Εντολές: Η γλώσσα μηχανής

- **Αρχιτεκτονική Σύνολο Εντολών MIPS (assembly)**
- **αναπαράσταση στο υλικό (MIPS)**
- **μετάφραση από HLL σε LLL**
  - **Δομές ελέγχου και βασικοί τύποι δεδομένων**

Κεφ. 2 + APPENDIX B

2.1-2.10,2.13, Παράρτημα Β

Διασύνδεση Υλικού/Λογισμικού

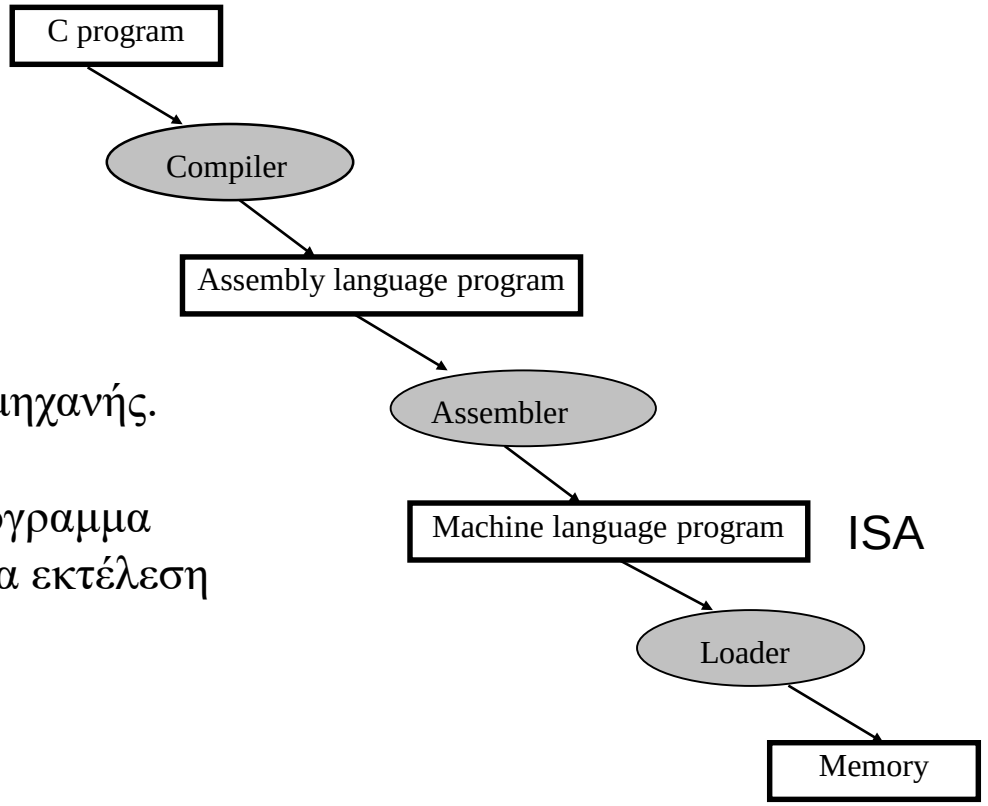
Η συμβολική αναπαράσταση των εντολών ονομάζεται συμβολική γλώσσα, και η αντίστοιχη αναπαράσταση με αριθμούς ονομάζεται γλώσσα μηχανής (ISA).

- Η ιεραρχία μετάφρασης για ένα πρόγραμμα από υψηλού επιπέδου γλώσσα μέχρι τη γλώσσα μηχανής.

- ένα πρόγραμμα σε C (υψηλού επιπέδου γλώσσα) μεταφράζεται σε συμβολική γλώσσα από τον μεταγλωττιστή

- ο συμβολομεταφραστής μεταφράζει την συμβολική γλώσσα σε γλώσσα μηχανής.

- Το πρόγραμμα που τοποθετεί το πρόγραμμα της γλώσσας μηχανής στη μνήμη για εκτέλεση ονομάζεται **φορτωτής (loader)**.



Αρχιτεκτονική Συνόλου Εντολών
Instruction Set Architecture (ISA)

- σύνολο εντολών
- τελεστές (add,sub,shift, load, store, branch,..)
- τελεσταίοι (καταχωρητές, μνήμη και άμεσες τιμές)
- σύνταξη εντολών και κωδικοποίηση στο δυαδικό
- σημασιολογία εντολών
- Αριθμός καταχωρητών, μέγεθος καταχωρητών, ονόματα καταχωρητών
- Μέγεθος μνήμης, Endianess
- Συγχρονισμός
- Διαχείριση διακοπών και εξαιρέσεων
- Από ποια διεύθυνση ξεκινά η εκτέλεση η πρώτη εντολή (PC)

…

MIPS Instruction Set Architecture (MIPS ISA)
Αρχιτεκτονική Συνόλου Εντόλων MIPS

Πολλά χαρακτηριστικά των εντολών της μηχανής MIPS οφείλονται στη κανονικοποίηση, όπως:

- όλες οι εντολές έχουν το ίδιο μέγεθος (4-bytes 32 bits),
- απαιτούνται πάντα τρείς καταχωρητές για τις αριθμητικές εντολές και τα πεδία των καταχωρητών κρατούνται στην ίδια θέση για όλα τα είδη εντολών.

Τελεσταίοι (MIPS operands)

Registers and Memory

32 integer register $0-$31 (sp:29, gp:28) + Hi and Lo Registers (multiply and divide)

32 floating point register $f1,$f31
16 double precision $f0 (0-1), $f2 (2-3), ...$f30 (30-31)
cond (1 bit register for fp compares)
Called coprocessor1 (c1) registers

2^{32} bytes Memory M[0]
M[1] ….
M[4294967295]

PC (program counter) – address of next instruction to be executed

Coprocessor0 (c0) Registers για (exceptions and interrupts)
EPC, Count, Status, Cause,…

Category	Instructions	Example	Meaning	Comments
Arithmetic	Add	add $1, $2, $3	$1 = $2 + $3	3 operands; data in register
	Subtract	sub $1, $2, $3	$1 = $2 - $3	3 operands; data in register
	Add Immediate	addi $1, $2, 100	$1 = $2 + 100	Used to add constants
Data Transfer	Load word	lw $1, 100($2)	$1 = Memory[$2+100]	Data from memory to register
	Store word	sw $1, 100($2)	Memory[$2+100] = $1	Data from register to memory
	Load upper immediate	lui $1, 100	$1 = 100 * 2^{16}	Loads constants in upper 16 bits
Conditional Branch	Branch on equal	beq $1, $2, 25	if ($1 == $2) go to PC + 4 + 100	Equal test; PC relative branch
	Branch on not equal	bne $1, $2, 25	if ($1 != $2) go to PC + 4 + 100	Not equal test; PC relative
	Set on less than	slt $1, $2, $3	if ($2<$3) $1 = 1; else $1 = 0	Compare less than; for beq, bne
	Set less than immediate	slti $1, $2, 100	if ($2<100) $1 = 1; else $1 = 0	Compare less than constant
Unconditional Jump	Jump	j 2500	go to 10000	Jump to target address
	Jump register	jr $31	go to $31	For switch, procedure return
	Jump and link	jal 2500	$31 = PC + 4; go to 10000	For procedure call

multiply	mult	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
multiply unsigned	multu	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
divide	div	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
divide unsigned	divu	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Unsigned quotient and remainder
move from Hi	mfhi	$s1	$s1 = Hi	Used to get copy of Hi
move from Lo	mflo	$s1	$s1 = Lo	Used to get copy of Lo

MIPS floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	`add.s $f2,$f4,$f6`	$f2 = $f4 + $f6	FP add (single precision)
	FP subtract single	`sub.s $f2,$f4,$f6`	$f2 = $f4 – $f6	FP sub (single precision)
	FP multiply single	`mul.s $f2,$f4,$f6`	$f2 = $f4 × $f6	FP. multiply (single precision)
	FP divide single	`div.s $f2,$f4,$f6`	$f2 = $f4 / $f6	FP divide (single precision)
	FP add double	`add.d $f2,$f4,$f6`	$f2 = $f4 + $f6	FP add (double precision)
	FP subtract double	`sub.d $f2,$f4,$f6`	$f2 = $f4 – $f6	FP sub (double precision)
	FP multiply double	`mul.d $f2,$f4,$f6`	$f2 = $f4 × $f6	FP multiply (double precision)
	FP divide double	`div.d $f2,$f4,$f6`	$f2 = $f4 / $f6	FP divide (double precision)
Data transfer	load word copr. 1	`lwc1 $f1,100($s2)`	$f1 = Memory[$s2 + 100]	32-bit data to FP register
	store word copr. 1	`swc1 $f1,100($s2)`	Memory[$s2 + 100] = $f1	32-bit data to memory
Condi-tional branch	*branch on FP true*	`bc1t 25`	*if (cond == 1) go to PC + 4 + 100*	*PC-relative branch if FP cond.*
	branch on FP false	`bc1f 25`	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	`c.lt.s $f2,$f4`	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	`c.lt.d $f2,$f4`	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than double precision

R,I,J Instruction types

Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 b
Format R	op	rs	rt	rd	Shamt	funct	Arithmetic Instruction
Format I	op	rs	rt	Address / immediate 16 bits			Transfer, branch, Imm.
Format J	op	Target address 24					Jump instruction

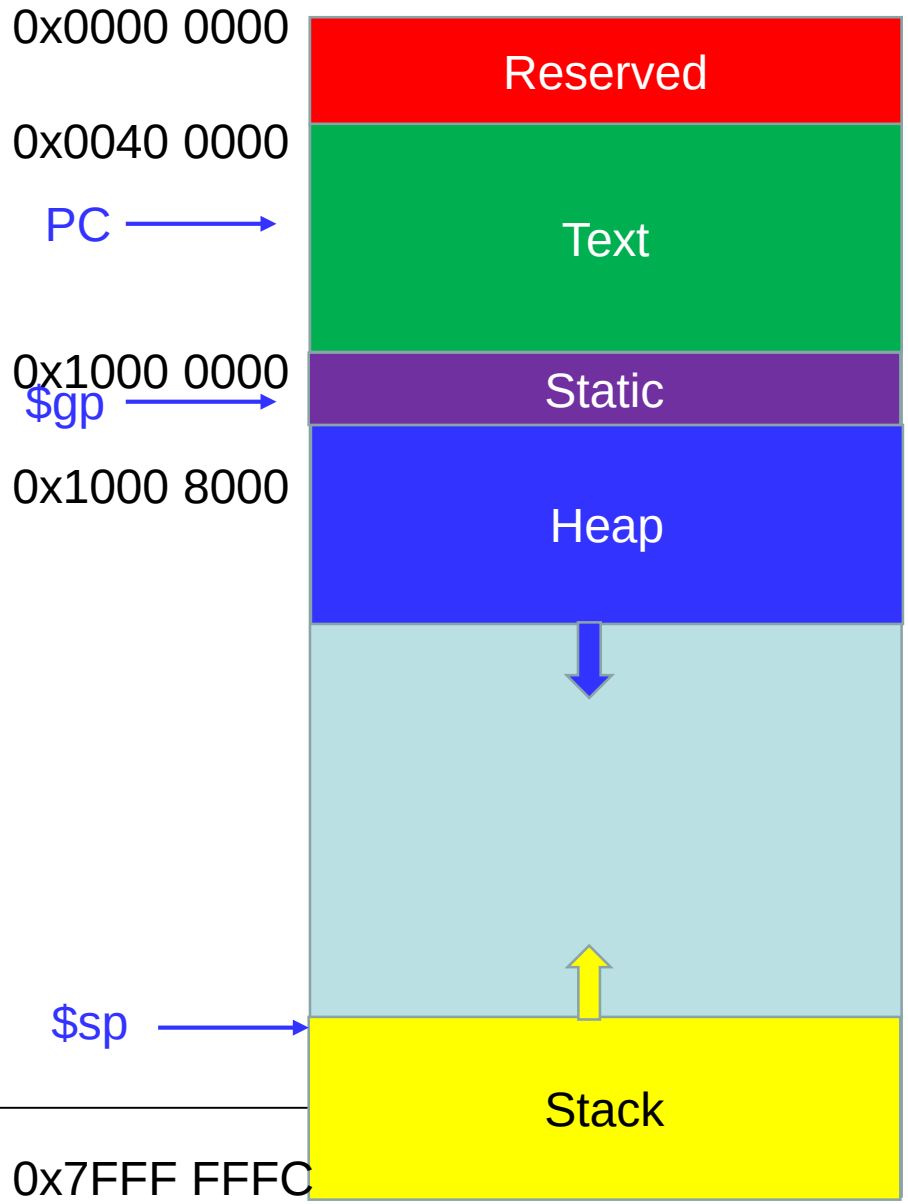
Name	Format	Example						Comments
Add	R	0	2	3	1	0	32	add $1, $2, $3
Sub	R	0	2	3	1	0	34	sub $1, $2, $3
Addo	I	8	2	1	100			addi $1, $2, 100
Lw	I	35	2	1	100			lw $1, 100($2)
Sw	I	43	2	1	100			sw $1, 100($2)
Lui	I	15	0	1	100			lui $1,100
Beq	I	4	1	2	25			beq $1, $2, 100
Bne	I	5	1	2	25			bne $1, $2, 100
Slt	R	0	2	3	1	0	42	slt $1, $2, $3
Slti	I	10	2	1	100			slti $1, $2, 100
J	J	2	2500					j 10000
Jr	R	0	31	0	0	0	8	jr $31
Jal	J	3	2500					jal 10000

Memory Model MIPS

Text: κώδικας
Static: global variables, constants
Heap: dynamic
Stack: local

User Address Space
0x0000 0000 – 0x7FFF FFFF

Kernel Address Space
0x8000 0000 – 0xFFFF FFFF



Αριθμητικές Εντολές

- Κάθε μηχανή πρέπει να είναι ικανή να κάνει αριθμητικές πράξεις.

