Οργάνωση Υπολογιστών

5 “συστατικά” στοιχεία

-Επεξεργαστής:

datapath (δίοδος δεδομένων) (1) και control (2)

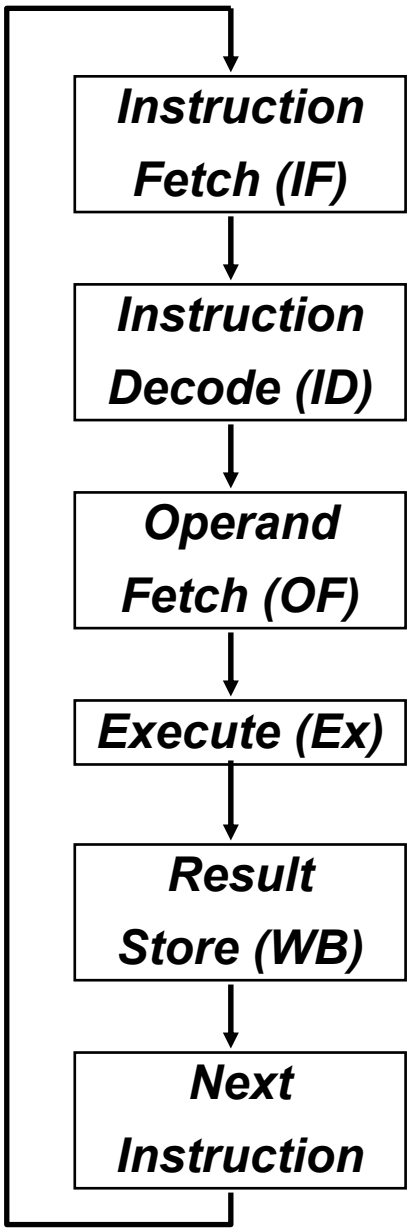
-Μνήμη (3)

-Συσκευές Εισόδου (4), Εξόδου (5) (*Μεγάλη ‘ποικιλία’ !!)*

Συσκευές γρήγορες π.χ. κάρτες γραφικών, αργές π.χ. πληκτρολόγιο.

Για το I/O έχει γίνει η λιγότερη έρευνα …..(I/O busses , I/O switched fabrics …)

Ιεραρχία Μνήμης: καταχωρητές, κρυφή μνήμη (L1), κρυφή μνήμη (L2), κύρια Μνήμη- ΠΟΛΥ ΣΗΜΑΝΤΙΚΟ ΣΤΟΙΧΕΙΟ!



Αρχιτεκτονικές Συνόλου Εντολών

Instruction Set Architectures

Αριθμός εντολών

Μορφή Εντολών:

μεταβλητό ή σταθερό μέγεθος bytes για κάθε εντολή; (8086 1-17 bytes, MIPS 4 bytes)

Πώς γίνεται η αποκωδικοποίηση (ID);

Που βρίσκονται τα ορίσματα (operands) και το αποτέλεσμα:

Μνήμη-καταχωρητές, *πόσα ορίσματα*, τι μεγέθους;

Ποια είναι στη μνήμη και ποια όχι;

Πόσοι κύκλοι για κάθε εντολή;

Κατηγορίες Αρχιτεκτονικών Συνόλου Εντολών

(ISA Classes)

1. Αρχιτεκτονικές Συσσωρευτή (accumulator architectures)

   (μας θυμίζει κάτι?)

2. Αρχιτεκτονικές επεκταμένου συσσωρευτή ή καταχωρητών ειδικού σκοπού (extended accumulator ή special purpose register)

3. Αρχιτεκτονικές Καταχωρητών Γενικού Σκοπού

   3α. register-memory

   3b. register-register (RISC)

Αρχιτεκτονικές Συσσωρευτή (1)

1η γενιά υπολογιστών: h/w ακριβό, μεγάλο μέγεθος καταχωρητή.

Ένας καταχωρητής για όλες τις αριθμητικές εντολές (*συσσώρευε* όλες τις λειτουργίες → *Συσσωρευτής (Accum)*

Σύνηθες: 1ο όρισμα είναι ο Accum, 2ο η μνήμη, αποτέλεσμα στον Accum π.χ. add 200

Παράδειγμα: A = B + C

Accum = Memory(AddressB);	**Load AddressB**
Accum = Accum + Memory(AddressC);	**Add AddressC**
Memory(AddressA) = Accum;	**Store AddressA**

Όλες οι μεταβλητές αποθηκεύονται στη μνήμη. Δεν υπάρχουν βοηθητικοί καταχωρητές

Αρχιτεκτονικές Συσσωρευτή (2)

Κατά:

Χρειάζονται πολλές εντολές για ένα πρόγραμμα

Κάθε φορά πήγαινε-φέρε από τη μνήμη
(? Κακό είναι αυτό)
Bottleneck ο Accum!

