ΕΛΛΗΝΙΚΟ ΑΝΟΙΚΤΟ ΠΑΝΕΠΙΣΤΗΜΙΟ

Digital Systems Laboratory Media Computing

Σχολή Θετικών Επιστημών & Τεχνολογίας

Σύντομο Πρόγραμμα Σπουδών (ΣΠΣ)

Σχεδίαση ενσωματωμένων συστημάτων και εφαρμογές μικροελεγκτών στο Διαδίκτυο των Πραγμάτων (IoT)

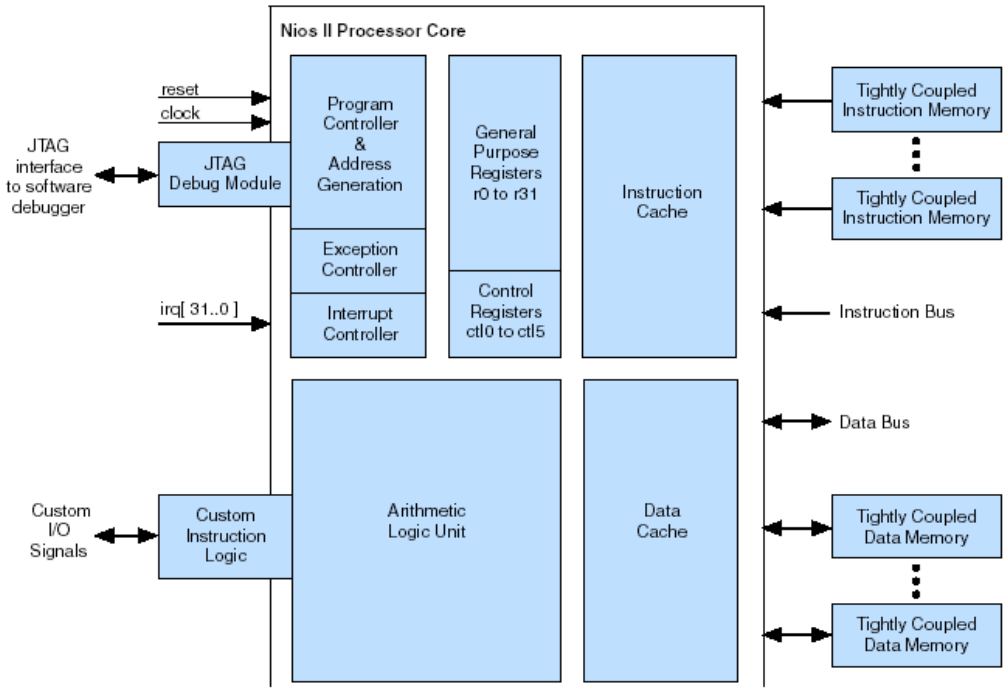
ΣΕΣ

(Embedded system design and microcontroller applications for the Internet of Things)

Παραδείγματα προγραμματισμού CPU NIOS-II

Εισαγωγή

Η γλώσσα assembly είναι μια γλώσσα προγραμματισμού χαμηλού επιπέδου και για αυτό το λόγο απαιτείται γνώση για την δομή του επεξεργαστή, ώστε να κατανοηθεί ο τρόπος λειτουργίας και τα αποτελέσματα από την εκτέλεσή της. Στην περίπτωση του επεξεργαστή Nios II η βασική αρχιτεκτονική είναι η ακόλουθη:



Βασικό ρόλο επίσης έχουν οι καταχωρητές γενικής χρήσης και ειδικού σκοπού που για τον επεξεργαστή Nios II αποτυπώνονται συνοπτικά στον παρακάτω πίνακα:

Register	Name	Function	Register	Name	Function
r0	zero	0x00000000	r16		
r1	at	Assembler Temporary	r17		
r2		Return Value	r18		
r3		Return Value	r19		
r4		Register Arguments	r20		
r5		Register Arguments	r21		
r6		Register Arguments	r22		
r7		Register Arguments	r23		
r8		Caller-Saved Register	r24	et	Exception Temporary
r9		Caller-Saved Register	r25	bt	Breakpoint Temporary *(1)*
r10		Caller-Saved Register	r26	gp	Global Pointer
r11		Caller-Saved Register	r27	sp	Stack Pointer
r12		Caller-Saved Register	r28	fp	Frame Pointer
r13		Caller-Saved Register	r29	ea	Exception Return Address
r14		Caller-Saved Register	r30	ba	Breakpoint Return Address *(1)*
r15		Caller-Saved Register	r31	ra	Return Address

Notes to Table 3–1:
(1) This register is used exclusively by the JTAG debug module.

Βασικές λειτουργίες του επεξεργαστή είναι:

- Η μετακίνηση δεδομένων - Data transfer
  - μεταξύ θέσεων μνήμης (τύπου ld/st)
  - μεταξύ καταχωρητών (τύπου mov)
- Η εκτέλεση αριθμητικών πράξεων:
  - Arithmetic
  - Logical
  - Comparator
  - Shift

Για την εκτέλεση των παραπάνω ορίζεται το σύνολο των εντολών γλώσσας μηχανής του επεξεργαστή τις οποίες σε άμεσα υψηλότερο επίπεδο προγραμματισμού περιγράφουμε στην σύνταξη των εντολών assembly, οι οποίες συνοπτικά αποτυπώνονται στον παρακάτω πίνακα:

Category	Instruction				Meaning
Arithmetic	addi	rB,	rA,	imm	rB +- rA + imm
	add	re,	rA,	rB	re +- rA + rB
	sub	re,	rA,	rB	re +- rA − rB
	and	re,	rA,	rB	re <- rA *and* rB
	andi	rB,	rA,	imm	rB <- rA *and* irrun.,
	or	re,	rA,	rB	re <- rA m·rB

Logical	ori	rB, rA, imm	rB <- rA *m· immu*
	xor	re, rA, rB	re <- rA *xor* rB
	xori	rB, rA, imm	rB <- rA *xor* imm,,
	nor	re, rA, rB	re <- rA *nm·* rB
Comparator	cmpgei	rB, rA, imm	rB <- (rA imm_.)? 1:0
	cmplti	rB, rA, imm	rB <- (rA < imm .)? 1:0
	cmpnei	rB, rA, imm	rB <- (rA /= imm_.)? 1:0
	cmpeqi	rB, rA, imm	rB <- (rA = imm_.)? 1:0
	cmpgeui	rB, rA, imm	rB +- (rA.,, imm,,)? 1:0
	cmpltui	rB, rA, imm	rB +- (rA.,, < imm,,)? 1:0
	cmpge	re, rA, rB	re +- (rA rB)? 1:0
	cmplt	re, rA, rB	re +- (rA < rB)? 1:0
	cmpne	re, rA, rB	re +- (rA /= rB)? 1:0
	cmpeq	re, rA, rB	re +- (rA = rB)? 1:0
	cmpgeu	re, rA, rB	re +- (rA.,, rB,,)? 1:0
	cmpltu	re, rA, rB	re +- (rA.,, < rB,,)? 1:0
Shift	sll	re, rA, rB	re +- rA «rB4..o
	slli	re, rA, imm	re +- rA «i..o
	srl	re, rA, rB	re +- rAu »rB4..o
	srli	re, rA, imm	re <- rAu » i ..o
	sra	re, rA, rB	re <- rA» »rB4..o
	srai	re, rA, imm	re <- rA, » i ..o
	rol	re, rA, rB	re <- rA *rol* rB4..o
	ror	re, rA, rB	re <- rA *ror* rB4..o
	roli	re, rA, imm	re <- rA *rol* i ..o
Memory[1]	ldw	rB, imm (rA)	rB <- MEM[imm„+rA]
	stw	rB, imm (rA)	MEM[imm,+rA] +- rB
Branch	br	imm	goto PC+4+imm_.
	bge	rA, rB, imm	if (rA rB) goto PC+4+imm»
	blt	rA, rB, imm	if (rA < rB) goto PC+4+imm»
	bne	rA, rB, imm	if (rA /= rB) goto PC+4+imm»
	beq	rA, rB, imm	if (rA = rB) goto PC+4+imm»
	bgeu	rA, rB, imm	if (rA,, rBu) goto PC+4+imm»
	bltu	rA, rB, imm	if (rA,, < rBu) goto PC+4+imm»
Jump	call	imm	goto imm «2; ra +- PC+4
	callr	rA	goto rA; ra +- PC+4
	ret		goto ra
	jmp	rA	goto rA
	jmpi	imm	goto imm «2
Misc	break		stops the processor[2]
	mov	rC, rA	rC <- rA ⇔ add rC, rA, r0
	movi	rB, IMMED16	sign extends the IMMED16 value to 32 bits and loads it into register B ⇔ addi rB, r0,