- Κάθε αριθμητική εντολή στη γλώσσα MIPS έχει **τρεις μεταβλητές** (τελεσταίους).

```
add  a, b, c          # a= b+c
```

Παράδειγμα C : f=(g+h) - (i+j)

Τι κώδικα θα μπορούσε να παράξει ο μεταγλωττιστής της C;

```
add t0, g, h        #  προσωρινή μεταβλητή t0, περιέχει το g+h
add t1, i, j        #  προσωρινή μεταβλητή t1, περιέχει το i+j
sub f, t0 , t1      #  f = t0 - t1 = (g+h) - (i+j)
```

Τελεσταίοι του υλικού του υπολογιστή

- Στις συμβολικές γλώσσες οι τελεσταίοι (operators) των αριθμητικών εντολών είναι καταχωρητές *ή μνήμη* .
- Η αρχιτεκτονική MIPS έχει 32 ακέραιους καταχωρητές, οι οποίοι συμβολίζονται με $0, $1, ..., $31 (32 bits each)

- **Παράδειγμα** Έστω η πιο κάτω εντολή στη C : f = (g + h) - (i + j) ; μεταγλωττιστής της C συσχετίζει τις μεταβλητές f, g, h, i, και j με τους καταχωρητές $16, $17, $18, $19, και $20 αντιστοίχως.

```
add $8, $17, $18      # ο καταχωρητής $8 έχει το αποτέλεσμα g+h
add $9, $19, $20      # ο καταχωρητής $9 έχει το αποτέλεσμα i+j
sub  $16, $8, $9      #  ο καταχωρητής $16 έχει το αποτέλεσμα
                      # (g+h)-(i+j), την τιμή της μεταβλητής f
```

Οι καταχωρητές $8 και $9 αντιστοιχούν σε προσωρινές μεταβλητές

HLL εντολές εκφράζονται με πολλές LLL εντολές

Μνήμη αρχιτεκτονική MIPS

2^{32} x 8 bits

Byte Addressable

Little or Big Endian (can select at boot time)

Εντολές Μνήμης:
φόρτωσή (load) και αποθήκευση (store)

όχι πράξεις με τελεσταίους στην μνήμη

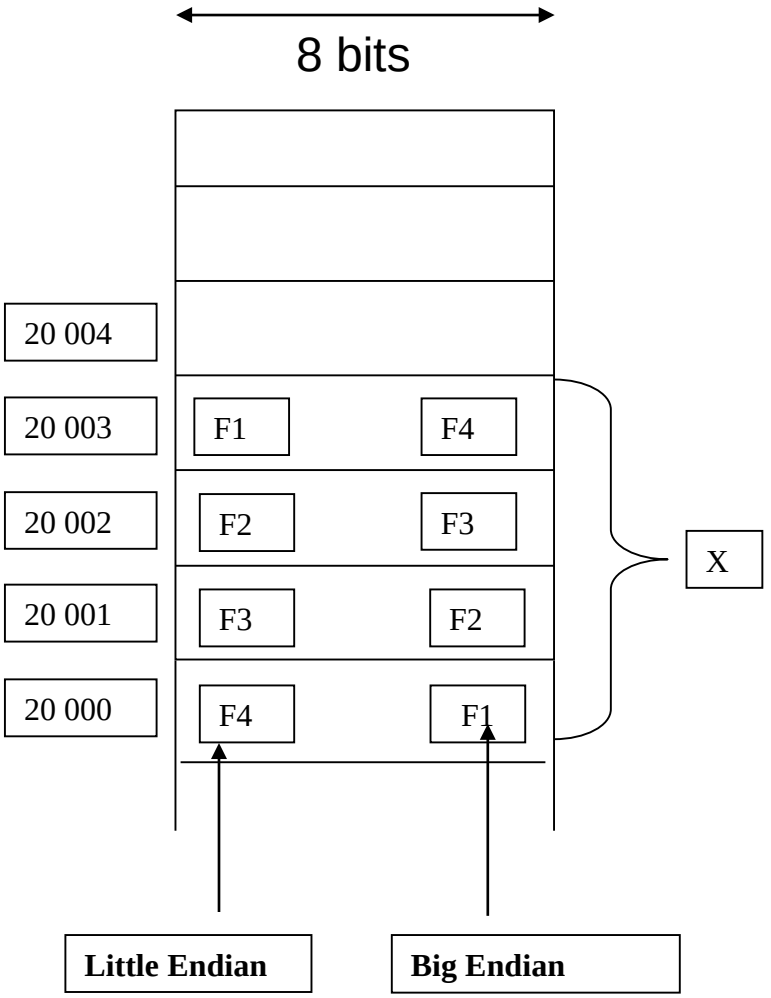
Στον MIPS υπάρχουν εντολές για load/store
lw/sw (4-bytes, word),
lh/sh (16 bit, half-word, 2-bytes)
lb/sb (8bit – 1 bytes)
ld/sd (8-bytes, double word)

-Υπάρχουν δύο τρόποι για αρίθμηση των Bytes σε μία λέξη: με τον πρώτο τρόπο ο μικρότερος αριθμός είναι ο πιο αριστερά και με τον δεύτερο τρόπο είναι ο πιο δεξιά:

- Big endian : Byte # 0 1 2 3
- Little endian : Byte # 3 2 1 0

- Ο επεξεργαστής MIPS εργάζεται και με τούς δύο τρόπους.
- Η οικογένεια επεξεργαστών της Intel ακολουθεί το Little Endian.

X=F1F2F3F4hex



Εντολή φόρτωσης - Η εντολή που μεταφέρει δεδομένα από την μνήμη σε ένα καταχωρητή, ονομάζεται **load -πχ lw (load word)** διαβάζει 4-byte ξεκινώντας από την διεύθυνση Ry+offset και τα αποθηκεύει στον Rx

```
lw Rx,offset(Ry) # Rx = [address = Ry+offset]
```

Παράδειγμα 4: Υποθέστε ότι ο πίνακας A έχει 100 integers στοιχεία και ο μεταγλωττιστής έχει συσχετίσει τις μεταβλητές g, h, και i με τους καταχωρητές $17, $18, και $19. Η βάση του πίνακα (διεύθυνση του πρώτου στοιχείου στον πίνακα) δίνεται είναι η διεύθυνση βρίσκεται στον $20
Μεταφράστε:

```
g = h + A[i];
```

```
add $21,$20,$19  // $21 προσωρινή
lw $8,0($21)
add $17,$18,$8
```

$20 base register A
$19 index register i

Διασύνδεση υλικού/λογισμικού

- Το μέγεθος του τύπου των μεταβλητών επηρεάζει την διευθυνσιοδότηση στην μνήμη
- g = h + A[i]; // A πίνακας ακεραίων κάθε ακέραιος 4-bytes
- A $20, i $19, g $17, $h 18

```
sll  $21,$19,2      // i*4
add $21,$20,$21     // base+i*4
lw $8,0($21)        // $8=[address]
add $17,$18,$8
```

sll shift left $19 by two bits

Εντολή Αποθήκευσης: Η αντίστροφη εντολή της load (φόρτωσε), είναι η **store -πχ sw** (φύλαξε μια λέξη). Αποθήκευση 4-bytes του Rx ξεκινώντας από την διεύθυνση Ry+offset

```
sw Rx,offset(Ry) # [Ry+offset] = Rx
```

- Παράδειγμα 5- Υποθέστε ότι η μεταβλητή h συσχετίζεται με τον καταχωρητή $18. Υποθέστε επίσης ότι ο καταχωρητής $19 έχει τιμή 4 x i. διεύθυνση του A βρίσκεται στο $20. Δίνεται ο πιο κάτω κώδικας σε C :

```
A[i] = h + A[i];
```

Ποιος είναι ο αντίστοιχος κώδικας σε γλώσσα MIPS;

```
add $21,$20,$19  // $21 προσωρινή
lw $8,0($21)
add $17,$18,$8
sw $17,0($21)
```

Διασύνδεση υλικού / λογισμικού

- Τα περισσότερα προγράμματα χρησιμοποιούν περισσότερες τοπικές μεταβλητές σε σχέση με τους καταχωρητές που διαθέτει η μηχανή.
  - Ο μεταγλωττιστής προσπαθεί να τοποθετήσει τις τοπικές μεταβλητές που χρησιμοποιούνται συχνότερα, στους καταχωρητές, και τις υπόλοιπες (μεταβλητές) στην μνήμη.

- Η διαδικασία της τοποθέτησης των λιγότερα χρησιμοποιημένων μεταβλητών στην μνήμη, ονομάζεται **register spilling.**
  - Αν, για παράδειγμα, έχουμε περισσότερες από 32 μεταβλητές σε ένα πρόγραμμα. τότε υποχρεωτικά κάποιες θα πρέπει να βρίσκονται στην μνήμη (για τον MIPS).

(Κωδικοποίηση) Αναπαράσταση εντολών στην μηχανή

- Όλες οι εντολές στο MIPS έχουν μέγεθος 32 bits
- Διαφορετικοί τύποι εντολών

- Η γενική μορφή εντολών **R-type** στη μηχανή MIPS με τα αντίστοιχα ονόματα των πεδίων.

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

op: η λειτουργία της εντολής (operation of the instruction)

rs: ο πρώτος καταχωρητής εισόδου (the first register, source operand)

rt: ο δεύτερος καταχωρητής εισόδου (the second register, source operand)

rd: ο καταχωρητής που θα πάρει το αποτέλεσμα (the register destination operand)

shamt: ποσότητα μετακίνησης

funct: συνάρτηση (function). Αυτό το πεδίο αποτελεί διαφοροποίηση της εντολής από το πεδίο op.

Αναπαράσταση εντολών R-type

add $8, $17, $18

op	rs	rt	rd	shamt	funct
0	17	18	8	0	32
000000	**10001**	**10010**	**01000**	**00000**	**100000**
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Το πρώτο και τελευταίο πεδίο (0 & 32) σε συνδυασμό δηλώνουν ότι η πράξη που θα εκτελεστεί είναι η πρόσθεση.
- Το δεύτερο πεδίο δίνει τον αριθμό του καταχωρητή ($17) που είναι η πηγή του πρώτου τελεσταίου της πρόσθεση.
- Το τρίτο πεδίο δίνει τον αριθμό του καταχωρητή ($18) που είναι η πηγή του δεύτερου τελεσταίου της πρόσθεσης.
- Το τέταρτο πεδίο δίνει τον αριθμό του καταχωρητή ($8) που θα δεχτεί το αποτέλεσμα της πρόσθεσης.
- Το πέμπτο πεδίο δεν χρησιμοποιείται σε αυτή την εντολή και για αυτό παίρνει την τιμή 0.
- $8 = $17+$18

Δεύτερο είδος εντολής ονομάζεται **I-type**

- χρησιμοποιείται από τις εντολές μετακίνησης δεδομένων (π.χ. **lw, sw**).
- Τα πεδία αυτής της μορφής είναι:

op	rs	rt	Imm
6 bits	5 bits	5 bits	16 bits

- Για παράδειγμα για την εντολή: **lw $8, 4($19)**
  - Η τιμή **19** θα τοποθετηθεί στο πεδίο **rs**
  - Η τιμή **8** θα τοποθετηθεί στο πεδίο **rt**
  - και offset στο **πεδίο Immediate**.