Υπέρ:

Εύκολοι compilers, κατανοητός προγραμματισμός,
εύκολη σχεδίαση h/w

Λύση; Πρόσθεση καταχωρητών για συγκεκριμένες λειτουργίες
(ISAs καταχωρητών ειδικού σκοπού)

Αρχιτεκτονικές Επεκταμένου Συσσωρευτή

Καταχωρητές ειδικού σκοπού π.χ. δεικτοδότηση, αριθμητικές πράξεις

Υπάρχουν εντολές που τα ορίσματα είναι όλα σε καταχωρητές

Κατά βάση (π.χ. σε αριθμητικές εντολές) το ένα όρισμα στη μνήμη.

Αρχιτεκτονικές Καταχωρητών γενικού σκοπού

Register-memory

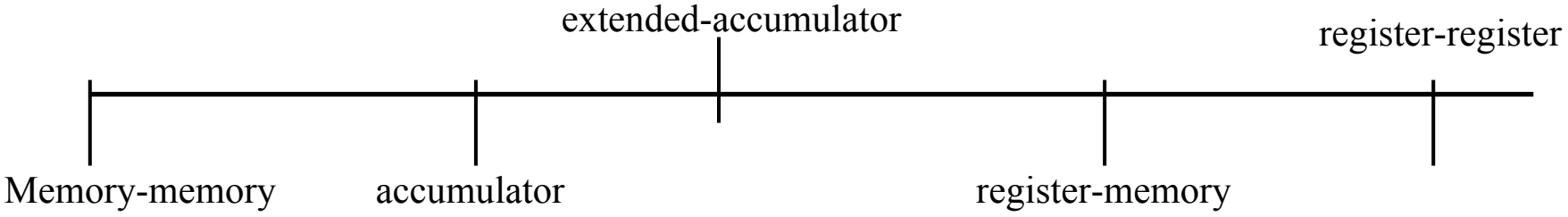
Αφήνουν το ένα όρισμα να είναι στη μνήμη (πχ. 80386)

Register-register (load store) (1980+)

A=B+C

Load R1, B
Add R1, C
Store A, R1

Load R1, B
Load R2, C
Add R3, R1, R2
Store A, R3



Αρχιτεκτονική Στοίβας

Καθόλου registers! Stack model ~ 1960!!!

Στοίβα που μεταφέρονται τα ορίσματα που αρχικά βρίσκονται στη μνήμη. Καθώς βγαίνουν γίνονται οι πράξεις και το αποτέλεσμα ξαναμπαίνει στη στοίβα.

Θυμάστε τα HP calculators με reverse polish notation

A=B+C
push Address C
push AddressB
add
pop AddressA

Εντολές μεταβλητού μήκους:

1-17 bytes 80x86

1-54 bytes VAX, IBM

Γιατί??

Instruction Memory ακριβή, οικονομία χώρου!!!!

Εμείς στο μάθημα: register-register ISA! (load- store)

1. Οι καταχωρητές είναι γρηγορότεροι από τη μνήμη
2. Μειώνεται η κίνηση με μνήμη
3. Δυνατότητα να υποστηριχθεί σταθερό μήκος εντολών
4. (τα ορίσματα είναι καταχωρητές, άρα ό αριθμός τους (πχ. 1-32 καταχωρητές) όχι δ/νσεις μνήμης

Compilers πιο δύσκολοι!!!

Βασικές Αρχές Σχεδίασης (patterson-hennessy COD2e)

1. Η ομοιομορφία των λειτουργιών συμβάλλει στην απλότητα του υλικού (Simplicity favors Regularity)
2. Όσο μικρότερο τόσο ταχύτερο! (smaller is faster)
3. Η καλή σχεδίαση απαιτεί σημαντικούς συμβιβασμούς (Good design demands good compromises)

Γενικότητες? Θα τα δούμε στη συνέχεια.....