[1] **Παραλλαγές για άλλα μεγέθη:** ldb (Load Byte), ldbu (Load Byte Unsigned), ldh (Load Halfword), ldhu (Load Halfword Unsigned), stb (Store Byte), sth (Store Halfword)
Παραλλαγές για πρόσβαση σε περιφερειακά I/O (cache bypass): ldwio (LoadWord I/O), ldbio (Load Byte I/O), ldbuio (Load Byte Unsigned I/O), ldhio (Load Halfword I/O), ldhuio (Load Halfword Unsigned I/O), stwio (StoreWord I/O), stbio (Store Byte I/O), sthio (Store Halfword I/O)

Data transfer	movui	rB, IMMED16	IMMED16 zero extends the IMMED16 value to 32 bits and loads it into register B ⇔ ori rB, r0, IMMED16
	movia	rB, LABEL	loads a 32-bit value that corresponds to the address LABEL into register B. ⇔ orhi rB, r0, %hi(LABEL) ori rB, rB, %lo(LABEL)

Για την εξοικείωση και την εκσφαλμάτωση προγραμμάτων γραμμένων σε γλώσσα assembly διατίθενται είτε εργαλεία προσομοίωσης της λειτουργίας του επεξεργαστή κατά την φάση επεξεργασίας προγραμμάτων γραμμένων σε γλώσσα assembly είτε παρακολούθησης σε πραγματικό χρόνο καταστάσεων και τιμών καταχωρητών με κατάλληλα εργαλεία. Η απλούστερη εκδοχή είναι η χρήση προγραμμάτων προσομοίωσης.

Ένα πρόγραμμα εξομοίωσης που μπορεί να εγκατασταθεί τοπικά γραμμένο σε γλώσσα java είναι το JNiosEmu. Πρόκειται για ένα περιβάλλον ανάπτυξης και εξομοιωτή που επιτρέπει προγραμματισμό στον assembler, συναρμολογεί εύκολα και εκτελεί τον κώδικα και ο χρήστης μπορεί να βλέπει τι συμβαίνει με τις διαδοχικές τιμές των καταχωρητών και της μνήμης κατά τη διαδικασία εκτέλεσης κάθε μίας εντολής, χωρίς την προηγούμενη γνώση σύνθετων αλυσίδων εργαλείων. Το πρόγραμμα καθώς και οδηγίες χρήσης και εγκατάστασης διατίθενται στην διεύθυνση: http://stpe.github.io/jniosemu/.

Ένα πρόγραμμα εξομοίωσης σε περιβάλλον web (μέσα από το πρόγραμμα περιήγησης) είναι το CPUlator, που είναι ένας προσομοιωτής επεξεργαστών Nios II, ARMv7 και MIPS (επεξεργαστής και συσκευές I / O υπό συγκεκριμένες διαμορφώσεις) και ένα πρόγραμμα εντοπισμού σφαλμάτων που εκτελείται μέσα από ένα πρόγραμμα περιήγησης ιστού. Είναι σχεδιασμένο ως εργαλείο για την εκμάθηση προγραμματισμού γλωσσών συναρμολόγησης και οργάνωσης ηλεκτρονικών υπολογιστών. Το πρόγραμμα καθώς και οδηγίες χρήσης και εγκατάστασης διατίθενται στην διεύθυνση: https://cpulator.01xz.net.

Να σημειωθεί ότι ο επεξεργαστής Nios II είναι παραμετροποιήσιμος με αποτέλεσμα να μην υποστηρίζεται απαραίτητα όλη η λειτουργικότητα (ή όλες οι υποστηριζόμενες από τον gnu assembler directives) ή κάθε λογισμικό εξομοίωσης της λειτουργίας μπορεί να παρουσιάζει αποκλίσεις (bugs) τα οποία όμως στη γενική περίπτωση δεν αναμένεται να σας προβληματίσουν.

Μέρος 1

Προγραμματισμός με γλώσσα μηχανής (Assembly)

Παράδειγμα 1

Να υλοποιηθεί το αντίστοιχο του παρακάτω ψευδοκώδικα γλώσσας C θεωρώντας ότι οι μεταβλητές αρχικοποιούνται σε καταχωρητές γενικού σκοπού του Nios II σε γλώσσα assembly:

```
unsigned int a = 0x00000000;
unsigned int b = 0x00000001;
unsigned int c = 0x00000002;
```

```
a = b + c;
```

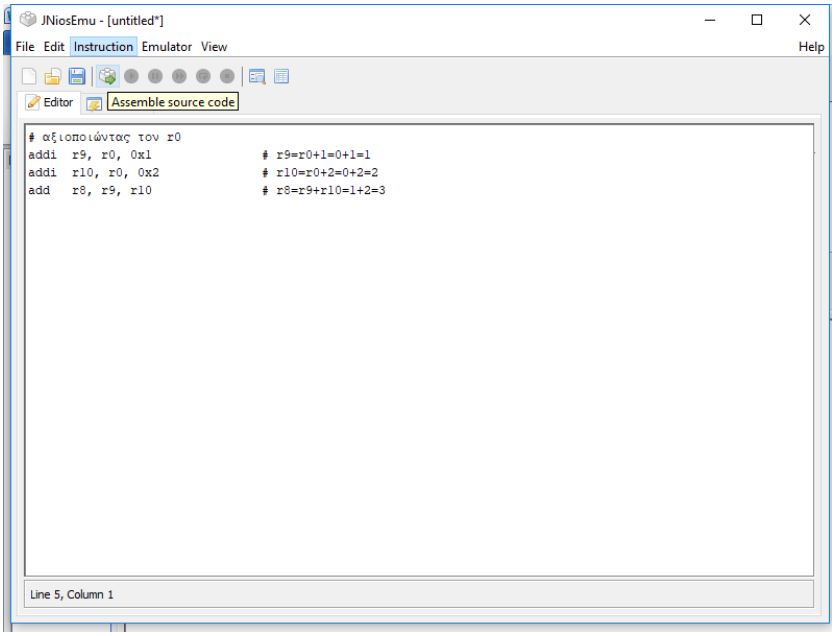
Ενδεικτικό αποτέλεσμα

```
# αξιοποιώντας τον r0
addi  r9, r0, 0x1          # r9=r0+1=0+1=1
addi  r10, r0, 0x2         # r10=r0+2=0+2=2
add   r8, r9, r10          # r8=r9+r10=1+2=3
```