35	19	8	4

  - Σημειώστε ότι το πεδίο **rt** στην εντολή **lw** δίνει το πεδίο που θα πάρει το αποτέλεσμα.
  - Στο sw το πεδίο rt περιέχει τον καταχωρήτη με την τιμή που θα πάει στην μνήμη **sw $10, 16($29)**

43	29	10	16

Κωδικοποιήσεις των εντολών που μελετήσαμε μέχρι τώρα

Instruction	Format	op	rs	rt	rd	shamt	funct	Immediate
Add	R	0	Reg	Reg	Reg	0	32	n.a.
Sub	R	0	Reg	Reg	Reg	0	34	n.a.
Lw	I	35	Reg	Reg	n.a.	n.a.	n.a.	Imm15:0
Sw	I	43	Reg	Reg	n.a.	n.a.	n.a.	Imm15:0

- Μεγέθη των πεδίων για τις πιο πάνω εντολές:

Field	op	rs	rt	rd	shamt	funct	Imm
Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	16 bits

- οι εντολές add και sub έχουν το ίδιο op πεδίο.
- Το πεδίο funct προσδιορίζει ποια αριθμητική πράξη (πρόσθεση ή αφαίρεση) πρέπει να εκτελέσει.

Παράδειγμα 6- A[300] = h + A[300] ; //A πίνακας ακεραίων 4-bytes καθε int

Υποθέστε ότι η μεταβλητή h συσχετίζεται με τον καταχωρητή $18 και επίσης ότι ο καταχωρητής $19 έχει την τιμή A. Η εντολή μεταγλωττίζεται σε:

```
lw $8, 1200($19)      # ο καταχωρητής $8 παίρνει προσωρινά  το A[300]
add $8, $18 , $8      # ο καταχωρητής $8 παίρνει προσωρινά h+A[300]
sw $8, 1200($19)      # Το h+A[300] φυλάγεται πίσω στη θέση A[300]
```

Ποιός είναι ο κώδικας σε γλώσσα μηχανής (MIPS) για τις πιο πάνω εντολές;

op	rs	rt	(rd)	(shamt)	Imm /funct

Η μορφή των τριών εντολών στο δεκαδικό σύστημα είναι:

op	rs	rt	(rd)	(shamt)	Imm /funct
35	19	8	1200		
0	18	8	8	0	32
43	19	8	1200		

- Η εντολή lw προσδιορίζεται από τον αριθμό 35 (πεδίο op).
- Η εντολή add προσδιορίζεται από το πρώτο και τελευταίο πεδίο (0 και 32)
- Η εντολή sw προσδιορίζεται από τον αριθμό 43 στο πρώτο πεδίο
- Η αντίστοιχη δυαδική μορφή των εντολών (τρόπος αναπαράστασης τους σε γλώσσα μηχανής) είναι:

100011	10011	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	10011	01000	0000 0100 1011 0000		

- Προσέξτε την ομοιότητα της πρώτης και τρίτης εντολής στην δυαδική απεικόνιση.
  - Η μόνη διαφορά τους βρίσκεται στο πρώτο πεδίο, στο τρίτο bit από τα αριστερά.

Διασύνδεση Υλικού/Λογισμικού

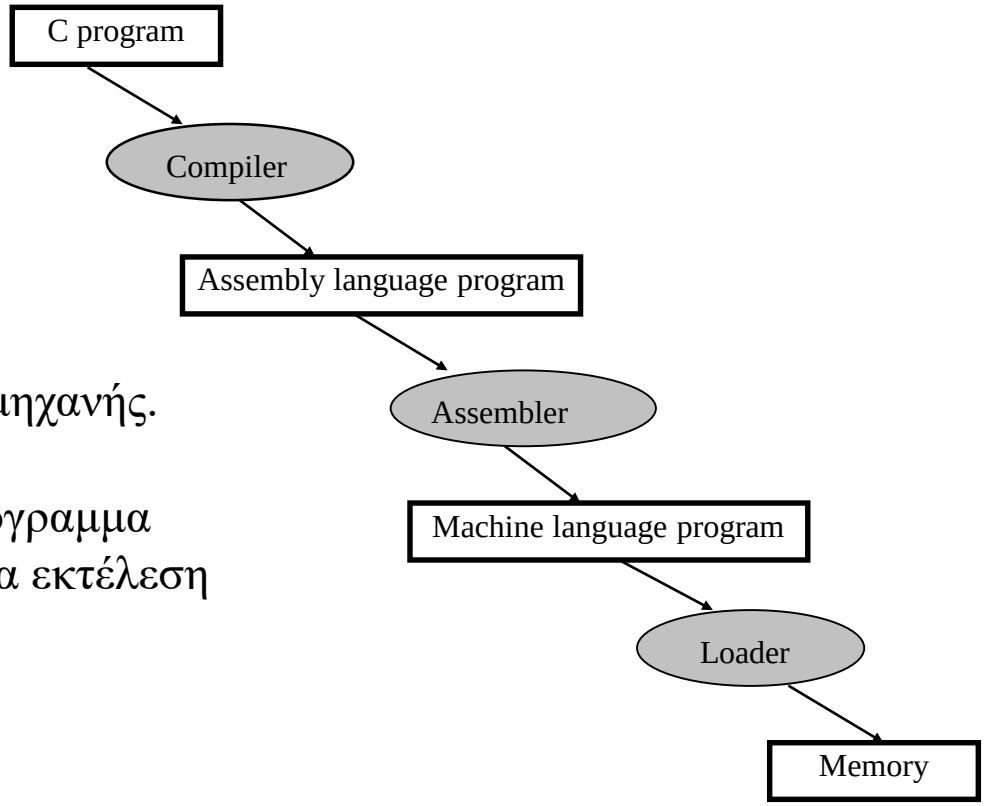
Η συμβολική αναπαράσταση των εντολών ονομάζεται συμβολική γλώσσα, και η αντίστοιχη αναπαράσταση με αριθμούς ονομάζεται γλώσσα μηχανής.

- Η ιεραρχία μετάφρασης για ένα πρόγραμμα από υψηλού επιπέδου γλώσσα μέχρι τη γλώσσα μηχανής.

- ένα πρόγραμμα σε C (υψηλού επιπέδου γλώσσα) μεταφράζεται σε συμβολική γλώσσα από τον μεταγλωττιστή

- ο συμβολομεταφραστής μεταφράζει την συμβολική γλώσσα σε γλώσσα μηχανής.

- Το πρόγραμμα που τοποθετεί το πρόγραμμα της γλώσσας μηχανής στη μνήμη για εκτέλεση ονομάζεται **φορτωτής (loader)**.



Η μεγάλη εικόνα:

- Σήμερα οι υπολογιστές είναι κτισμένοι πάνω σε δύο βασικές αρχές:
  1. Οι εντολές απεικονίζονται ως αριθμοί.
  2. Τα προγράμματα μπορούν να αποθηκευθούν στη μνήμη για να διαβαστούν, ή να γραφούν, όπως οι αριθμοί.
  3. Το ίδιο και τα δεδομένα: μεταβλητές, προσωρινά αποτελέσματα, δομές

- Αυτή η αρχή ονομάζεται **"η έννοια του αποθηκευμένου προγράμματος"** (stored program concept). John Von Neuman (1946) John Von Neuman, J. Presper Eckert and John William Mauchly

- **Αποθηκευμένα Προγράμματα --Τα προγράμματα φυλάγονται στη μνήμη σαν κώδικας σε γλώσσα μηχανής**. (ανεξάρτητα της υλοποίησης)

Καταχωρητής $0

- Ο καταχωρητής **$0** στη μηχανή MIPS έχει πάντα την **τιμή 0**.

```
add $8, $0, $18      # ο καταχωρητής 8 παίρνει το άθροισμα του
                     # καταχωρητή 0 και 18.
```

- ο συμβολομεταφραστής της γλώσσας MIPS έχει την πιο κάτω εντολή αν και δεν ανήκει στην αρχιτεκτονική της μηχανής MIPS:

```
move $8, $18         #ο καταχωρητής 8 θα πάρει το περιεχόμενο του
                     #καταχωρητή 18.
```

- Ο συμβολομεταφραστής μεταφράζει την πιο πάνω εντολή στην αντίστοιχη εντολή του MIPS που είναι: **add $8, $0, $18**

- **Υπάρχουν διάφορες ψευδο-εντολές στον συμβολομεταφραστή**

PC (program counter)

- PC 32-bit καταχώρητης για καθορισμο ροής ελέγχου
- Κάθε κύκλο ο καταχωρητής PC περιέχει την διεύθυνση της εντολής που εκτελείται
- Αυξάνεται κατά 4 για να προχώρησει στην επόμενη εντολή (4-bytes instructions)
  - PC= PC+4 για κάθε εντολή που δεν είναι διακλάδωση

Εντολές διακλαδώσεις/ ανάληψη αποφάσεων

- Η μηχανή MIPS υποστηρίζει δύο εντολές για λήψη αποφάσεων, παρόμοιες με το if και το goto,

  beq register1, register2, L1

  Label: SingExtention(Imm15:0): 32767..-32768

- Η πιο πάνω εντολή **(branch equal)** μεταφέρει την εκτέλεση του προγράμματος στην εντολή που έχει **ετικέτα (label)** L1 αν τα δεδομένα του register1 ισούνται με τα δεδομένα του register2.

If (Reg1==Reg2)

PC = PC + 4 + L1 x 4 (relative addressing)

Else

PC =PC + 4

bne register1, register2, L1

- **branch not equal**
- Αυτές οι δύο εντολές ονομάζονται **διακλαδώσεις υπό συνθήκη (conditional branches)**.

- Παράδειγμα if: Δίνεται ο ακόλουθος κώδικας σε γλώσσα C :

```
if (i==j)
        f = g + h;
f = f – i;
```

Οι μεταβλητές f, g, h, i και j είναι τοποθετημένες στους καταχωρητές $16 έως $20.

```
        bne   $19, $20, L1          # go to L1  αν  i != j
        add  $16, $17, $18          # f = g + h (αγνοείται αν i = j)
L1:    sub $16, $16, $19            # f = f - i  εκτελείται πάντοτε
```

L1: ετικέτες χρήσιμες για προσδιορισμό στόχων branches

Ο συμβολομεταφραστής υπολογίζει την πραγματική μέσα στην εντολή

-Op bne=5

5	19	20	1
6 bits	5 bits	5 bits	16 bits

Παράδειγμα do-while: Δίνεται ο ακόλουθος βρόγχος σε C :

```
do{ g = g + A[i];
          i = i + j;
}while(i!=h);
```

- Υποθέστε ότι
  - ο πίνακας A έχει 100 στοιχεία words (32 bit=word = 4 bytes)
  - μεταγλωττιστής συσχετίζει τις μεταβλητές g, h, i, και j με τους καταχωρητές $17, $18, $19 και $20.
  - διεύθυνση του πίνακα A στο $21.
  - Ο δείκτης του πίνακα i, πρέπει να πολλαπλασιάζεται επί 4

```
Loop :    sll  $9, $19, 2        # Προσωρινός καταχωρητής $9 = i x 4,
          add  $11,$9,$21        # υπολογισμός δείκτη
          lw   $8, 0 ($11)       # Προσωρινός καταχωρητής $8 = A[i]
          add  $17, $17, $8      # g=g+A[i]
          add  $19, $19, $20     # i = i+j
          bne  $19, $18, Loop    # Πήγαινε στο Loop αν   i ≠ h
```

J (jump) εντολή διακλάδωσης χωρίς συνθήκη

Σύνταξη: j label

Σημασία: μετά την εκτέλεση της εντολής τύπου jump η ροή ελέγχου συνεχίζει στην εντολή με το label

Format (j-type):

optype	Target
6-bit	26-bit

Διευθυνσιοδότηση με διακλάδωση και άμεσες τίμες

- τρίτη μορφή εντολών στον MIPS, η **J-type**,
- 6 bits για το πεδίο op (λειτουργίας) και τα υπόλοιπα 26 bits είναι για το πεδίο των διευθύνσεων.

$PC = PC_{31\_28}::IR_{25\_0}::00$

Program counter (PC), ή instruction address register: καταχωρητής ο οποίος αποθηκεύει (κρατεί) την διεύθυνση της επόμενης εντολής που θα εκτελεσθεί.