MIPS σύνολο εντολών:

Λέξεις των 32 bit (μνήμη οργανωμένη σε bytes, ακολουθεί το μοντέλο big Endian)

32 καταχωρητές γενικού σκοπού - REGISTER FILE

Θα μιλήσουμε για: εντολές αποθήκευσης στη μνήμη (lw, sw)

Αριθμητικές εντολές (add, sub κλπ)

Εντολές διακλάδωσης (branch instructions)

Δεν αφήνουμε τις εντολές να έχουν μεταβλητό πλήθος ορισμάτων- π.χ. add a,b,c **πάντα**: a=b+c

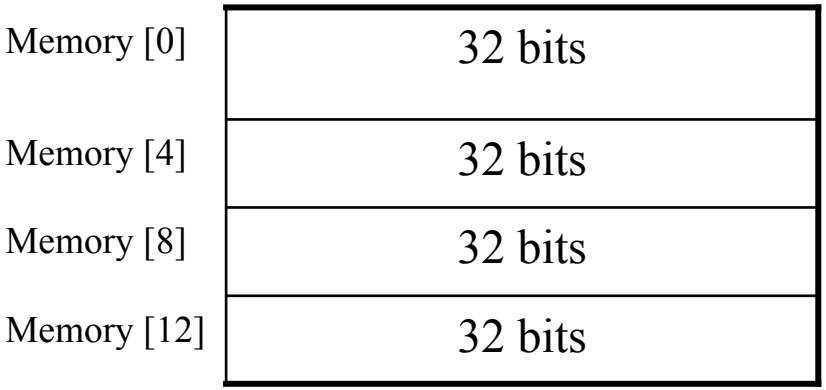
Θυμηθείτε την 1η αρχή: *Η ομοιομορφία των λειτουργιών συμβάλλει στην απλότητα του h/w*

Αφού οι καταχωρητές είναι τόσο....«γρήγοροι» γιατί να μην μεγαλώσουμε το μέγεθος του register file?

2η αρχή: Όσο μικρότερο τόσο ταχύτερο!

Αν το register file πολύ μεγάλο, πιο πολύπλοκη η αποκωδικοποίηση, πιο μεγάλος ο κύκλος ρολογιού (φάση ID) άρα.....υπάρχει tradeoff

Μνήμη οργανωμένη σε bytes: (Κάθε byte και ξεχωριστή δνση)
2^{30} λέξεις μνήμης των 32 bit (4 bytes) κάθε μια



$s0, $s1,… καταχωρητές (μεταβλητές συνήθως)
$t0, $t1,… καταχωρητές (προσωρινές τιμές)
$zero ειδικός καταχωρητής περιέχει το 0

Κανόνες Ονοματοδοσίας και Χρήση των MIPS Registers

- Εκτός από το συνήθη συμβολισμό των καταχωρητών με $ ακολουθούμενο από τον αριθμό του καταχωρητή, μπορούν επίσης να παρασταθούν και ως εξής :

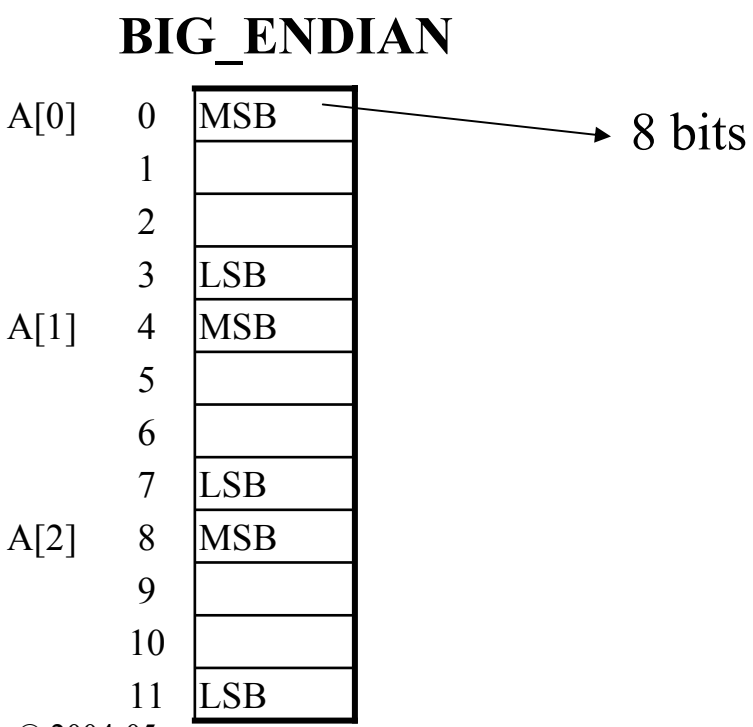
Αρ. Καταχωρητή	Όνομα	Χρήση	Preserved on call?
0	$zero	Constant value 0	n.a.
1	$at	Reserved for assembler	όχι
2-3	$v0-$v1	Values for result and expression evaluation	όχι
4-7	$a0-$a3	Arguments	ναι
8-15	$t0-$t7	Temporaries	όχι
16-23	$s0-$s7	Saved	ναι
24-25	$t8-$t9	More temporaries	όχι
26-27	$k0-$k1	Reserved for operating system	ναι
28	$gp	Global pointer	ναι
29	$sp	Stack pointer	ναι
30	$fp	Frame pointer	ναι
31	$ra	Return address	ναι