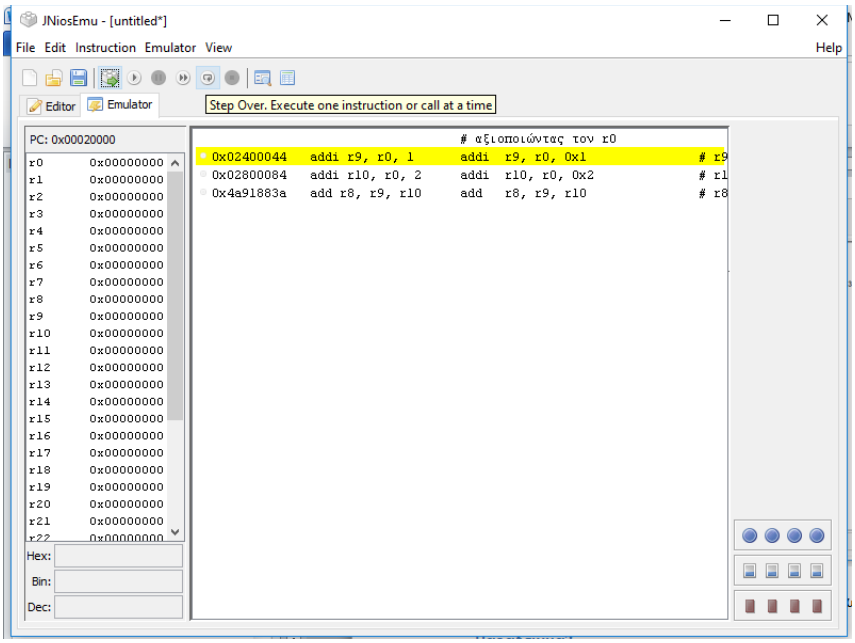
Η τεχνική αυτή αποδίδει ορθά αποτελέσματα όσο ικανοποιείται ο περιορισμός της **addi** :

- rB ← rA + σ (IMM16),
- Sign-extends the 16-bit immediate value and adds it to the value of rA. Stores the sum in rB.
- Άρα για IMM16≥0x8000 ο αριθμός λαμβάνεται ως αρνητικός
    - addi r8, r9, 0x8000 -> r8 = r9 + 0xFFFF8000

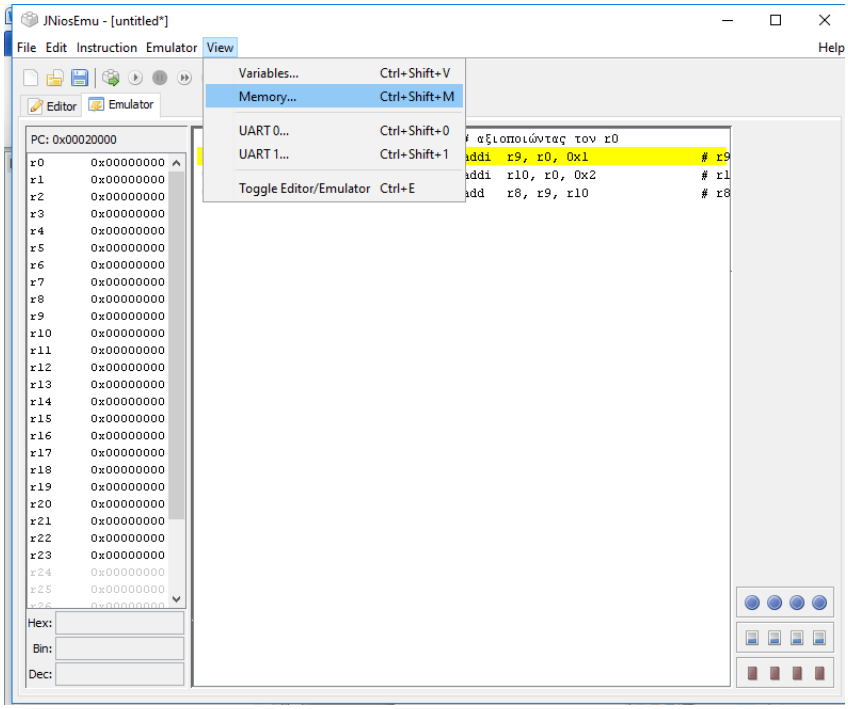
Τρέχοντας το αντίστοιχο παράδειγμα στον εξομοιωτή JNiosEmu καταρχήν το αντιγράφουμε στο παράθυρο συγγραφής του κώδικα (editor) και στην συνέχεια πατάμε το πλήκτρο συναρμολόγησης (assemble):

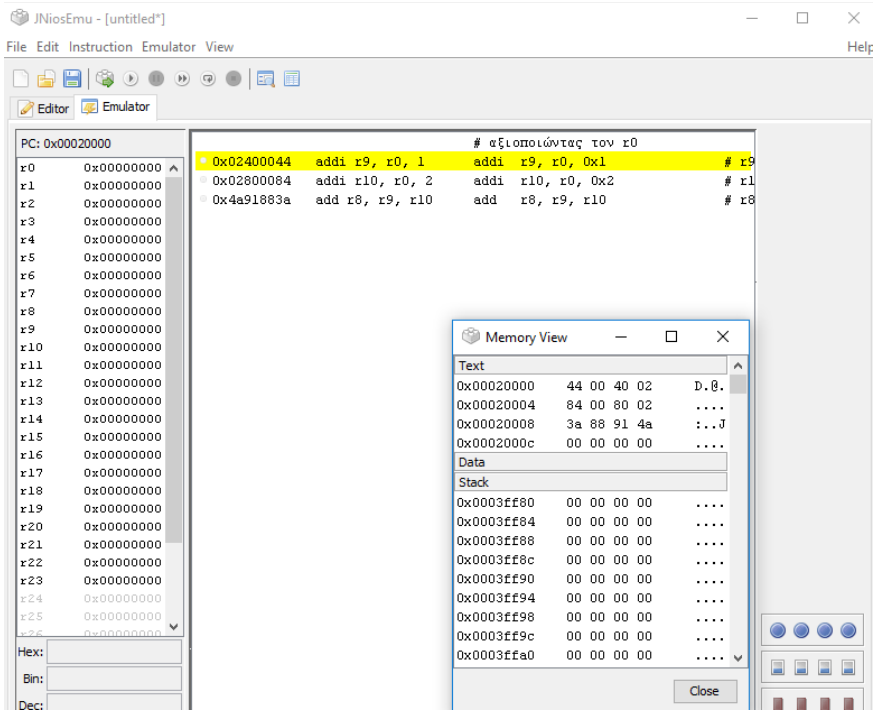


Μεταβαίνουμε αυτόματα στο παράθυρο εξομοίωσης του κώδικα (emulator), όπου εμφανίζεται ο κώδικας μηχανής που προέκυψε σε δεκαεξαδικές τιμές και οι τιμές καταχωρητών:



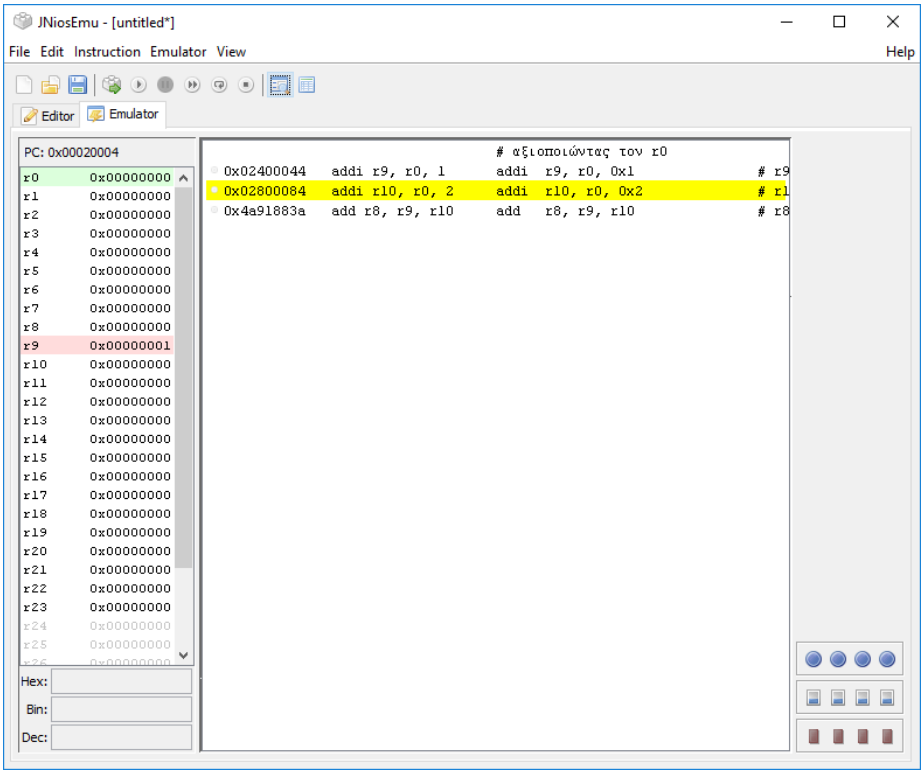
Επιλέγοντας από το μενού view->memory μπορούμε να δούμε ακόμα τις περιοχές μνήμης ενδιαφέροντος και τα περιεχόμενά τους:



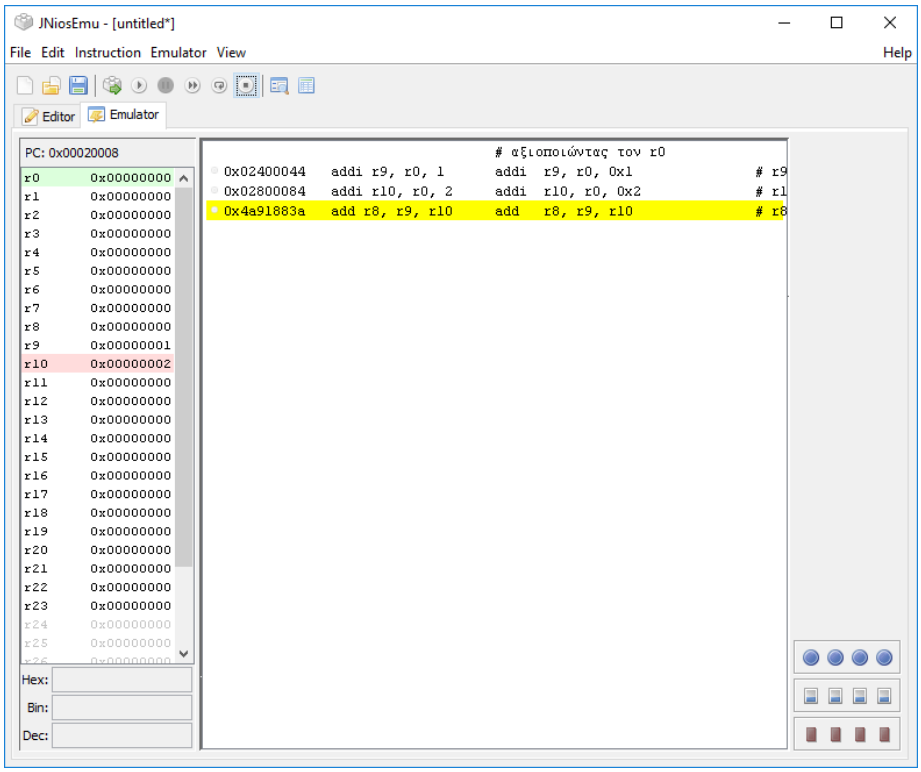


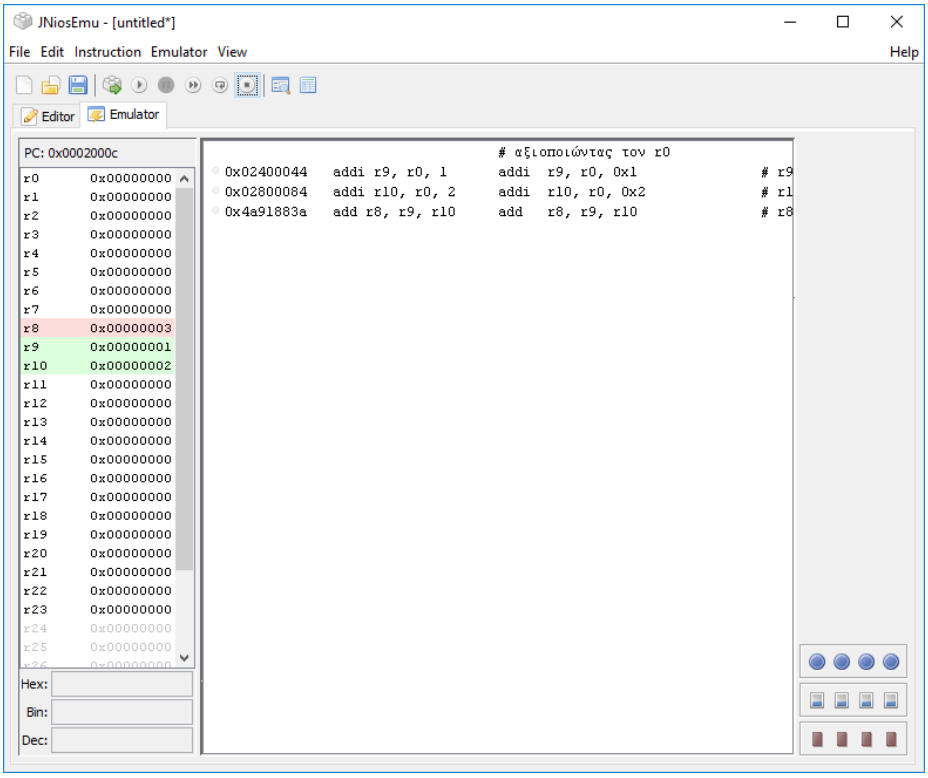
Μπορούμε να εκτελέσουμε τον κώδικα εντολή-εντολή επιλέγοντας από το μενού step-over. Σε κάθε βήμα επισημαίνεται η εντολή που εκτελέστηκε και οι τρέχουσες τιμές των καταχωρητών που επηρεάζονται και μνήμης. Στην συνέχεια βλέπουμε το αποτέλεσμα της εκτέλεσης της εντολής:

```
addi  r9, r0, 0x1          # r9=r0+1=0+1=1
```



Στην συνέχεια βλέπουμε τα διαδοχικά αποτελέσματα της εκτέλεσης των εντολών:
`addi r10,r0,0x2 # r10=r0+2=0+2=2` και `add r8, r9, r10# r8=r9+r10=1+2=3`





Παράδειγμα 2

Να υλοποιηθεί το αντίστοιχο του παρακάτω ψευδοκώδικα γλώσσας C θεωρώντας ότι οι μεταβλητές αρχικοποιούνται σε καταχωρητές γενικού σκοπού του Nios II σε γλώσσα assembly:

```
unsigned int a = 0x00000000;
unsigned int b = 0x11223344;
unsigned int c = 0x55667788;
a = b + c;
```