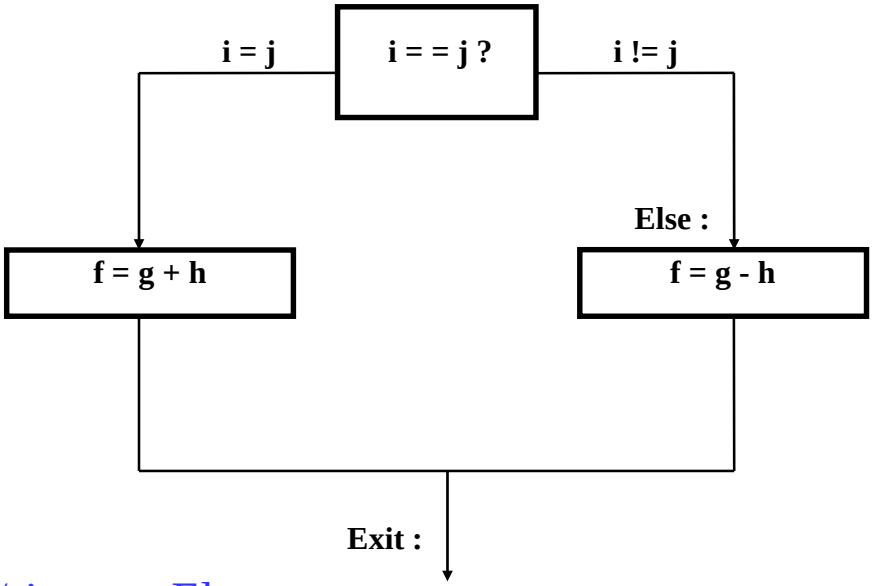
0x04001000 j 10000 # goto location PC_{31_28} :10000x4 = 40000

2	10000
6 bits	26 bits

Διασύνδεση υλικού / λογισμικού

Οι μεταγλωττιστές συχνά δημιουργούν **μεταπηδήσεις και ετικέτες** που δεν παρουσιάζονται στις γλώσσες υψηλού επιπέδου. Για παράδειγμα, χρησιμοποιώντας τις μεταβλητές και τους καταχωρητές του πιο πάνω παραδείγματος, ο πιο κάτω κώδικας σε C:

```
if (i= =j)
     f=g+h;
else
     f=g-h;
```



μεταγλωττίζεται στον πιο κάτω κώδικα MIPS:

```
        bne     $19, $20, Else      # αν i ≠ j goto Else
        add     $16, $17, $18       # f = g + h ( αγνοείται αν i ≠ j )
        j       Next                # go to Next
Else :  sub     $16, $17, $18       # f = g - h ( αγνοείται αν i = j )
Next :
```

Διασύνδεση λογισμικού / υλικού

```
while   (save [i] == k)
        i = i + j;
```

- i, j και k, αντιστοιχούν στους καταχωρητές $19, $20 και $21, επίσης
- ο πίνακας save ξεκινά στην διελυθυνση που περιέχει το $10.

```
Loop:   sll $9, $19, 2          # Προσωρινός καταχωρητής  $9 = i x 4
        add  $9,$10,$9          # save + i x 4
        lw    $8, 0 ($9)        # Προσωρινός καταχωρητής  $8 = save [i]
        bne  $8, $21, Exit      # go to Exit if save [i] != k
        add  $19, $19, $20      # i=i+j
        j     Loop              # goto Loop
Exit:
```

Δίνεται ο πιο κάτω κώδικας σε MIPS που αντιπροσωπεύει ένα while loop σε μια γλώσσα ψηλού επιπέδου:

```
Loop :  sll   $9, $19, 2
        lw    $8, 1200 ($9)
        bne   $8, $21, Exit
        add   $19, $19, $20
        j   Loop
Exit:
```

Εάν η διεύθυνση της πρώτης εντολής είναι το 80000 γράψετε την κωδικοποίηση στο δεκαδικό των Labels Exit και Loop

80000	0	19	9	0	2	0
80004	35	9	8	1200		
80008	5	8	21			
80012	0	19	20	19	0	32
80016	2					
80020	…					

Διασύνδεση λογισμικού / υλικού

- Αν και σχεδόν όλες οι διακλαδώσεις υπό συνθήκη είναι σε κοντινές διευθύνσεις, υπάρχει ανάγκη για διακλάδωση και σε μακρινές διευθύνσεις.
  - Για παράδειγμα η εντολή:

```
beq $18, $19, L1
```

μπορεί να αντικατασταθεί με δύο εντολές, ώς ακολούθως, για να επιτευχθεί η διακλάδωση στη μακρινή διεύθυνση L1:

```
    bne $18, $19, L2
    j     L1
L2:
```

slt (set-less-than)

- Η εντολή **set on less than (slt)**, μας επιτρέπει να κάνουμε σύγκριση, π.χ.
  - **slt $8, $19, $20**

- Ο καταχωρητής $8 παίρνει την τιμή 1, αν η τιμή στον καταχωρητή $19 είναι πιο μικρή από την τιμή στον καταχωρητή $20, διαφορετικά ο καταχωρητής $8 παίρνει την τιμή 0.

- για να ελέξουμε αν η μεταβλητή στον καταχωρητή $16 είναι πιο μικρή από την τιμή στον καταχωρητή $17

```
slt    $1, $16, $17          # S1=1  if  $16 < $17
bne  $1, $0, Less            # goto   Less  if  $1 ≠ $0.
```

- Οι εντολές slt και bne εκτελούν την συνθήκη ελέγχου για το μικρότερο.
- Στην πραγματικότητα ο συμβαλόμεταφραστής του MIPS, μεταφράζει την εντολή *ψευδοεντολή* **blt (branch on less than)** στις δύο πιο πάνω εντολές.
- blt $16,$17,Less - προσοχή στην χρήση του $1

jr (jump register / indirect jump)

- εντολή **jr (jump register)** εκτελεί μεταπήδηση χωρίς συνθήκη στην διεύθυνση (αντίστοιχη ετικέτα) που είναι αποθηκευμένη στον καταχωρητή.

jr $9 **# ο καταχώρητης $9 περιέχει την διεύθυνση της εντολής που θα εκτελεστεί μετά**

Bitwise Λογικοί Τελεστές

Shift left	<<	sll	R-type
Shift right	>>	srl	R-type
Bitwise AND	&	and,andi	R/I-type
Bitwise OR	|	or, ori	R/I-type
Bitwize NOT	~	nor	R-type

sll $10,$16,4 # $10 = $16 << 4 = $16 x 2^4

op	rs	rt	rd	shamt	func
0	0	16	10	4	0

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_2$ 9

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_2$ (9<<4) = 144

Bitwise Λογικοί Τελεστές

A: 0000 0000 0000 0000 0000 0000 0011 1001_2 = 0x00 00 00 39

B: 0000 0000 0000 0000 0000 0000 1001 1000_2 = 0x00 00 00 98

Operation	Result
A| B	
A&B	
Shift left A by 4	
Shift right B by 2	

Unsigned and Signed Τελεστές

- Unsigned: zero-extend άμεση τιμή από 16 σε 32 bit
- Signed: sign-extend άμεση τιμή από 16 σε 32 bit
- Αριθμητικές πράξεις μπορεί να προκαλέσουν exception σε περίπτωση overflow (επιλογή για ορισμό στην αρχιτεκτονική)
- Logical operations immediate zero extend

```
add $1,$2,$3
addi $1,$2,0xf123        $1 = $2 + 0xfffff123
addu $1,$2,$3
addiu $1,$2,0xf123       $1 = $2 + 0x0000f123
ori $1,$2,0xf0           $1 = $2 | 0x000000f0
andi $1,$2,0xff00        $1 = $2 & 0x0000ff00
```

- Η μορφή εντολών **I-type** είναι για **Immediate-type** εντολές (δηλ. εντολές στις οποίες ένας από τους τελεσταίους τους είναι σταθερά). Το πεδίο του MIPS στο οποίο αποθηκεύεται η σταθερά έχει μέγεθος 16 bits.

- Η εντολή πρόσθεσης - add η οποία έχει ένα σταθερό τελεσταίο ονομάζεται add immediate ή addi. Για να προσθέσουμε τον αριθμό 4 στον καταχωρητή 29 γράφουμε απλά:

addi $28, $29, 4 # $28 = $29 + 4

Ο κώδικας μηχανής:

op	rs	rt	immediate
8	29	28	4

001000	11101	11100	0000 0000 0000 0100

- Οι τελεσταίοι άμεσης πρόσβασης/στεθερές (immediate) είναι επίσης πολύ χρήσιμοι για τις συγκρίσεις.
- Όπως έχουμε τονίσει ο καταχωρητής $0 έχει πάντα την τιμή 0, έτσι μπορούμε να συγκρίνουμε με το μηδέν.
- Για σύγκριση με άλλες τιμές, μπορούμε να χρησιμοποιήσουμε την εντολή **slti** (set on less than immediate),
- Για παράδειγμα στις πιο κάτω εντολές θέλουμε να συγκρίνουμε αν το περιεχόμενο του καταχωρητή 18 είναι μικρότερο από το 10 :

```
slti   $8, $18, 10      # $8 = 1      if   $18<10
bne  $8, $0, Less       # goto  Less  if   $18<10
```

LUI Load Upper Immediate

- Η χρήση σταθερών τελεσταίων είναι πολύ συχνή.
- Αν και οι περισσότερες σταθερές χωράνε στο πεδίο των 16bits, μερικές φορές μπορεί να έχουμε και πιο μεγάλες σταθερές.
- Η εντολή του MIPS **load upper immediate (lui),** μεταφέρει τα 16 bits της σταθερά στα πιο ψηλά 16 bits του καταχωρητή, αφήνοντας τα χαμηλότερα 16 bits να προσδιορισθούν από μία άλλη εντολή

- lui Rx, Imm15:0 # Rx = Imm15:0 0000 0000 0000 0000

op	destination	immediate
001111	00000	01000	0000 0000 1111 1111

- Ο κώδικας σε γλώσσα μηχανής για την εντολή lui $8, 255

0000 0000 1111 1111 0000 0000 0000 0000

Τα περιεχόμενα του $8 μετά από την εκτέλεση της εντολής lui

- Ποίος θα είναι ο κώδικας στη συμβολική γλώσσα MIPS για να φορτώσουμε την πιο κάτω 32-bit σταθερά στο καταχωρητή $16:

  0000 0000 0011 1101 0000 1001 0000 0000 = 0x003D0900

- Ποίος θα είναι ο κώδικας στη συμβολική γλώσσα MIPS για να φορτώσουμε την πιο κάτω 32-bit σταθερά στο καταχωρητή $16:

  0000 0000 0011 1101 0000 1001 0000 0000 = 0x003D0900

	$16
lui $16, 0x3D	
addiu $16,$16,0x900	

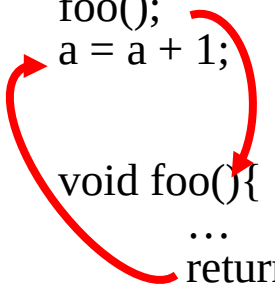
- Ποίος θα είναι ο κώδικας στη συμβολική γλώσσα MIPS για να φορτώσουμε την πιο κάτω 32-bit σταθερά στο καταχωρητή $16:

  0000 0000 0011 1101 0000 1001 0000 0000

	$16
lui $16, 0x3D	0x003D0000
addiu $16,$16,0x900	0x003D0900

Υποστήριξης διαδικασιών/συναρτήσεων σε επίπεδο υλικού

- Μια ΑΣΕ πρέπει να παρέχει ένα τρόπο μεταφοράς της ροής εκτέλεσης του προγράμματος σε μια διαδικασία και ακολούθως να επιτρέπει την επιστροφή στην επόμενη εντολή (αμέσως μετά από την εντολή που κάλεσε τη διαδικασία).



```
foo();
a = a + 1;


void foo(){
        …
        return;
}
```

- Η αρχιτεκτονική MIPS υποστηρίζει την εντολή **jump - and - link (jal)** που
  - αποθηκεύει τη διεύθυνση της εντολής που ακολουθεί το jal στον καταχωρητή $31 ($ra)
  - μεταπηδά σε μία διεύθυνση

jal ProcedureAddress # $31 = PC+4, PC = ProcedureAddress (Label)

- Η εντολή που μεταπηδά στην επιστροφή (return) είναι η **jr $31**

- **Στην περίπτωση που έχουμε φωλιασμένα (nested) jal τότε είναι απαραίτητο να φυλαχτεί η αρχική τιμή του $ra πριν να εκτελέσουμε τα jal που ακολουθούν.**
  - Ο ιδανικός τρόπος για επίτευξη αυτής της λειτουργίας είναι η χρησιμοποίηση της στοίβας (stack). Last-In-First-Out (LIFO)

- Χρειάζεται ένας δείκτης που θα δείχνει στην κορυφή της στοίβας, δηλαδή εκεί που η επόμενη εντολή πρέπει να τοποθετήσει τους καταχωρητές της. Stack Pointer (sp)

- Η τοποθέτηση δεδομένων στη στοίβα ονομάζεται **push**, ενώ η μετακίνηση τους από την στοίβα **pop**.

