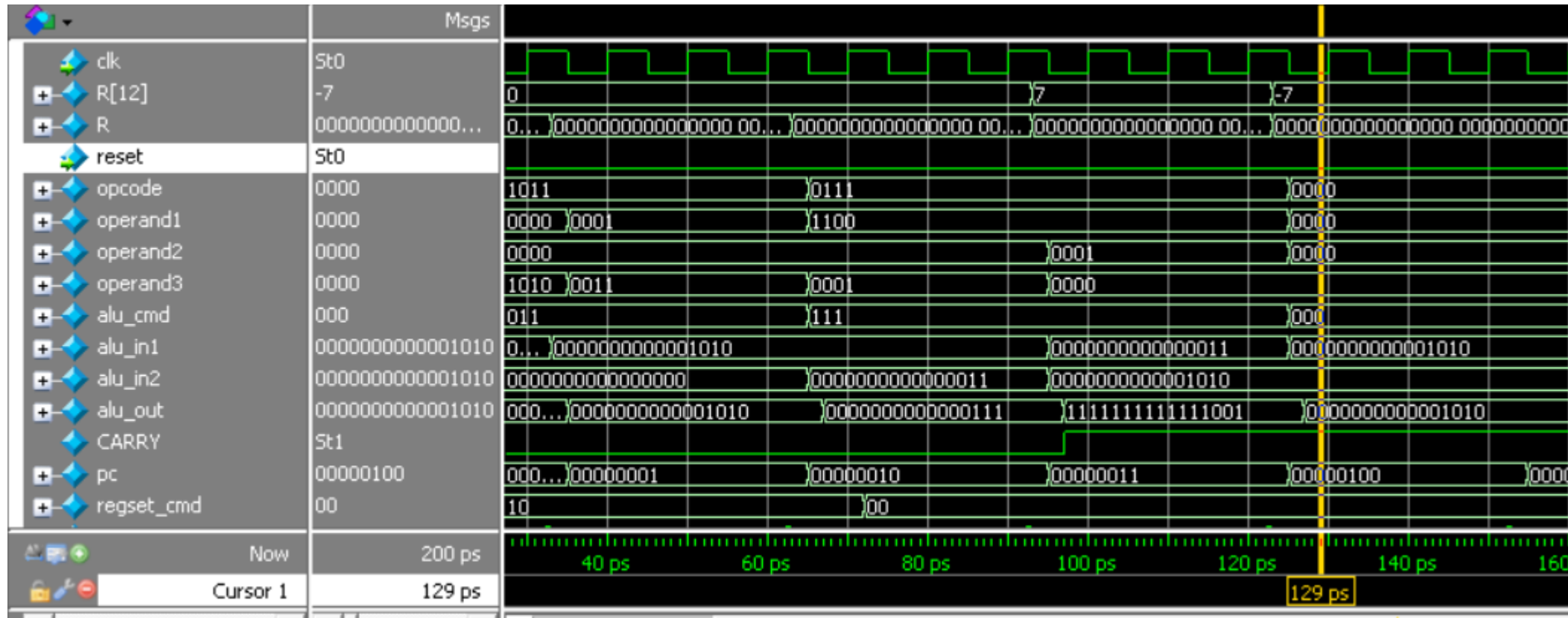
```
///SUB test sequence
```

```
i=0; cpuinst.raminst.mem[0]={cpuinst.OP_SHORT_TO_REG, R0, 8'b00001010}; //0: R0=10;
```

```
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_SHORT_TO_REG, R1, 8'b00000011}; //1: R1=3;
```

```
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_SUB, R12, R0, R1};          //2: R12=R0-R1
```

```
i=i+1;cpuinst.raminst.mem[i]={cpuinst.OP_SUB, R12, R1, R0};          //3: R12=R1-R0
```



# Βρόγχος if (R1!=α)

Αν θέλουμε να πούμε

If(R1!=0)

μπορούμε πολύ εύκολα να κάνουμε:

BNZ R1 Διεύθυνση

Αν θέλουμε να πούμε

If(R1!=1)

Τότε χρειαζόμαστε το εξής:

Rx=R1-1

BNZ Rx Διεύθυνση

Γενικά όταν έχουμε σχέση βρόγχου της μορφής

If(R!=a)

Τότε κάνουμε το εξής:

Rx=R-a;

BNZ Rx Διεύθυνση

Παράδειγμα στόχος:

If(R1!=4)

{

R5=R5+6;