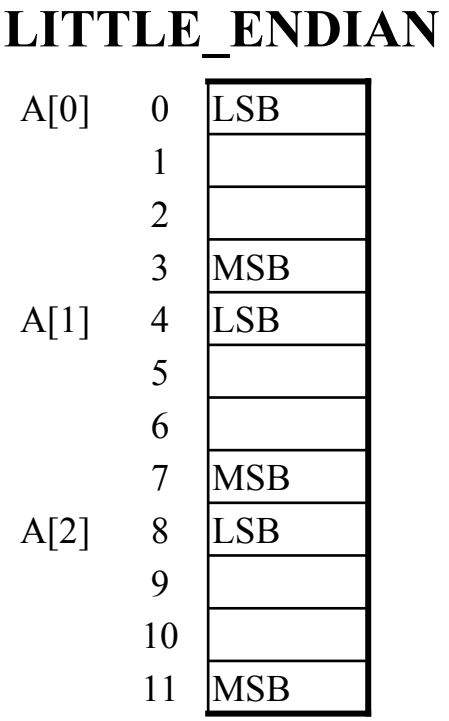
Big Endian vs Little Endian

Big Endian: Η δνση του πιο σημαντικού byte (MSB) είναι και **δνση** της λέξης

Little Endian: Η δνση του λιγότερο σημαντικού byte (LSB) είναι και **δνση** της λέξης

Η λέξη αποθηκεύεται πάντα σε συνεχόμενες θέσεις:
δνση, δνση+1,..,δνση+3





MIPS ISA (βασικές εντολές)

Αριθμητικές εντολές add, sub: πάντα τρία ορίσματα - **ποτέ** δνση μνήμης!

```
add $s1, $s2, $s3     #  $s1 = $s2+$s3
sub $s1, $s2, $s3     #  $s1 = $s2-$s3
```

Εντολές μεταφοράς δεδομένων (load-store):

Εδώ έχουμε αναφορά στη μνήμη (πόσοι τρόποι? Θα το δούμε στη συνέχεια)

```
lw $s1, 100($s2) # $s1 = Memory(100+$s2) (load word)
sw $s1, 100($s2) # Memory(100+$s2) = $s1(store word)
```

MIPS Arithmetic Instructions
Παραδείγματα

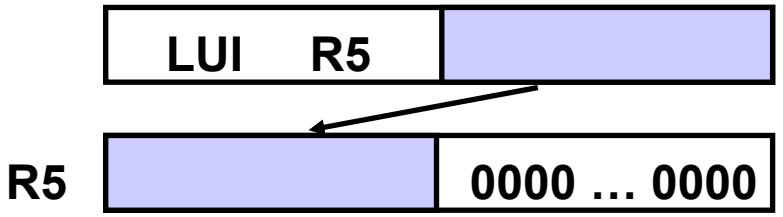
Instruction	*Παράδειγμα*	*Έννοια*	*Σχόλια*
add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exception possible
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; no exceptions
subtract unsigned	subu $1,$2,$3	$1 = $2 – $3	3 operands; no exceptions
add imm. unsign.	addiu $1,$2,100	$1 = $2 + 100	+ constant; no exceptions
multiply	mult $2,$3	Hi, Lo = $2 x $3	64-bit signed product
multiply unsigned	multu$2,$3	Hi, Lo = $2 x $3	64-bit unsigned product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
divide unsigned	divu $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Unsigned quotient & remainder
Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
Move from Lo	mflo $1	$1 = Lo	Used to get copy of Lo