Ενδεικτικό αποτέλεσμα

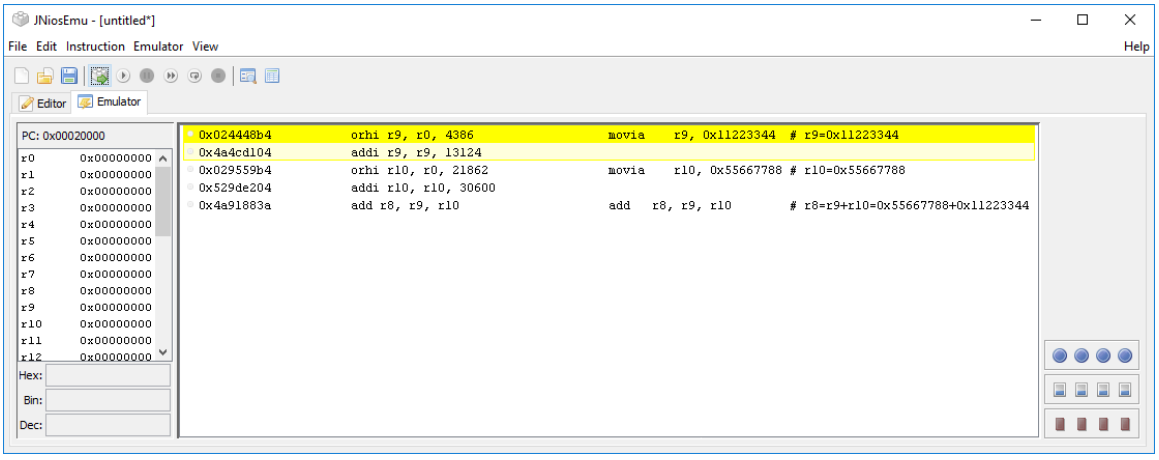
Σε αυτή την περίπτωση αντίθετα με το προηγούμενο παράδειγμα δεν ικανοποιείται ο περιορισμός της **addi** οπότε θα πρέπει να ανατρέξουμε σε άλλες εντολές μεταφοράς δεδομένων σε καταχωρητές και εκτέλεσης πράξεων με μεταβλητές 32-bit.

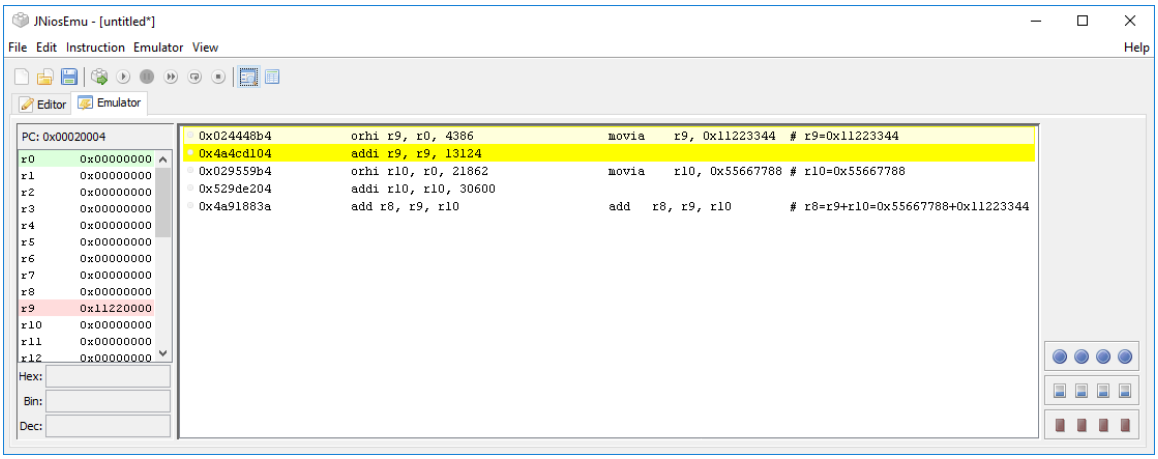
```
movhi r9, 0x1122      # r9=0x11220000
ori   r9, r9, 0x3344  # r9=0x11220000 || 0x00003344= 0x11223344
movhi r10, 0x5566     # r10=0x55660000
ori   r10, 0x7788     # r10=0x55667788
add   r8, r9, r10     # r8=r9+r10=0x55667788+0x11223344
```

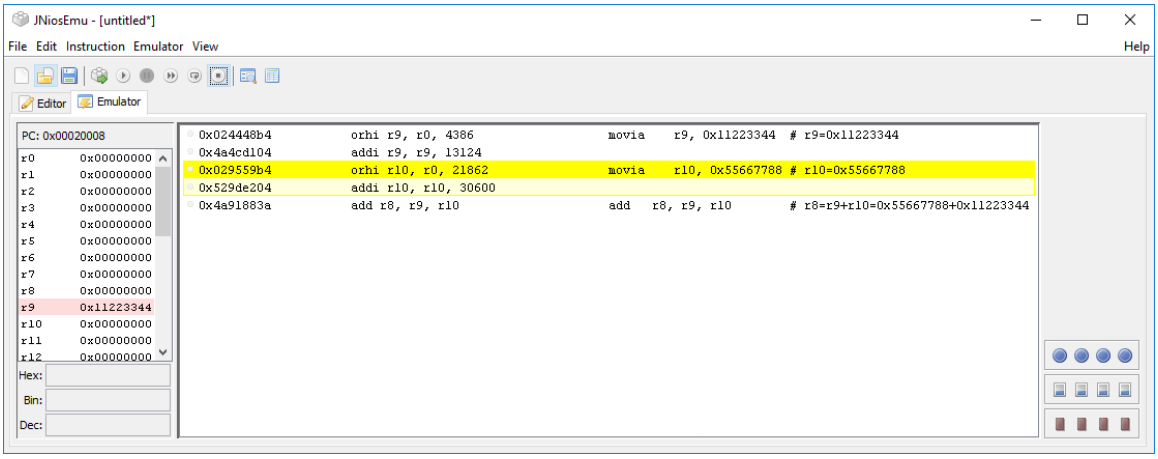
Εναλλακτικά η παραπάνω υλοποίηση θα μπορούσε να γίνει και με χρήση μακροεντολών ως ακολούθως:

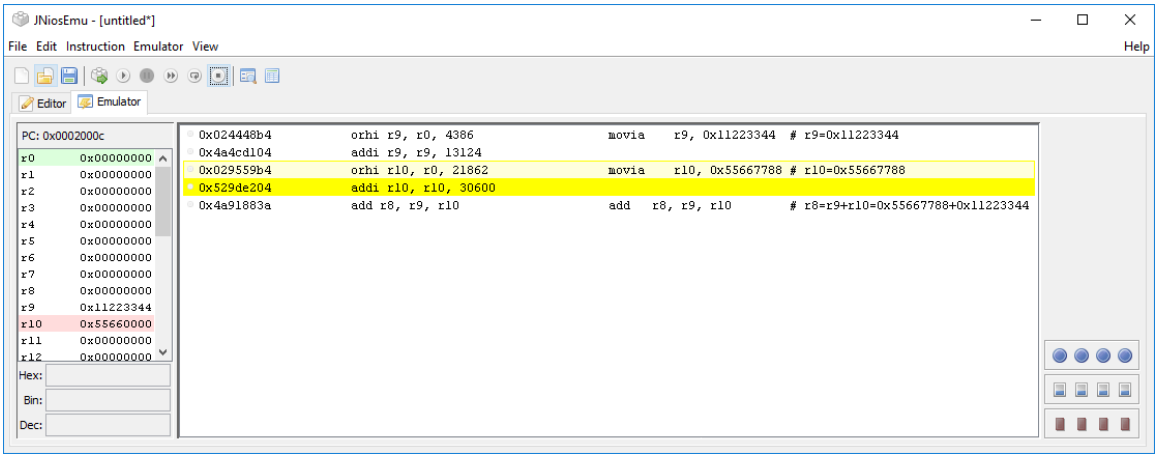
```
movia    r9, 0x11223344    # r9=0x11223344
movia    r10, 0x55667788   # r10=0x55667788
add   r8, r9, r10          # r8=r9+r10=0x55667788+0x11223344
```