Υποστήριξης διαδικασιών/συναρτήσεων σε επίπεδο υλικού

```
100   foo();
104   a = a + 1;


200   void foo(){

          …
240       x= foo1();
244       x = x + y;
          …
280       return;
      }


300   int foo1(){
          ….
364       return k;
      }
```

PC	ra	sp
100		0

Stack

0	
1	
2	
3	

Υποστήριξης διαδικασιών/συναρτήσεων σε επίπεδο υλικού

```
100    foo();
104    a = a + 1;


200    void foo(){

           …
240        x= foo1();
244        x = x + y;
           …
280        return;
       }


300    int foo1(){
           ….
364        return k;
       }
```

PC	ra	sp
100	104	0
200		

Stack

0	
1	
2	
3	

Υποστήριξης διαδικασιών/συναρτήσεων σε επίπεδο υλικού

```
100    foo();
104    a = a + 1;



200    void foo(){

           …
240        x= foo1();
244        x = x + y;
           …
280        return;
       }


300    int foo1(){
           ….
364        return k;
       }
```

PC	ra	sp
100	104	0
200		
…		
240		

Stack

0	
1	
2	
3	

Υποστήριξης διαδικασιών/συναρτήσεων σε επίπεδο υλικού

```
100    foo();
104    a = a + 1;



200    void foo(){

           …
240        x= foo1();
244        x = x + y;
           …
280        return;
       }



300    int foo1(){
           ….
364        return k;
       }
```

PC	ra	sp
100	104	0
200	244	1
…		
240		
300		

Stack

0	104
1	
2	
3	

Υποστήριξης διαδικασιών/συναρτήσεων σε επίπεδο υλικού

```
100    foo();
104    a = a + 1;



200    void foo(){

           …
240        x= foo1();
244        x = x + y;
           …
280        return;
       }



300    int foo1(){
           ….
364        return k;
       }
```

PC	ra	sp
100	104	0
200	244	1
…		
240		
300		
…		
364		

Stack

0	104
1	
2	
3	

Υποστήριξης διαδικασιών/συναρτήσεων σε επίπεδο υλικού

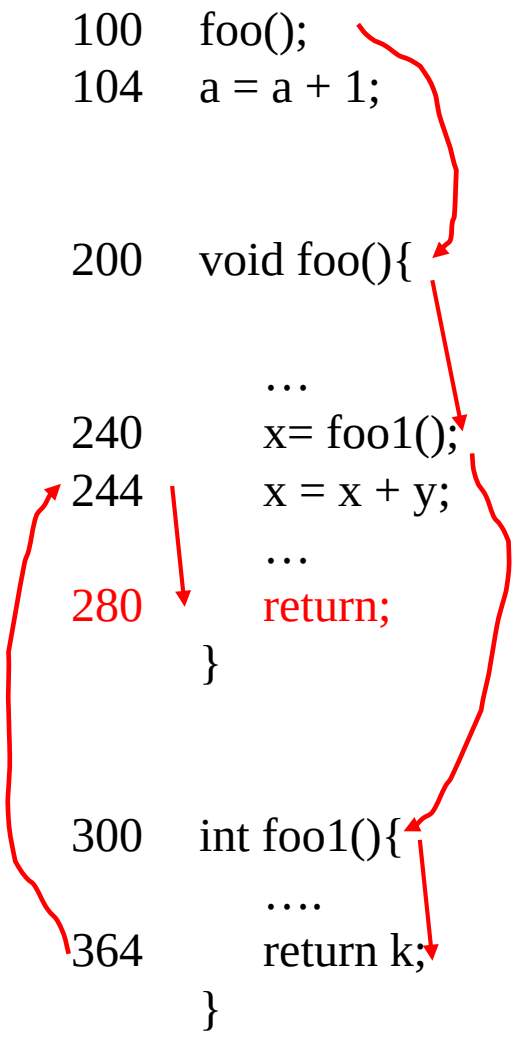
```
100    foo();
104    a = a + 1;



200    void foo(){


           …
240        x= foo1();
244        x = x + y;
           …
280        return;
       }



300    int foo1(){
           ….
364        return k;
       }
```

PC	ra	sp
100	104	0
200	244	1
…	104	0
240		
300		
…		
364		
244		

Stack

0	104
1	
2	
3	

Υποστήριξης διαδικασιών/συναρτήσεων σε επίπεδο υλικού

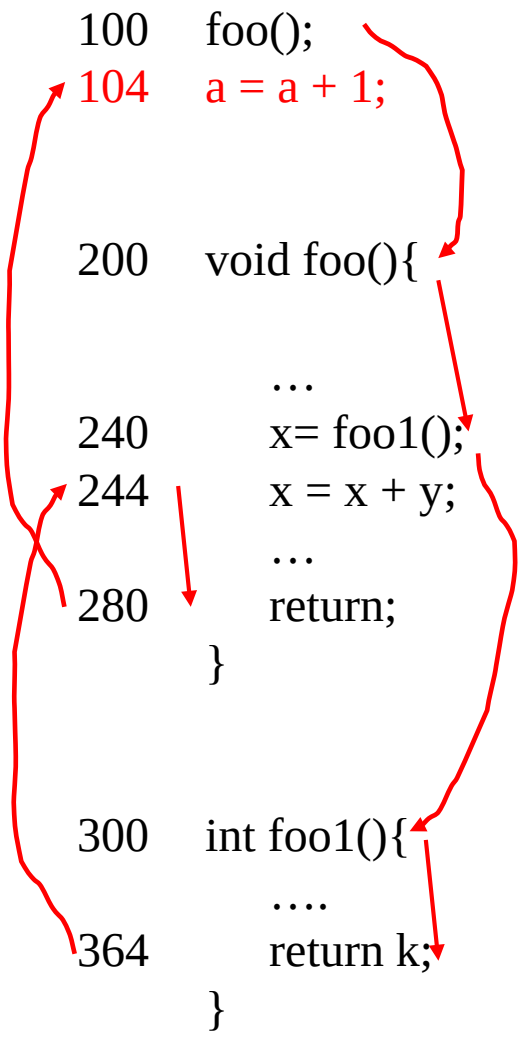


PC	ra	sp
100	104	0
200	244	1
…	104	0
240		
300		
…		
364		
244		
…		
280		

Stack

0	104
1	
2	
3	

Υποστήριξης διαδικασιών/συναρτήσεων σε επίπεδο υλικού



PC	ra	sp
100	104	0
200	244	1
…	104	0
240		
300		
…		
364		
244		
…		
280		
104		

Stack

0	104
1	
2	
3	

Name	Format	Example						Comments
Add	R	0	2	3	1	0	32	add $1, $2, $3
Sub	R	0	2	3	1	0	34	sub $1, $2, $3
Addo	I	8	2	1	100			addi $1, $2, 100
Lw	I	35	2	1	100			lw $1, 100($2)
Sw	I	43	2	1	100			sw $1, 100($2)
Lui	I	15	0	1	100			lui $1,100
Beq	I	4	1	2	25			beq $1, $2, 100
Bne	I	5	1	2	25			bne $1, $2, 100
Slt	R	0	2	3	1	0	42	slt $1, $2, $3
Slti	I	10	2	1	100			slti $1, $2, 100
J	J	2	2500					j 10000
Jr	R	0	31	0	0	0	8	jr $31
Jal	J	3	2500					jal 10000
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
Format R	R	op	rs	rt	rd	Shamt	funct	Arithmetic Instruction format
Format I	I	op	rs	rt	Address / immediate			Transfer, branch, Imm. Format
Format J	J	op	Target address					Jump instruction format

Πολλά χαρακτηριστικά των εντολών της μηχανής MIPS οφείλονται στη κανονικοποίηση, όπως:

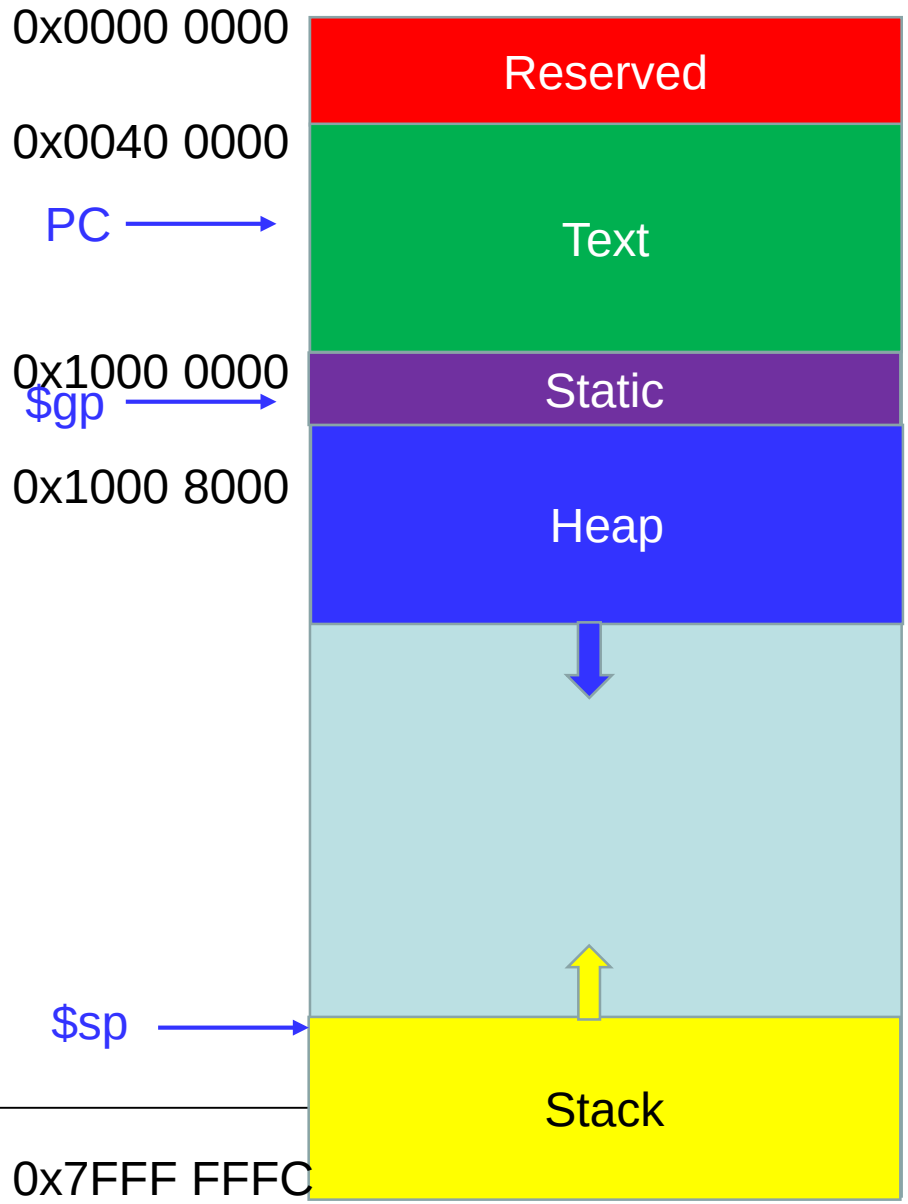
- όλες οι εντολές έχουν το ίδιο μέγεθος,
- απαιτούνται πάντα τρείς καταχωρητές για τις αριθμητικές εντολές και τα πεδία των καταχωρητών κρατούνται στην ίδια θέση για όλα τα είδη εντολών.