MIPS Logic/Shift Instructions
Παραδείγματα

Instruction	*Παράδειγμα*	*Έννοια*	*Σχόλια*
and	and $1,$2,$3	$1 = $2 & $3	3 reg. operands; Logical AND
or	or $1,$2,$3	$1 = $2 | $3	3 reg. operands; Logical OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	3 reg. operands; Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	3 reg. operands; Logical NOR
and immediate	andi $1,$2,10	$1 = $2 & 10	Logical AND reg, constant
or immediate	ori $1,$2,10	$1 = $2 | 10	Logical OR reg, constant
xor immediate	xori $1, $2,10	$1 = ~$2 &~10	Logical XOR reg, constant
shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
shift right arithm.	sra $1,$2,10	$1 = $2 >> 10	Shift right (sign extend)
shift left logical	sllv $1,$2,$3	$1 = $2 << $3	Shift left by variable
shift right logical	srlv $1,$2, $3	$1 = $2 >> $3	Shift right by variable
shift right arithm.	srav $1,$2, $3	$1 = $2 >> $3	Shift right arith. by variable

MIPS data transfer instructions
Παραδείγματα

Instruction	*Σχόλια*
sw 500($4), $3	Store word
sh 502($2), $3	Store half
sb 41($3), $2	Store byte
lw $1, 30($2)	Load word
lh $1, 40($3)	Load halfword
lhu $1, 40($3)	Load halfword unsigned
lb $1, 40($3)	Load byte
lbu $1, 40($3)	Load byte unsigned
lui $1, 40	Load Upper Immediate (16 bits shifted left by 16)



MIPS Branch, Compare, Jump
Παραδείγματα

Instruction	*Παράδειγμα*	*Έννοια*
branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+100 *Equal test; PC relative branch*
branch on not eq.	bne $1,$2,100	if ($1!= $2) go to PC+4+100 *Not equal test; PC relative branch*
set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0 *Compare less than; 2's comp.*
set less than imm.	slti $1,$2,100	if ($2 < 100) $1=1; else $1=0 *Compare < constant; 2's comp.*
set less than uns.	sltu $1,$2,$3	if ($2 < $3) $1=1; else $1=0 *Compare less than; natural numbers*
set l. t. imm. uns.	sltiu $1,$2,100	if ($2 < 100) $1=1; else $1=0 *Compare < constant; natural numbers*
jump	j 10000	go to 10000 *Jump to target address*
jump register	jr $31	go to $31 *For switch, procedure return*
jump and link	jal 10000	$31 = PC + 4; go to 10000 *For procedure call*

Παράδειγμα: υποθέτουμε ότι η μεταβλητή h είναι αποθηκευμένη στον καταχωρητή $s2 και η αρχή του πίνακα A (base address) βρίσκεται στο καταχωρητή $s3

Ο A είναι πίνακας 100 λέξεων των 32 bit εκάστη.

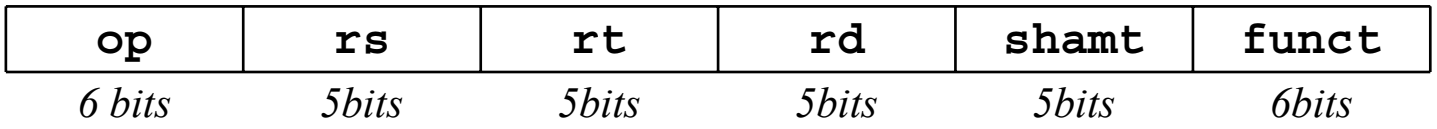
Πώς γράφεται το: A[12]= h + A[8] ;

```
lw $t0, 32($s3)     # temporary reg. $t0 gets A[8]
add $t0, $s2, $t0   # temporary reg. $t0 gets h + A[8]
sw $t0, 48($s3)     # stores h+A[8] back into A[12]
```

Μορφή Εντολής - Instruction Format

Θυμηθείτε το 1ο κανόνα: *Η ομοιομορφία των λειτουργιών συμβάλλει στην απλότητα του υλικού*

R-Type
(register type)



`Op`: opcode
`rs`,`rt`: register source operands
`Rd`: register destination operand
`Shamt`: shift amount
`Funct`: op specific (function code)

```
add $rd, $rs, $rt
```

MIPS R-Type (ALU)

R-Type: Όλες οι εντολές της ALU που χρησιμοποιούν 3 καταχωρητές

- Παραδείγματα :
  - `add $1,$2,$3`
  - `sub $1,$2,$3`
  - `and $1,$2,$3`
  - `or $1,$2,$3`

Destination register in rd

Operand register in rt

Operand register in rs

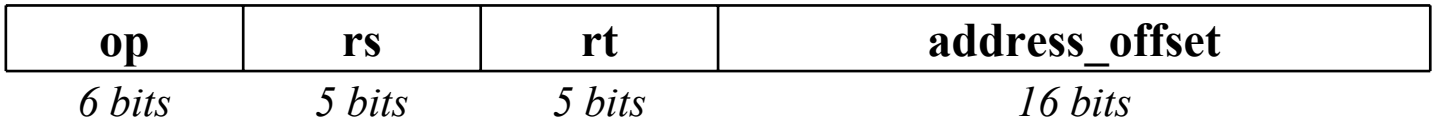
Ερώτηση: Μας αρκεί το R-Type?