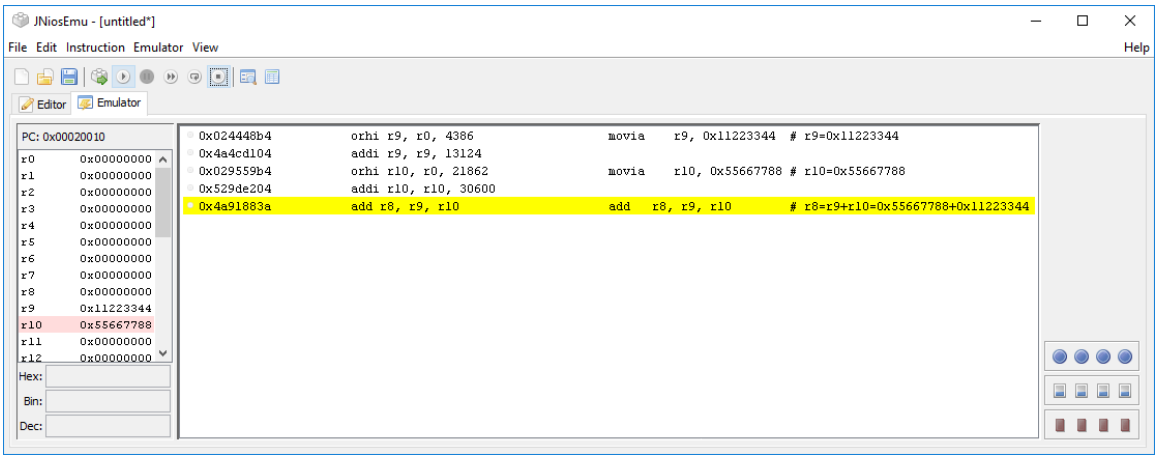
Δείχνουμε το αποτέλεσμα συναρμολόγησης της δεύτερης περίπτωσης στον JNiosEmu, όπου φαίνεται η ανάλυση κάθε μακροεντολής στις επιμέρους assembly εντολές μέσω των οποίων υλοποιείται.

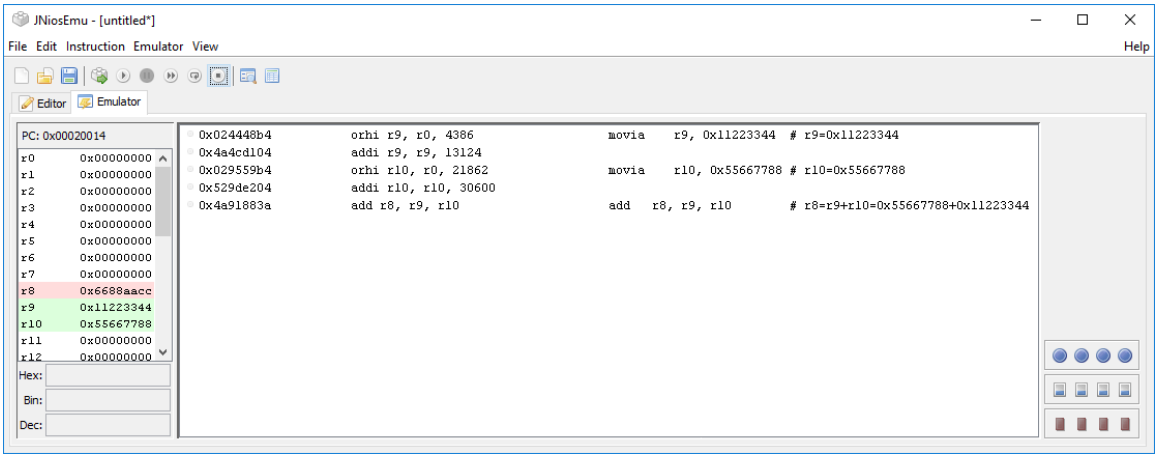



JNiosEmu - [untitled*]
File Edit Instruction Emulator View
Help
Editor
Emulator
PC: 0x00020004
r0 0x00000000
r1 0x00000000
r2 0x00000000
r3 0x00000000
r4 0x00000000
r5 0x00000000
r6 0x00000000
r7 0x00000000
r8 0x00000000
r9 0x11220000
r10 0x00000000
r11 0x00000000
r12 0x00000000
Hex:
Bin:
Dec:
0x024448b4 orhi r9, r0, 4386 movia r9, 0x11223344 # r9=0x11223344
0x4a4cd104 addi r9, r9, 13124
0x029559b4 orhi r10, r0, 21862 movia r10, 0x55667788 # r10=0x55667788
0x529de204 addi r10, r10, 30600
0x4a91883a add r8, r9, r10 add r8, r9, r10 # r8=r9+r10=0x55667788+0x11223344


JNiosEmu - [untitled*]
File Edit Instruction Emulator View
Help
Editor
Emulator
PC: 0x00020008
r0 0x00000000
r1 0x00000000
r2 0x00000000
r3 0x00000000
r4 0x00000000
r5 0x00000000
r6 0x00000000
r7 0x00000000
r8 0x00000000
r9 0x11223344
r10 0x00000000
r11 0x00000000
r12 0x00000000
Hex:
Bin:
Dec:
0x024448b4 orhi r9, r0, 4386 movia r9, 0x11223344 # r9=0x11223344
0x4a4cd104 addi r9, r9, 13124
0x029559b4 orhi r10, r0, 21862 movia r10, 0x55667788 # r10=0x55667788
0x529de204 addi r10, r10, 30600
0x4a91883a add r8, r9, r10 add r8, r9, r10 # r8=r9+r10=0x55667788+0x11223344


JNiosEmu - [untitled*]
File Edit Instruction Emulator View
Help
Editor
Emulator
PC: 0x0002000c
r0 0x00000000
r1 0x00000000
r2 0x00000000
r3 0x00000000
r4 0x00000000
r5 0x00000000
r6 0x00000000
r7 0x00000000
r8 0x00000000
r9 0x11223344
r10 0x55660000
r11 0x00000000
r12 0x00000000
Hex:
Bin:
Dec:
0x024448b4 orhi r9, r0, 4386 movia r9, 0x11223344 # r9=0x11223344
0x4a4cd104 addi r9, r9, 13124
0x029559b4 orhi r10, r0, 21862 movia r10, 0x55667788 # r10=0x55667788
0x529de204 addi r10, r10, 30600
0x4a91883a add r8, r9, r10 add r8, r9, r10 # r8=r9+r10=0x55667788+0x11223344





Παράδειγμα 3

Να υλοποιηθεί το αντίστοιχο του παρακάτω ψευδοκώδικα γλώσσας C θεωρώντας ότι οι μεταβλητές αρχικοποιούνται σε εξωτερική μνήμη του Nios II σε γλώσσα assembly:

```
unsigned int a = 0x00000000;
unsigned int b = 0x11223344;
unsigned int c = 0x55667788;
a = b + c;
```