Memory Model MIPS

Text: κώδικας
Static: global variables, constants
Heap: dynamic
Stack: local

User Address Space
0x0000 0000 – 0x7FFF FFFF

Kernel Address Space
0x8000 0000 – 0xFFFF FFFF



Στοίβα

- Χώρος στην μνήμη για δέσμευση των τοπικών μεταβλητών (παραμέτρων, τιμή επιστροφής) κάθε συνάρτησης
- Πόσος χώρος; Εξαρτάται από το είδος και πλήθος μεταβλητών συνάρτηση
  - Επίσης τυποποίηση για προστασία καταχωρητών
- Δεσμεύεται ο χώρος όταν καλείται η συνάρτηση
- Αποδεσμεύεται όταν επιστρέφει η συνάρτηση
- $sp καταχωρήτης που δείχνει στην κορυφή της στοίβας
  - Δέσμευση αφαίρεση από το $sp
  - Αποδέσμευση πρόσθεση στο $sp

Τυποποίηση για Κλήση Συναρτήσεων (Calling Convention)

Χρήση τυποποίησης για μείωση κόστους προστασίας συναρτήσεων

1. **Αποθηκεύει αυτός που καλεί (caller save).** Η διαδικασία που καλεί είναι υπεύθυνη για το φύλαγμα και την επαναφορά των καταχωρητών, οι οποίοι πρέπει να φυλαχτούν κατά την διάρκεια της εκτέλεσης της διαδικασίας. Τότε η διαδικασία που καλείται μπορεί να αλλάξει οποιοδήποτε καταχωρητή.
2. **Αποθηκεύει αυτός που καλείται (call/callee save).** Φυλάει και επαναφέρει αυτός που καλείται. Η διαδικασία που καλείται φυλάει τους καταχωρητές, τους χρησιμοποιεί όπως θέλει και επαναφέρει τους αρχικούς στο τέλος του καλέσματος.
3. **Ο συνδυασμός των πιο πάνω**

Τυποποίηση MIPS: η Μέση Λύση

- Registers divided into sets:
  - Preserved on Call
  - Not Preserved on Call
- Preserved on Call
  - υποσύνολο καταχωρήτων που αυτός που καλεί ξέρει πως αυτοί θα διατηρηθούν με ευθύνη αυτού που καλείται
  - Αυτός που καλείται εάν αλλάζει preserved on call registers πρέπει να μεριμνήσει να τους διατηρήσει στην μνήμη και να τους επαναφέρει μέσα στους καταχώρητες πριν επιτρέψει
- Not Preserved on Call
  - υποσύνολο καταχωρήτων που αυτός που καλεί εάν τους χρειάζεται να διατηρηθούν πρέπει να τους αποθηκεύσει στην μνήμη πριν την κλήση και να τους επαναφέρει στους καταχωρητές μετά την επιστροφή
  - Αυτός που καλείται μπορεί να τους χρησιμοποιήσει χωρίς οποιαδήποτε περεταίρω ενέργεια

Τυποποίηση
Policy of Use Conventions

Register number		Usage	Preserved on call?
$zero	0	the constant value 0	n.a.
$v[0-1]	2-3	values for results and expression evaluation	no
$a[0-3]	4-7	arguments	no
$t[0-7]	8-15	temporaries	no
$s[0-7]	16-23	saved	yes
$t[8-9]	24-25	more temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

Register 1 ($at) reserved for assembler, 26-27 for operating system

Τυποποίηση Κλήσης

- Οι καταχωρητές που είναι preserved on call
  - όταν μια συνάρτηση μπορεί να αλλάξει την τιμή τους τότε πρέπει να αποθηκεύονται στην αρχή της συνάρτησης και να τους αποκαθιστά στο τέλος
  - sp και fp εξαίρεση (αφαιρεση στην αρχή /πρόσθεση στο τέλος)
- Οι άλλοι καταχωρητές
  - όταν μια συνάρτηση καλεί άλλη συνάρτηση και χρειάζεται να διατηρήσει τις τιμές σε προσωρινούς τότε πρέπει να αποθηκεύονται πριν την κλήση της άλλης συνάρτησης και να φορτώνονται μετά την επιστροφή

  σε leaf συναρτήσεις
  - temporaries για όλες τις μεταβλητές (μόνο αν χρειάζεται saved),

  αλλιώς σε non-leaf συναρτήσεις:
  - temporaries για προσωρινές μεταβλητές
  - saved για τοπικές

- Για υποστήριξη συναρτήσεων χρειαζόμαστε μεταφορά παραμέτρων σε μία διαδικασία.
    1. Στην αρχιτεκτονική MIPS η μέθοδος αυτή γίνεται με την τοποθέτηση των παραμέτρων στους καταχωρητές $4 έως $7. (a[0],a[1],a[2],a[3])
    2. Δεν είναι preserved on call, έτσι εάν μία διαδικασία χρειάζεται να καλέσει μια άλλη διαδικασία, αυτοί οι καταχωρητές μπορούν να φυλαχθούν και να τους επαναφέρομαι στη συνέχεια χρησιμοποιώντας τη στοίβα.

- Για τιμές επιστροφής χρησιμοποιούνται οι καταχωρητές $2 και $3 (v[0],v[1])

- Κάθε συνάρτηση που δεν είναι φύλλο αποθηκεύει στην στοίβα το return address register ($ra)

- Κάθε συνάρτηση δεσμεύει χώρο στην στοίβα για τις τοπικές της μεταβλητές

Τυποποίηση για FP registers in the lab

Συναρτήσεις, πέρασμα και επιστροφή Παραμέτρων

- Γράψετε συνάρτηση count1 που μετρά και επιστρέφει τα 1 που περιέχει η ακέραια παράμετρος x
- (Πχ εάν x=37 θα επιστρέψει 3 γιατί στο δυαδικό το 37 είναι το 0100101)

Συναρτήσεις, πέρασμα και επιστροφή Παραμέτρων

- Γράψετε συνάρτηση count1 που μετρά και επιστρέφει τα 1 που περιέχει η ακέραια παράμετρος x

```
int count1(int x){
        int n = 0, t;
        while(x!=0){
                t = x & 1;
                n=n+t;
                x=x>> 1;
        }
        return n;
}
```

Συναρτήσεις, πέρασμα και επιστροφή Παραμέτρων

- Γράψετε συνάρτηση count2 που μετρά και επιστρέφει το άθροισμά των 1 που περιέχουν οι ακέραιες παράμετροι της x και y

```
int count1(int x){
        int n = 0, t;
        while(x!=0){
                t = x & 1;
                n=n+t;
                x=x>> 1;
        }
        return n;
}
```

Συναρτήσεις, πέρασμα και επιστροφή Παραμέτρων

- Γράψετε συνάρτηση count2 που μετρά και επιστρέφει το άθροισμά των 1 που περιέχουν οι ακέραιες παράμετροι της x και y

```
int count1(int x){
        int n = 0, t;
        while(x!=0){
                t = x & 1;
                n=n+t;
                x=x>> 1;
        }
        return n;
}
```

```
int count2(int x, int y){

}
```

Συναρτήσεις, πέρασμα και επιστροφή Παραμέτρων

- Γράψετε συνάρτηση count2 που μετρά και επιστρέφει το άθροισμά των 1 που περιέχουν οι ακέραιες παράμετροι της x και y

```
int count1(int x){
        int n = 0, t;
        while(x!=0){
                t = x & 1;
                n=n+t;
                x=x>> 1;
        }
        return n;
}
```

```
int count2(int x, int y){
        int t = count1(x)+count1(y);
        return t;
}
```

```
int count1(int x){
        int n = 0, t;
        while(x!=0){
                t = x & 1;
                n=n+t;
                x=x>> 1;
        }
        return n;
}
```

Variable to Register Mapping

x => a0

n => v0

t => t0

x => a0
n => v0
t => t0

```
int count1(int x){
        int n = 0, t;
        while(x!=0){
                t = x & 1;
                n=n+t;
                x=x>> 1;
        }
        return n;
}
```

```
count1: add         ,$0,$0
loop1:  beq         ,$0,end
        andi        ,       ,1
        addu
        srl         ,       ,1
        j
end:    jr
```

```
count1: add   $v0,$0,$0
loop1:  beq   $a0,$0,end
        andi  $t0,$a0,1
        addu  $v0,$v0,$t0
        srl   $a0,$a0,1
        j     loop1
end:    jr    $ra
```

x => a0
y => a1
t => v0

```
int count2(int x, int y){
        int t = count1(x)+count1(y);
        return t;
}
```

x => a0
y => a1
t => v0

```
int count2(int x, int y){
        int t = count1(x);
        t  = t +count1(y);
        return t;
}
```

```
count2:   add   $sp,$sp, -
          sw    $ra,0($sp)

          jal   count1          # pass a0, return v0

          jal   count1          # pass a1, return v0

          lw    $ra,0($sp)
          add   $sp,$sp,
          jr    $ra
```

```
int count2(int x, int y){
        int t = count1(x);
        t  = t +count1(y);
        return t;
}
```

x => a0
y => a1
t => v0

```
count2:  add   $sp,$sp, -12
         sw    $ra,0($sp)
         sw    $s0,4($sp)
         sw    $a1,8($sp)
         jal   count1
         add   $s0,$0,$v0
         lw    $a0,8($sp)
         jal   count1
         add   $v0,$s0,$v0
         lw    $s0,4($sp)
         lw    $ra,0($sp)
         add   $sp,$sp, 12
         jr    $ra
```

```
count1:  add   $v0,$0,$0
loop1:   beq   $a0,$0,end
         andi  $t0,$a0,1
         addu  $v0,$v0,$t0
         srl   $a0,$a0,1
         j     loop1
end:     jr    $ra
```

Συναρτήσεις φύλλα

```
int g( int x, int y ) {
 return (x + y);
}

g:
add $v0,$a0,$a1      # result is sum of args
 jr $ra              # return
```

Επεξέργασία με Πίνακες

```
void clear1 (int array[ ], int size)
        {   int i ;
        for  (i = 0; i < size; i =i + 1)
              array[i] = 0;  }
```

- clear1: Υποθέτουμε ότι οι δύο παράμετροι array και size βρίσκονται στους καταχωρητές $a0 και $a1, αντίστοιχα. Η μεταβλητή i βρίσκεται στο καταχωρητή $t0 ΚΑΙ πως πάντοτε το size>0

Ο κώδικας της διαδικασίας clear1 στη γλώσσα MIPS είναι:

```
        move    $t0, $zero      # i = 0
loop1:  sll     $t1, $t0, 2     # temp = i  x 4
        add     $t1, $a0, $t1   # temp = address of array[i]
        sw      $0, 0($t1)      # array[i] = 0
        addi    $t0, $t0,  1    # i = i + 1
        blt     $t0,$a1,loop1   # if (i<size) repeat
```

ΚΑΛΥΤΕΡΟΣ κώδικας της διαδικασίας clear1 στη γλώσσα MIPS είναι:

```
        move    $t0, $a0          # p= &array[0]
        sll     $t1,$a1,2         # size*4
        add     $t1,$t1,$t0       # base+size*4
loop1:  sw      $0, 0($t0)        # array[i] = 0
        addi    $t0, $t0,  4      # p = p + 4
        blt     $t0,$t1,loop1     # if (p<&array[size]) repeat
```

System Calls (SPIM)

- Support for some small number of useful services
  - Είσοδος, έξοδος
- Χρήση εντολής syscall
- Ποιο syscall καθορίζεται από την τιμή στον $v0
- Οι παράμετροι του syscall στους $a0-3
- Η τιμή επιστροφής στον $v0

```
.data

str1:

.asciiz “Output = “

.text

addi    $v0,$0,4
la      $a0,str1
syscall
addi    $v0,$0,1
addi    $a0,$0,5
syscall
```

System Calls (SPIM)

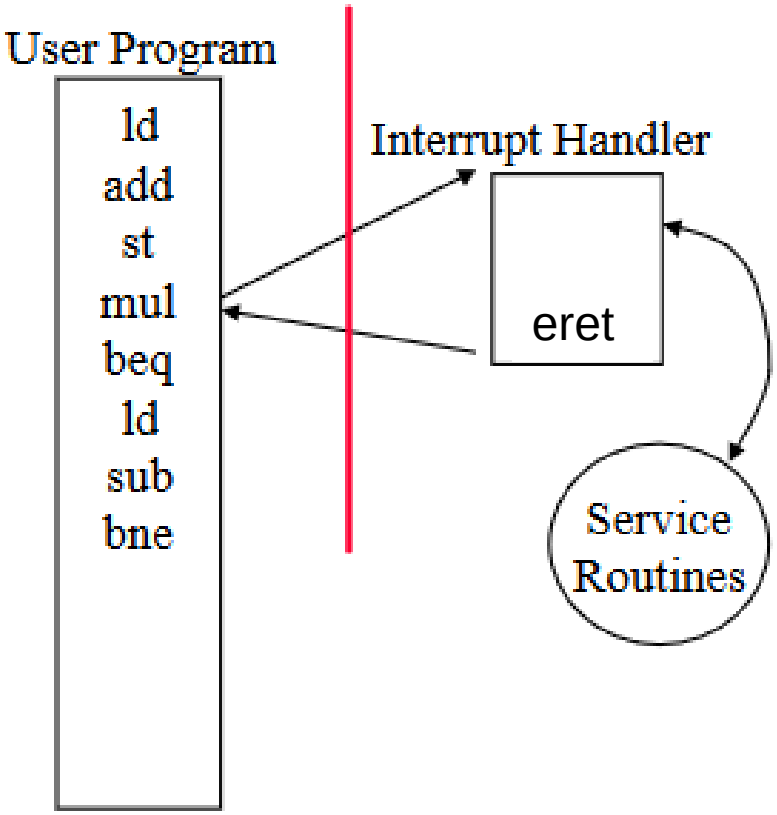
	κωδικός	ορίσματα	αποτέλεσμα
print_int	1	$a0	
print_string	4	$a0	
read_int	5		$v0
read_string	8	$a0=address, $a1= length	
sbrk	9	$a0= ποσότητα	$v0 (address)
exit	10		
print_char	11	$a0 = character	
read_char	12		$v0