Τι γίνεται με εντολές που θέλουν ορίσματα διευθύνσεις ή σταθερές? Θυμηθείτε, θέλουμε σταθερό μέγεθος κάθε εντολής (32 bit)

Απάντηση: Μάλλον όχι

Άρα: *Η καλή σχεδίαση απαιτεί σημαντικούς συμβιβασμούς (3η αρχή)*

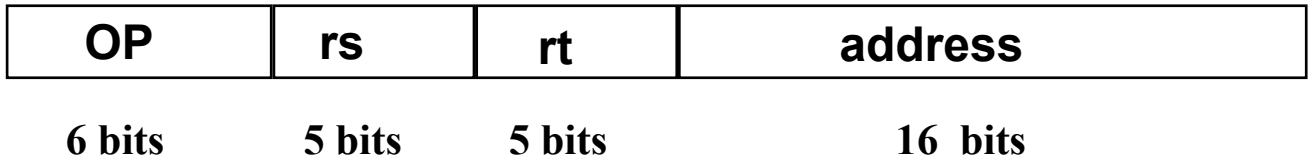
I-Type:




```
lw $rt, address_offset($rs)
```

Τα 3 πρώτα πεδία (op,rs, rt) έχουν το ίδιο όνομα και μέγεθος όπως και πριν

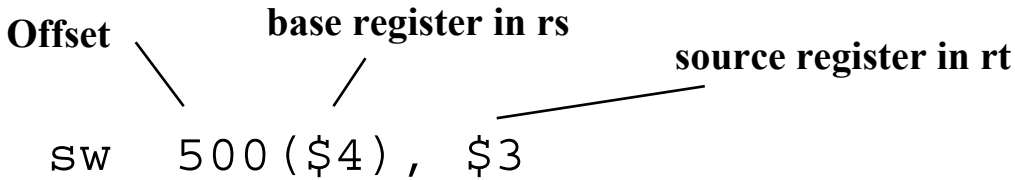
MIPS I-Type : Load/Store



– *address: 16-bit memory address offset in bytes added to base register.*

- Παραδείγματα :

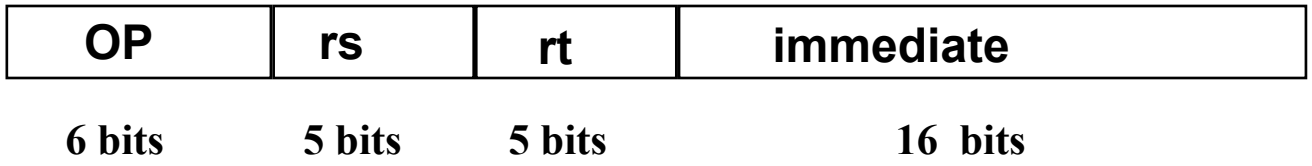
– Store word:



– Load word:

MIPS ALU I-Type

Οι I-Type εντολές της ALU χρησιμοποιούν 2 καταχωρητές και μία σταθερή τιμή
I-Type είναι και οι εντολές Loads/stores, conditional branches.

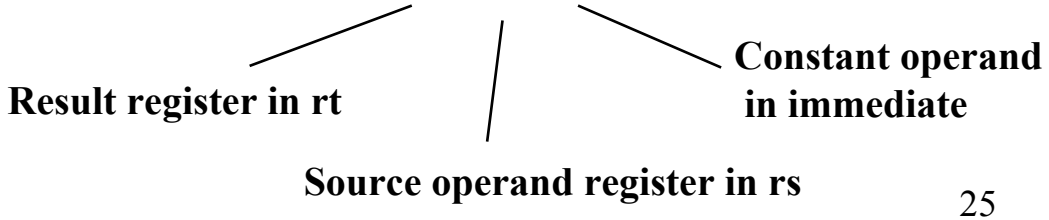


– *immediate: Constant second operand for ALU instruction.*

- Παραδείγματα :