Ενδεικτικό αποτέλεσμα

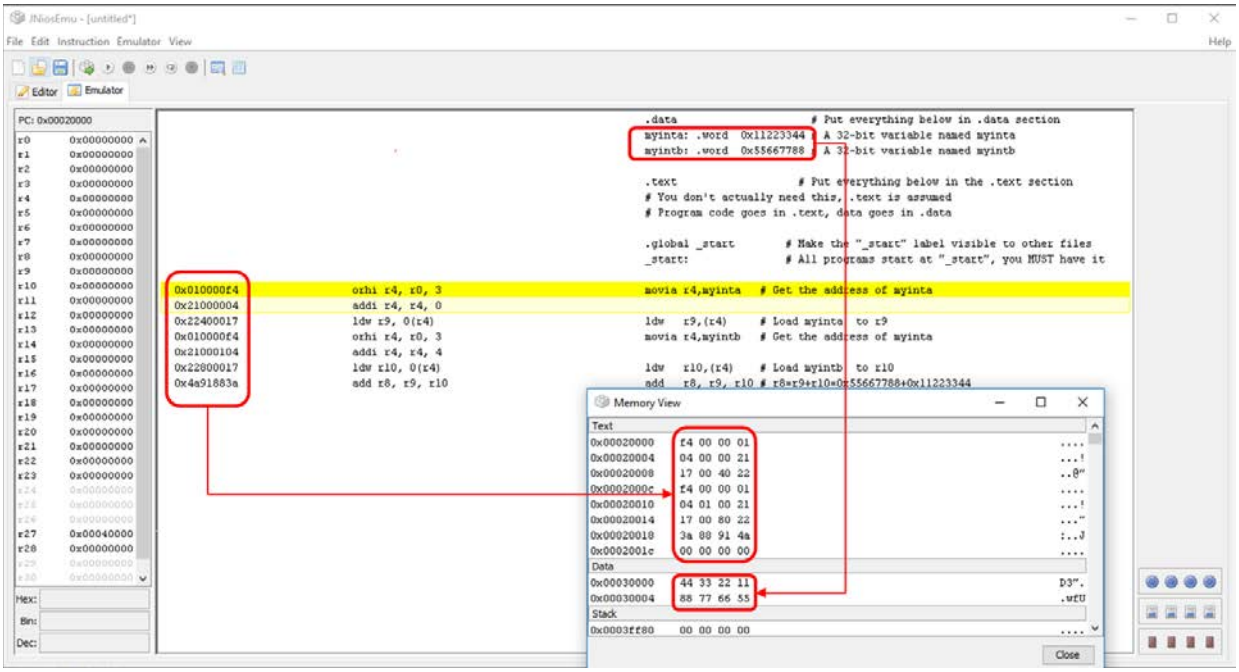
Σε αυτή την περίπτωση το προηγούμενο παράδειγμα με χρήση assembler directives και προσβάσεις μνήμης με μεταβλητές 32-bit μπορεί να διαμορφωθεί ως ακολούθως (για απλοποίηση αρχικά θεωρούμε ότι το αποτέλεσμα αποθηκεύεται σε καταχωρητή):

```
.data                       # Put everything below in .data section
myinta:    .word  0x11223344          # A 32-bit variable named myinta
myintb:    .word  0x55667788          # A 32-bit variable named myintb

.text                       # Put everything below in the .text section
```

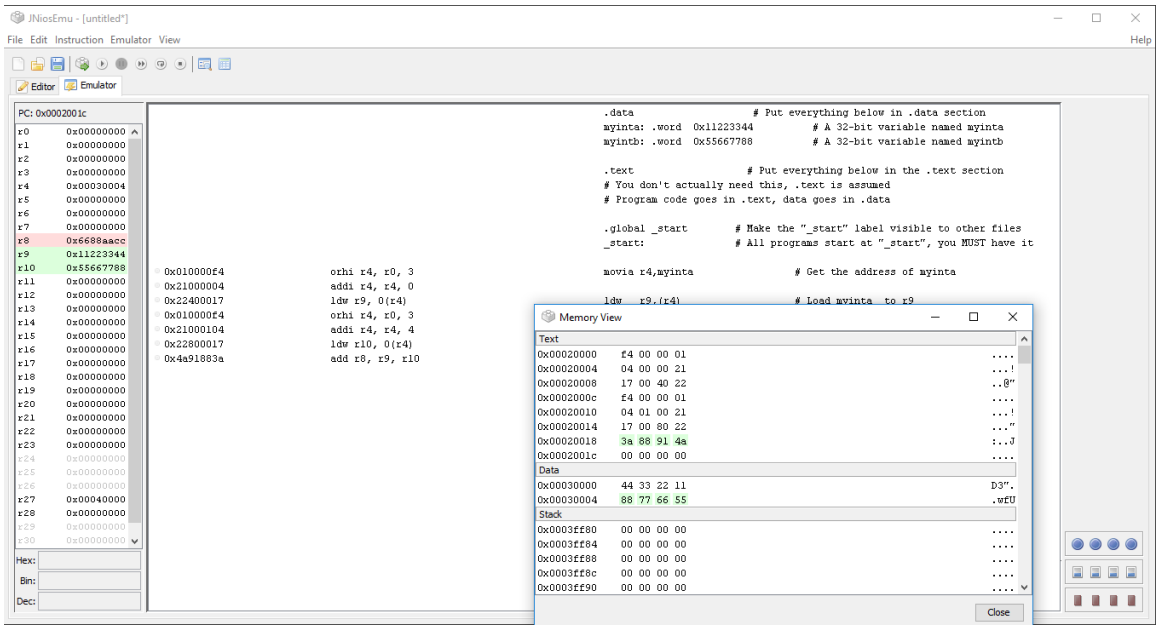
```
# You don't actually need this, .text is assumed
# Program code goes in .text, data goes in .data

.global _start        # Make the "_start" label visible to other files
_start:                # All programs start at "_start", you MUST have
it

movia r4,myinta            # Get the address of myinta
ldw   r9,(r4)              # Load myinta  to r9
movia r4,myintb            # Get the address of myinta
ldw   r10,(r4)             # Load myintb  to r10
add   r8, r9, r10          # r8=r9+r10=0x55667788+0x11223344
```

Δείχνουμε το αποτέλεσμα συναρμολόγησης του παραπάνω κώδικα στον JNiosEmu δείχνοντας και το παράθυρο ανάλυσης των δεδομένων στην μνήμη. Παρατηρούμε τα διαφορετικά τμήματα της μνήμης όπου καταχωρούνται:

- ο παραγόμενος κώδικας μηχανής-εντολές που θα εκτελέσει ο επεξεργαστής (έχει γίνει η υπόθεση ότι η εναρκτήρια διεύθυνση –αφορά το τμήμα .text- στη μνήμη είναι η 0x20000)
- η περιοχή μνήμης που αποθηκεύονται οι μεταβλητές (έχει γίνει η υπόθεση ότι η εναρκτήρια διεύθυνση –αφορά το τμήμα .data- στη μνήμη είναι η 0x30000)