}

else

{

R7=R7 xor R8;

}

R9=R2

*OP\_SHORT\_TO\_REG Rd value,  
για λόγους ευκολίας θα την λέω STR*

Παράδειγμα λύση:

//φτιάχνω πρώτα τα constants

1: STR R3 4 //R3=4

2: STR R4 6 //R4=4

3: SUB R2 R1 R3 //R2=R1-R3

4: BNZ R2 7//πήγαινε στην διεύθυνση που είναι μέσα στο if

5: XOR R7 R7 R8 //κώδικας μέσα στο else

6: BNZ R3 8 // ξέρουμε ότι το R3 δεν είναι 0

//οπότε είναι σα να γράφουμε πήγαινε στην γραμμή 8

7: ADD R5, R5, R4 //κώδικας μέσα στο if

8: MOV R9 R2

# Ειδική περίπτωση if (R1!=1)

Μπορεί να γίνει χωρίς αφαίρεση με χρήση της λογικής μάσκας XOR

Αν θέλουμε να ελέγξουμε ότι ο

R1=XXXX\_XXXX\_XXXX\_XXXX, είναι 1, τότε μπορούμε να φτιάξουμε την λογική μάσκα:

R2=0000\_0000\_0000\_0001 και να υπολογίσουμε το αποτέλεσμα της πράξης

R3= R1 xor R2

R3= XXXX\_XXXX\_XXXX\_XXXX xor 0000\_0000\_0000\_0001

XXXX\_XXXX\_XXXX\_XXX1  
xor 0000\_0000\_0000\_0001  
= XXXX\_XXXX\_XXXX\_XXX0

Το R3 θα είναι 0 μόνο και μόνο όταν το R1 είναι μονάδα.

Άρα:

MOV R2 0000\_0000\_0000\_0001

XOR R3 R1 R2

BNZ R3 διεύθυνση

# Έλεγχος μονά-ζυγά

Μας νοιάζει δηλαδή να ελέγξουμε μόνο το πρώτο bit (LSB) ενός καταχωρητή αν είναι 1 ή 0

R1=XXXX\_XXXX\_XXXX\_XXXX

Μπορεί να γίνει με χρήση της λογικής μάσκας AND

Αν θέλουμε να ελέγξουμε ότι ο

R1=XXXX\_XXXX\_XXXX\_XXXX, είναι μονός, τότε μπορούμε να φτιάξουμε την λογική μάσκα:

R2=0000\_0000\_0000\_0001 και να υπολογίσουμε το αποτέλεσμα της πράξης

R3= R1 AND R2

R3= XXXX\_XXXX\_XXXX\_XXXX AND 0000\_0000\_0000\_0001

    X1XX\_XX1X\_XX1X\_XXX?

and0000\_0000\_0000\_0001

===0000\_0000\_0000\_000?=R3

Το R3 θα είναι 0 μόνο και μόνο όταν το R1 είναι ζυγός.

Άρα:

MOV R2 0000\_0000\_0000\_0001

AND R3 R1 R2

BNZ R3 διεύθυνση

Άλλος τρόπος:

Μπορούμε να ολισθήσουμε τον καταχωρητή αριστερά κατά 15 θέσεις (BITS-1) ώστε να μείνει μόνο το LSB μέσα στον καταχωρητή. Μετά με έλεγχο BNZ μπορούμε να δούμε αν είναι μονός ή ζυγός



# Ολισθήσεις

Logical Shift Left (LSL) : θα χρησιμοποιήσετε τον operator  $\ll$ . Π.χ.:  $b=a\ll 3$ , το b είναι το a ολισθημένο κατά 3 θέσεις αριστερά. Καλύτερα να υλοποιηθεί για μεγαλύτερη ευκολία μέσα στην ALU. Μπορεί να υλοποιηθεί και στο registerfile.

Logical Shift Right (LSR): θα χρησιμοποιήσετε τον operator  $\gg$ . Π.χ.:  $b=a\gg 3$ , το b είναι το a ολισθημένο κατά 3 θέσεις δεξιά. Καλύτερα να υλοποιηθεί για μεγαλύτερη ευκολία μέσα στην ALU. Μπορεί να υλοποιηθεί και στο registerfile.

# Εύκολη και χρήσιμη εντολή η bitwise NOT

NOT Rd Ra, αντιστρέφει τα ψηφία του Ra και τα γράφει στον Rd (γίνεται με  $\text{out} = \sim \text{in1}$  στην ALU)