Σωρό (Heap)

Exceptions και Interrupts

- Exceptions:προκαλούνται από λάθη/εξαιρετικά γεγονότα κατά την εκτέλεση εντολών
- Interrupts: προκαλούνται από μονάδες Ε/Ε (I/O), debugging breakpoints
- Αντιμετωπίζονται με exception handlers
  - Μικρές ρουτίνες
- MIPS χρήση *coprocessor registers* για αντιμετώπιση interrupts και exceptions
  - Επιπλέον καταχωρητές στην ΑΣΕ

Program Interrupted



Interrupt/Exception Handler should not change user program state

Interrupts/exceptions seem as arbitrary function calls in the middle of your program

Exception Causing Instructions (Synchronous)

```
# divide by zero -- breakpoint
div $t0, $t0, $zero

# arithmetic overflow
li $t1, 0x7fffffff
addi $t1, $t1, 1

# non-existent memory address -- bus error
sw $t2, 124($zero)

# non-aligned address -- address error
sw $t2, 125($sp)
```

Interrupts from external devices (Asynchronous)

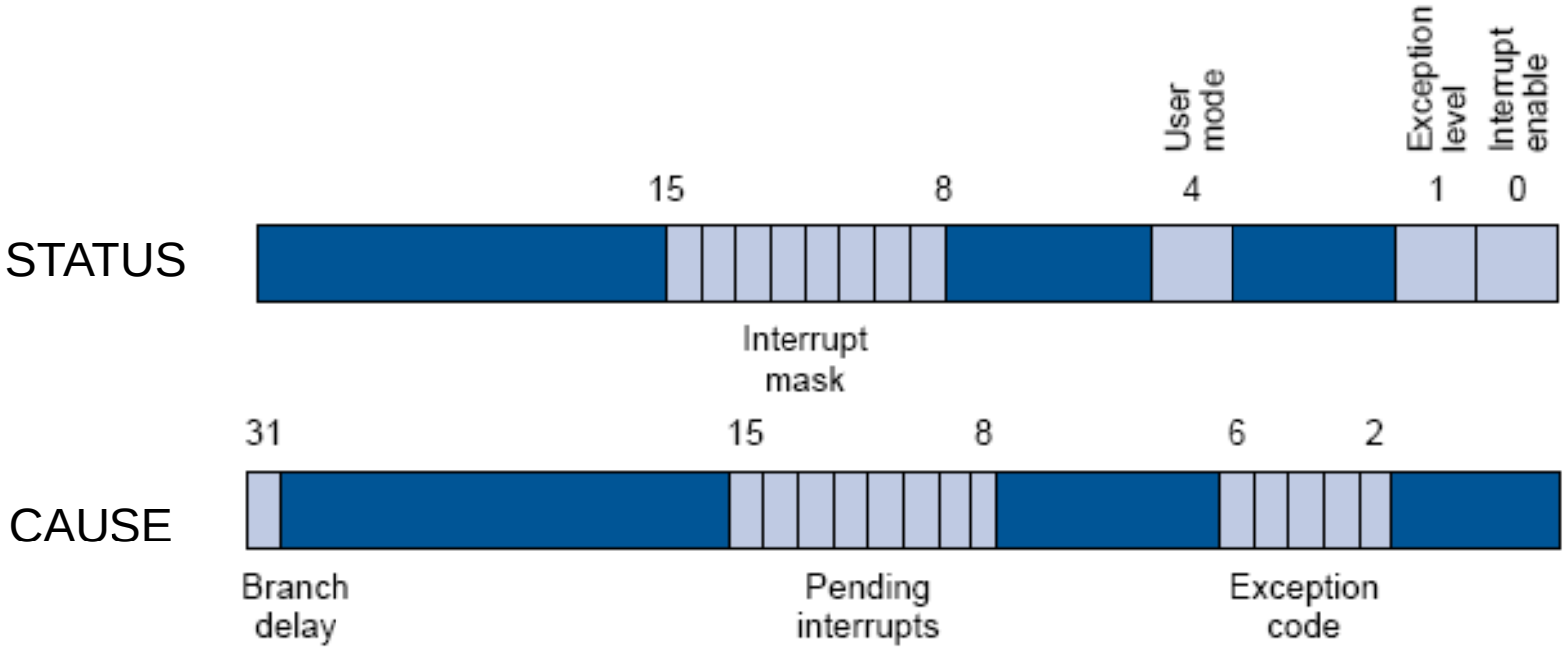
- User pressed a key on console
- Console ready to display another character
- Hardware timer expired
- Disk read complete (πχ DMA)
- Memory error (πχ ECC RAM)

Coprocessor Registers

Register name	Register number	Usage
BadVAddr	8	memory address at which an offending memory reference occurred
Count	9	timer
Compare	11	value compared against timer that causes interrupt when they match
Status	12	interrupt mask and enable bits
Cause	13	exception type and pending interrupt bits
EPC	14	address of instruction that caused exception
Config	16	configuration of machine

- Πρόσβαση με χρήση των εντολών ***mfc0*** και ***mtc0***
  - ***Επέκταση ΑΣΕ***
- Μετά από ένα exception το EPC περιέχει την διεύθυνση της “προβληματική” εντολής

Status και Cause Registers



- Για διακρίβωση τι προκαλεί ένα interrupt/exception
- Για καθορισμό των επιτρεπόμενων interrupts και exceptions (επίσης προτεραιότητα)
  - 6 level of interrupts and 2 software levels (user/kernel)
  - MASK bits: 1 enable interrupts, 0 disable interrupts

STATUS Register

- Exception Level bit: normally 0 unless there is an exception
- When EL is 1 no interrupts/exceptions are allowed and the EPC is not updated by subsequent exceptions
  - The above help to keep exception handler uninterrupted, EL is set to 0 by exception handler (when done)
- IE=1 permit interrupts, IE=0 interrupts disabled

Τι προκαλεί exceptions και ο κωδικός του στο Cause register

Number	Name	Cause of exception
0	Int	interrupt (hardware)
4	AdEL	address error exception (load or instruction fetch)
5	AdES	address error exception (store)
6	IBE	bus error on instruction fetch
7	DBE	bus error on data load or store
8	Sys	syscall exception
9	Bp	breakpoint exception
10	RI	reserved instruction exception
11	CpU	coprocessor unimplemented
12	Ov	arithmetic overflow exception
13	Tr	trap
15	FPE	floating point

- Μετά από exception η εκτέλεση συνεχίζει στην διεύθυνση 0x80000180 (kernel address space)

Τι συμβαίνει μετά…

- Το exception handler αναγνωρίζει τι προκάλεσε το exception ή interrupt και εκτελεί ανάλογο routine (service) ή τερματίζοντας την διεργασία που το προκάλεσε, π.χ.:
  - εκτέλεση μη αρχιτεκτονικής εντολής => kill
  - Page fault => service για να φέρουμε δεδομένα από τον δίσκο
  - User/system configuration => Αγνοεί το exception
    - Πχ αριθμητική υπερχείλιση

```
.ktext 0x80000180
mov  $k1, $at      # Save $at register
sw   $a0, save0    # Handler is not re-entrant and can't use
sw   $a1, save1    # stack to save $a0, $a1
                   # Don't need to save $k0/$k1

mfc0    $k0, $13        # Move Cause into $k0

srl     $a0, $k0, 2     # Extract ExcCode field
andi    $a0, $a0, 0xf

beqz    $a0,$0,done     # Branch if ExcCode is Int (0)


mov     $a0, $k0        # Move Cause into $a0
mfc0    $a1, $14        # Move EPC into $a1
jal     print_excp      # Print exception error message
```

Απλοποιημένο παράδειγμα

κώδικας που διαχειρίζεται τις διάφορες περιπτώσεις

```
done:  mfc0    $k0, $14      # Bump EPC
       addiu   $k0, $k0, 4   # Do not reexecute
                             # faulting instruction
       mtc0    $k0, $14      # EPC

       mtc0    $0, $13       # Clear Cause register

       mfc0    $k0, $12      # Fix Status register
       andi    $k0, 0xfffd   # Clear EXL bit
       ori     $k0, 0x1      # Enable interrupts
       mtc0    $k0, $12
       lw      $a0, save0    # Restore registers
       lw      $a1, save1
       mov     $at, $k1

       eret                  # Return to EPC
```

από Patterson και Hennessy)

```
.ktext 0x80000180
mov  $k1, $at      # Save $at register
sw   $a0, save0    # Handler is not re-entrant and can't use
sw   $a1, save1    # stack to save $a0, $a1
                   # Don't need to save $k0/$k1

mfc0    $k0, $13        # Move Cause into $k0

srl     $a0, $k0, 2     # Extract ExcCode field
andi    $a0, $a0, 0xf

beqz    $a0,$0,done     # Branch if ExcCode is Int (0)


mov     $a0, $k0        # Move Cause into $a0
mfc0    $a1, $14        # Move EPC into $a1
jal     print_excp      # Print exception error message
```

Απλοποιημένο παράδειγμα

κώδικας που διαχειρίζεται τις διάφορες περιπτώσεις

```
done:  mfc0    $k0, $14      # Bump EPC
       addiu   $k0, $k0, 4   # Do not reexecute
                             # faulting instruction
       mtc0    $k0, $14      # EPC

       mtc0    $0, $13       # Clear Cause register

       mfc0    $k0, $12      # Fix Status register
       andi    $k0, 0xfffd   # Clear EXL bit
       ori     $k0, 0x1      # Enable interrupts
       mtc0    $k0, $12
       lw      $a0, save0    # Restore registers
       lw      $a1, save1
       mov     $at, $k1

       eret                  # Return to EPC
```

Why we save $at in $k1?
Why we save $a0 and $a1?
Where do we save $a0 and $a1?
Why we do not save $k0 and $k1?

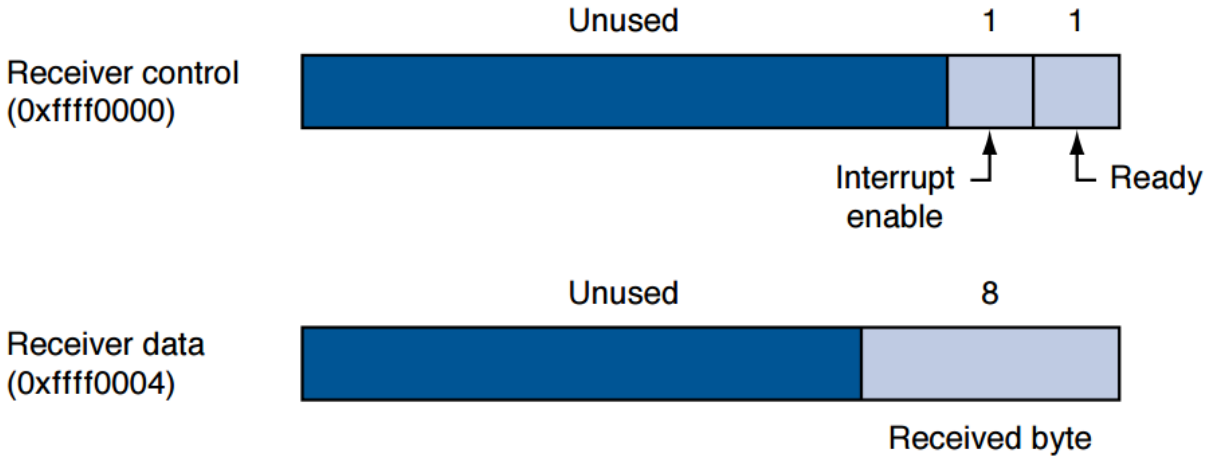
από Patterson και Hennessy)

- Why we save $at in $k1?
  - Handler uses pseudo-instructions and they will overwrite $at
- Why we save $a0 and $a1?
  - To have some registers to work with (e.g. to call a routine). You can save more if needed.
- Where do we save $a0 and $a1?
  - Not in the stack! It can be that the reason we had an exception is that the $sp got corrupted. We do not want exception handler to modify user/system data. We store them in kernel data segment reserved for this purpose

```
.kdata
save0: .space 4
save1: .space 4
```

- Why we do not save $k0 and $k1?
  - They are reserved for system (i.e. for exception and interrupts)

Είσοδος



Ready = 1 (char from keyboard ready and not read)
0 to 1 when a character is typed on keyboard
1 to 0 when character read from “Receiver Data”