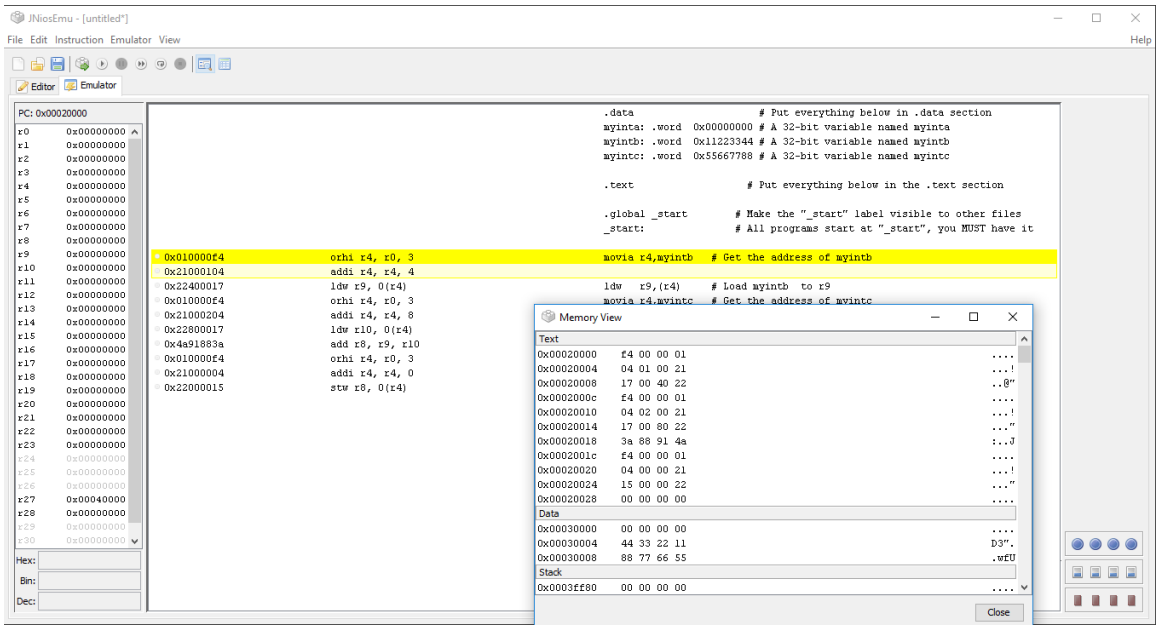


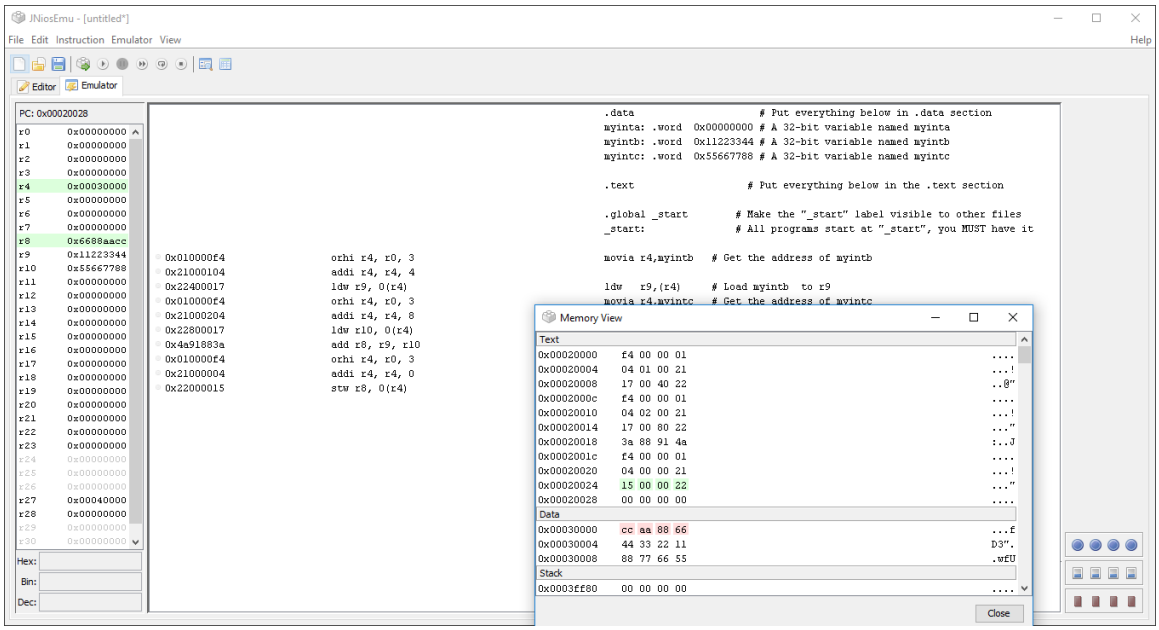
Στη συνέχεια δείχνουμε το τελικό αποτέλεσμα μετά από την εκτέλεση όλων των εντολών του παραπάνω κώδικα.



Επεκτείνουμε το προηγούμενο παράδειγμα, ώστε η υπολογιζόμενη τιμή να αποθηκεύεται στην μνήμη, οπότε και θα αποδίδεται πιο πιστά ο αρχικός κώδικας του ζητουμένου σε C και δείχνουμε τα αντίστοιχα περιεχόμενα της μνήμης με έμφαση στην περιοχή των δεδομένων, όπου καταχωρούνται οι 3 πλέον μεταβλητές του κώδικα πριν και μετά την εκτέλεση του κώδικα:

```
.data                          # Put everything below in .data section
myinta:    .word  0x00000000 # A 32-bit variable named myinta
myintb:    .word  0x11223344 # A 32-bit variable named myintb
myintc:    .word  0x55667788 # A 32-bit variable named myintc

.text                          # Put everything below in the .text section

.global _start         # Make the "_start" label visible to other files
_start:                  # All programs start at "_start", you MUST have
it

movia r4,myintb   # Get the address of myintb
ldw   r9,(r4)     # Load myintb  to r9
movia r4,myintc   # Get the address of myintc
ldw   r10,(r4)    # Load myintc  to r10
add   r8, r9, r10 # r8=r9+r10=0x55667788+0x11223344
movia r4,myinta   # Get the address of myinta
stw   r8, 0(r4)  # store r8 content in memory address myinta
```





Παράδειγμα 4

Να υλοποιηθεί κώδικας σε γλώσσα assembly, το οποίο να εκτελεί έναν κυκλικό βρόχο επανάληψης ξεκινώντας από την τιμή 0x10000 θεωρώντας ότι οι μεταβλητές αρχικοποιούνται σε καταχωρητές του Nios II

Ενδεικτικό αποτέλεσμα

Σε αυτή την περίπτωση θα πρέπει να ανατρέξουμε σε εντολές μεταφοράς δεδομένων σε καταχωρητές και εκτέλεσης πράξεων με μεταβλητές 32-bit,όπως παρακάτω:

```
.data
```

```
            .equ STARTVALUE, 10000      # the counter start value

counter:    .word STARTVALUE            # the counter variable
word:       .asciz "Test string"        # not used – for demo purposes

            .global main
            .text
main:       movia r8, counter           # Get the address of counter
            ldw r9, 0(r8)               # Load current value to r9
loop:       subi r9, r9, 1              # count=count-1
            bne r9, r0, loop            # loop until count=0 (r9=r0=0)
            stw r9, 0(r8)
```

Παράδειγμα 5

Να υλοποιηθεί κώδικας σε γλώσσα assembly, το οποίο να υλοποιεί το γινόμενο δύο διανυσμάτων Α, Β με Ν στοιχεία έκαστο ($P = \Sigma_{i=0}^{n-1} A(i) \times B(i)$) θεωρώντας ότι οι μεταβλητές αρχικοποιούνται αρχικοποιούνται σε εξωτερική μνήμη του Nios II και κάθε στοιχείο των διανυσμάτων Α, Β αποτελεί μεταβλητή 32-bit

Ενδεικτικό αποτέλεσμα

```
.include "nios_macros.s"
.global _start
_start:
movia r2, AVECTOR  # Register r2 is a pointer to vector A
movia r3, BVECTOR  # Register r3 is a pointer to vector B
movia r4, N
ldw r4, 0(r4)   # Register r4 is used as the counter for loop
iterations
add r5, r0, r0  # Register r5 is used to accumulate the product
LOOP: ldw r6, 0(r2)  # Load the next element of vector A
ldw r7, 0(r3)  # Load the next element of vector B
mul r8, r6, r7  # Compute the product of next pair of elements
add r5, r5, r8  # Add to the sum
addi r2, r2, 4  # Increment the pointer to vector A
addi r3, r3, 4  # Increment the pointer to vector B
subi r4, r4, 1  # Decrement the counter
bgt r4, r0, LOOP  # Loop again if not finished
stw r5, DOT_PRODUCT(r0)  # Store the result in memory
STOP: br STOP
N:
.word 6  # Specify the number of elements
AVECTOR:
.word 5, 3, -6, 19, 8, 12  # Specify the elements of vector A
BVECTOR:
.word 2, 14, -3, 2, -5, 36  # Specify the elements of vector B
DOT_PRODUCT:
.skip 4
```