  – add immediate: `addi $1,$2,100`

  – and immediate `andi $1,$2,10`



Εντολές διακλάδωσης-branching instructions

branch if equal

```
beq $s3, 4s4, L1 # goto L1 if $s3 equals $s4
```

branch if !equal

```
bne $s3, 4s4, L1 # goto L1 if $s3 not equals $s4
```

unconditional Jump

```
jr $t1 # goto $t1
```

..... είναι I –Type εντολές

```
slt $t0, $s3, $s4 #set $t0 to 1 if  $s3 is less
                  than $s4;else set $t0 to 0
```

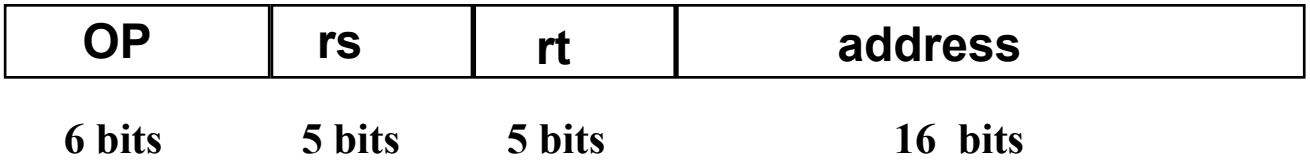
Όμως:

```
j L1 # goto L1
```

Πόσο μεγάλο είναι το μήκος του address L1;

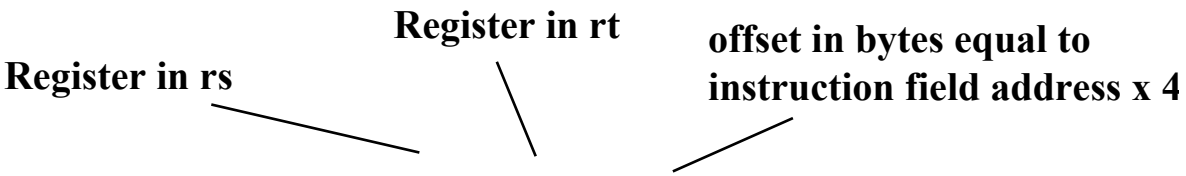
Πόσο «μεγάλο» μπορεί να είναι το άλμα;

MIPS Branch I-Type



– *address: 16-bit memory address branch target offset in words added to PC to form branch address.*

- Παραδείγματα :



  - Branch on equal `beq $1,$2,100`
  - Branch on not equal `bne $1,$2,100`

MIPS J-Type

J-Type: jump j, jump and link jal

– *jump target: jump memory address in words.*

- Παραδείγματα :

Jump memory address in bytes equal to instruction field jump target x 4

– Branch on equal `j 10000`

– Branch on not equal `jal 10000`

Απ' ευθείας διευθυνσιοδότηση- Σταθερές

Οι πιο πολλές αριθμητικές εκφράσεις σε προγράμματα, περιέχουν σταθερές: π.χ. index++

Στον κώδικα του gcc: 52% εκφράσεων έχουν constants

Στον κώδικα του spice: 69% των εκφράσεων!

Τι κάνουμε με τις σταθερές (και αν είναι > 16bit;)

Θέλουμε: `$s3=$s3+2`

```
lw $t0, addr_of_constant_2($zero)
add $s3,$s3,$t0
```

Αλλιώς: `addi $s3,$s3,2` (add immediate)

Όμοια: `slti $t0,$s2, 10 # $t0=1 if $s2<10`

Τρόποι Διευθυνσιοδότησης στον MIPS:

1. *Register* Addressing
2. *Base* or *Displacement* Addressing
3. *Immediate* Addressing
4. *PC-relative* addressing (address is the sum of the PC and a constant in the instruction)
5. *Pseudodirect* addressing (the jump address is the 26 bits of the instruction, concatenated with the upper bits of the PC)

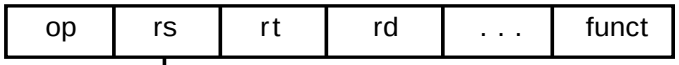
1. Immediate addressing


```
addi $rt,$rs,immediate
```