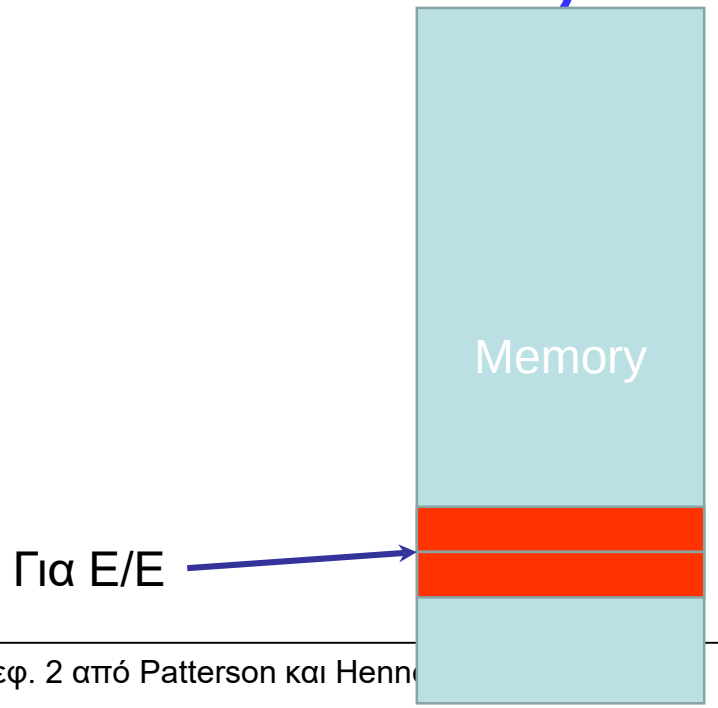
```
# routine called from handler when interrupted from input
        li $t0, 0xffff0004
        lw $v0, 0($t0)
        lw $t1, -4($t0)
        andi $t1, $t1, 0x02
        sw $t1, -4($t0)
        ret
```

Εάν το IE=1 τότε όποτε το Ready γίνεται 1 (έφθασαν νέα δεδομένα)

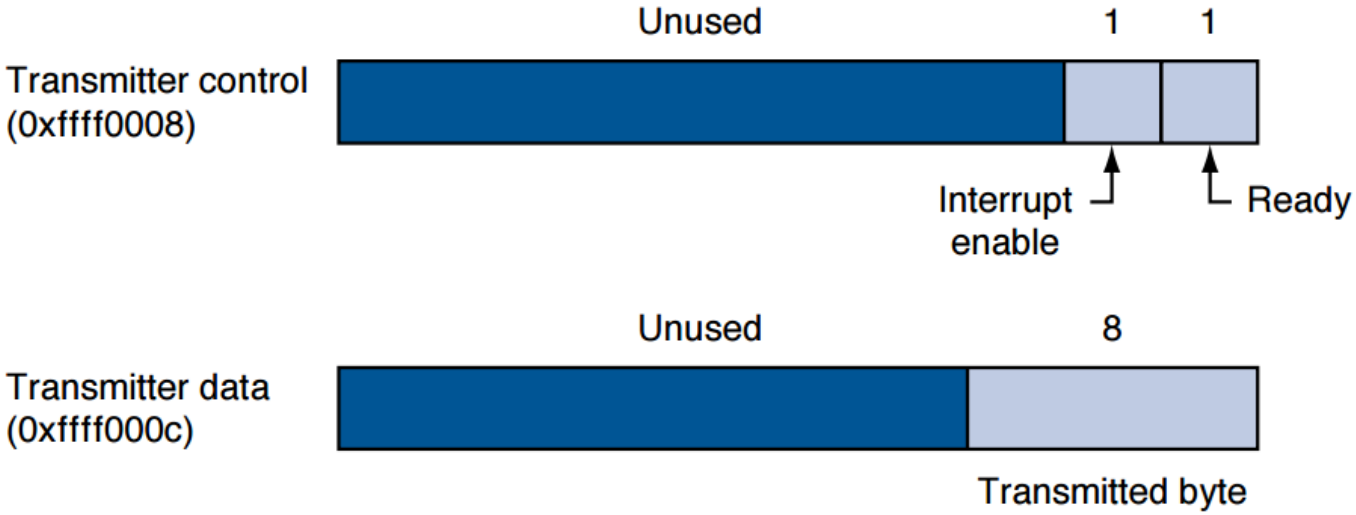
Το IE μπορεί να το ενημερώσει το πρόγραμμα (set to 0 or 1)

Memory Mapped I/O

- I/O Devices are identified/associated with specific addresses in memory of a process
- You do not read/write to memory



Έξοδος

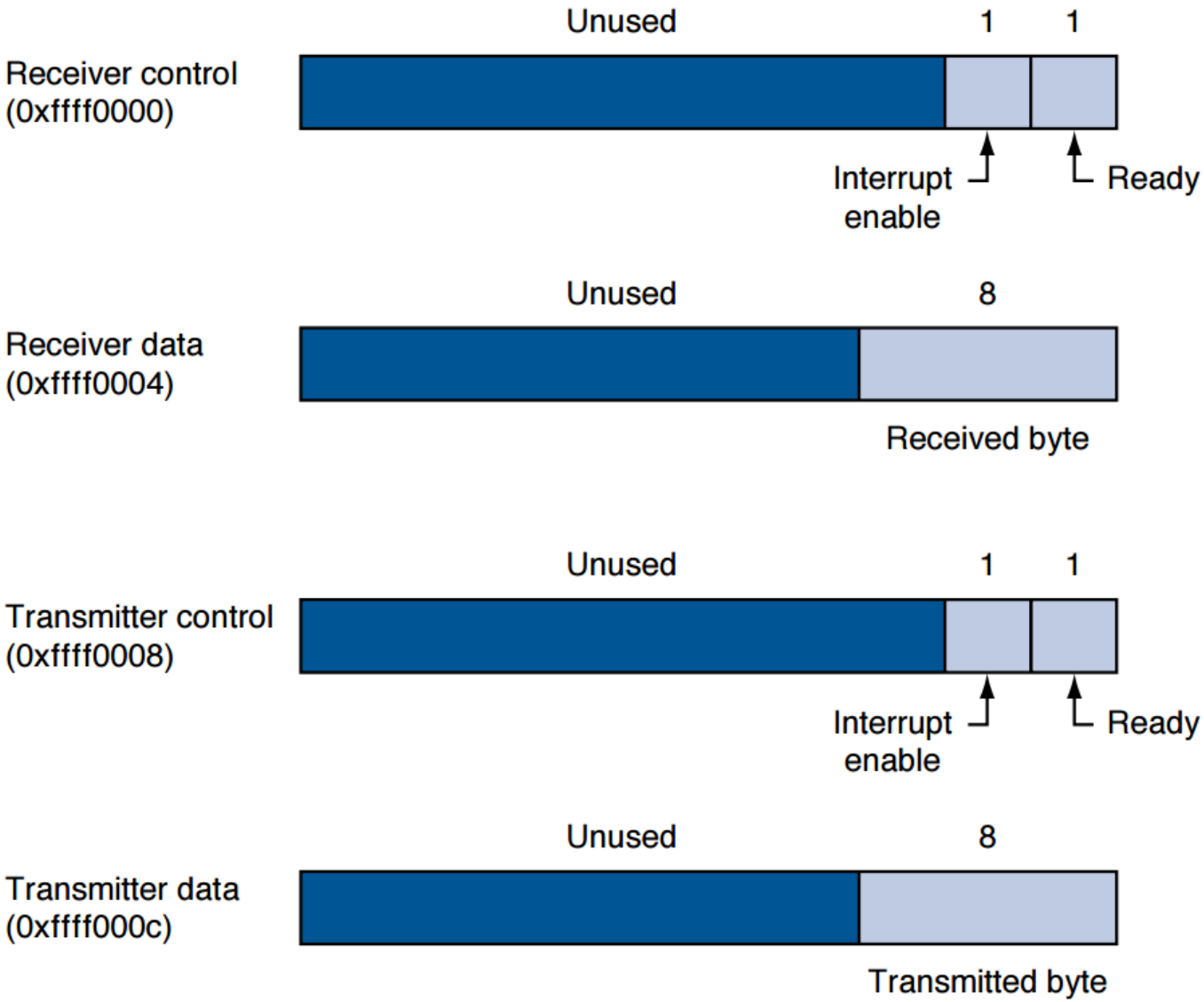


Εάν το IE=1 τότε όποτε το Ready γίνεται 1 έτοιμο για νέα δεδομένα

Το IE μπορεί να το ενημερώσει το πρόγραμμα

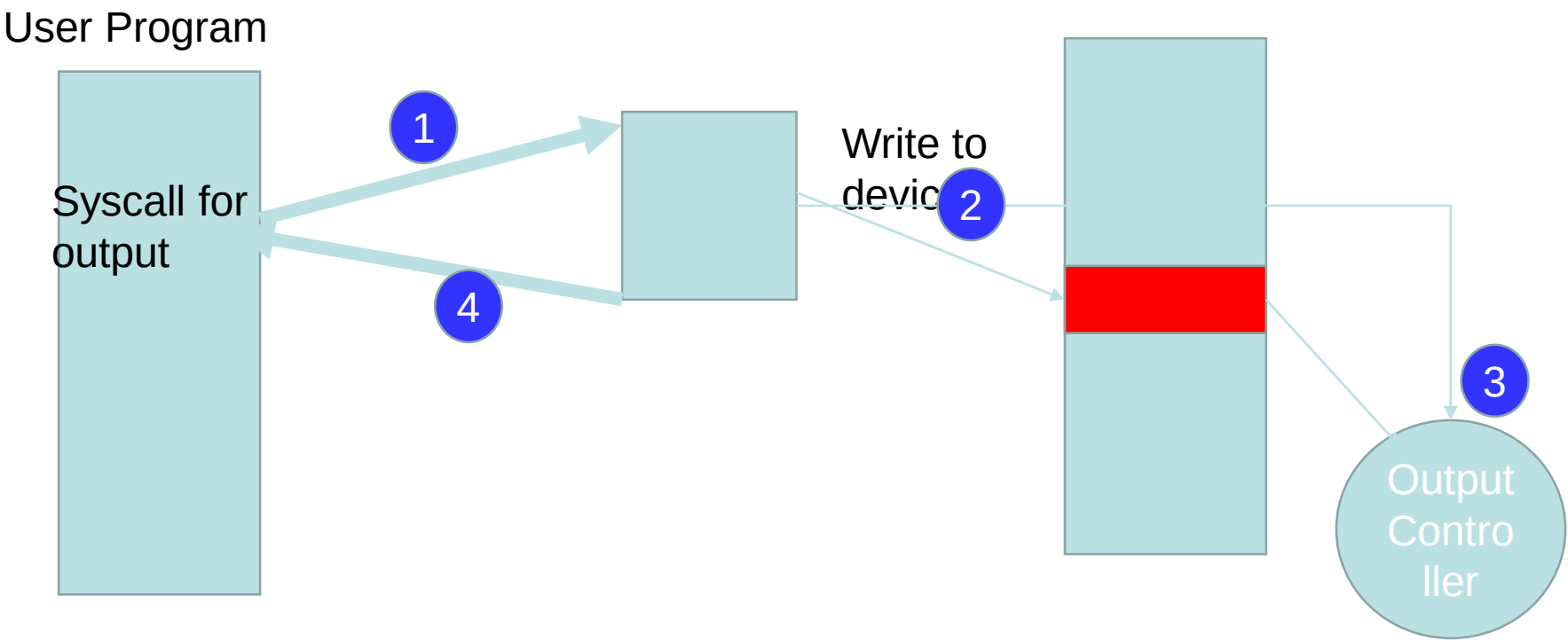
When Data is written RB is set to 0
RB should be 1 to write new data

```
# character in $a0, spin until device ready
            li $t0, 0xffff0008
wr poll:
            lw $v0, 0($t0)
            andi $v0, $v0, 0x01
            beq $v0, $zero, wr poll
            sw $a0, 4($t0)
```


Unused
1
1
Receiver control
(0xffff0000)
Interrupt
enable
Ready
Unused
8
Receiver data
(0xffff0004)
Received byte
Unused
1
1
Transmitter control
(0xffff0008)
Interrupt
enable
Ready
Unused
8
Transmitter data
(0xffff000c)
Transmitted byte

Actually reading and writing independent of processor

- Systems delegate input and output to IO device controllers
- Processor write data in memory and output controller do the actually output while processor continues executing program
- Read data by input controller in memory (while processor executes) and interrupt processor to read data when in memory



Output controller and program execution proceed in parallel

Interrupts vs Polling

Poll:

you need to spin wait until something happens or timeout

Interrupt:

processor gets interrupted when device interrupt bit enabled.

Έχουμε καλύψει ένα υποσύνολο της αρχιτεκτονικής MIPS

Το παράρτημα Β εξηγά το πως να καθορίσετε το περιεχόμενο των διαφόρων segments (.text, .data) εμβέλεια (.globl), alignment, strings, macros και σταθερές