π.χ.
```
lui  $t0, 255
slti $t0, $s1, 10
```

I-Type

2. Register addressing

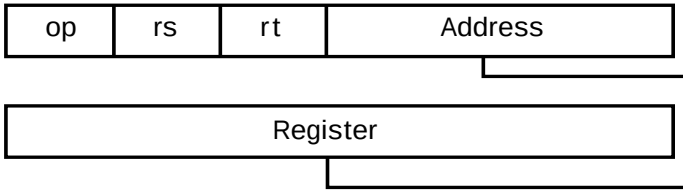


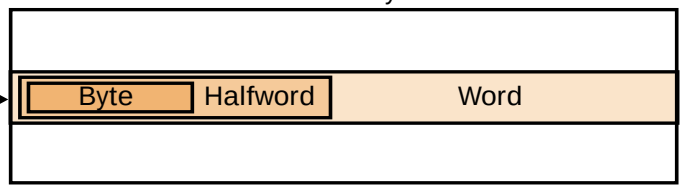
R-Type

```
add $rd,$rs,$rt
```

3. Base addressing

π.χ.
```
add $t0, $s1,$s2
```



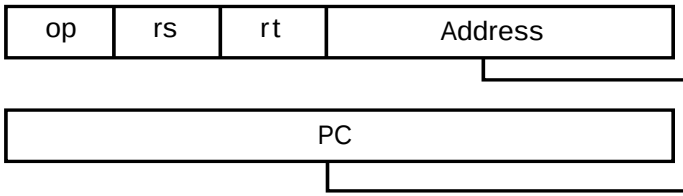


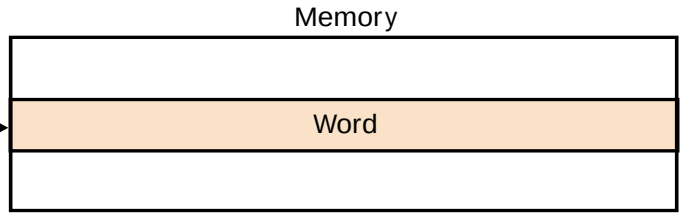
I-Type

```
lw $rt, address($rs)
```

π.χ.
```
lw $t1,100($s2)
```

4. PC-relative addressing



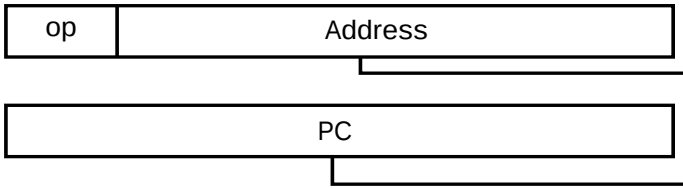


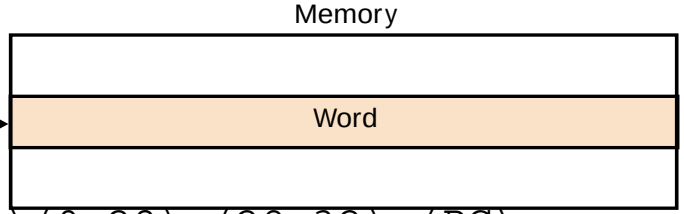
I-Type

```
bne $rs, $rt, address
```

π.χ.
```
bne $s0,$s1,L2
```

5. Pseudodirect addressing





J-Type

```
j address # goto (4 x address)(0:28)-(29:32)  (PC)
```

Addressing Modes : Παραδείγματα

Addr. mode	*Παράδειγμα*	*Έννοια*	*χρήση*
Register	add r4,r3	Regs[r4]← Regs[r4]+ Regs[r3]	a value is in register
Immediate	add r4,#3	Regs[r4]← Regs[r4]+3	for constants
Displacement	add r4,100(r1)	Regs[r4]← Regs[r4]+Mem[100+ Regs[r1]]	local variables
Reg. indirect	add r4,(r1)	Regs[r4]← Regs[r4]+Mem[Regs[r1]]	accessing using a pointer or comp. address
Indexed	add r4,(r1+r2)	Regs[r4]← Regs[r4]+Mem[Regs[r1]+ Regs[r2]]	array addressing (base +offset)
Direct	add r4,(1001)	Regs[r4]← Regs[r4]+Mem[1001]	addr. static data
Mem. Indirect	add r4,@(r3)	Regs[r4]← Regs[r4]+Mem[Mem[Regs[r3]]]	if R3 keeps the address of a pointer p, this yields *p
Autoincrement	add r4,(r3)+	Regs[r4]← Regs[r4]+Mem[Regs[r3]] Regs[r3]← Regs[r3]+d	stepping through arrays within a loop; d defines size of an element
Autodecrement	add r4,-(r3)	Regs[r3]← Regs[r3]-d Regs[r4]← Regs[r4]+Mem[Regs[r3]]	similar as previous
Scaled	add r4,100(r2)[r3]	Regs[r4]← Regs[r4]+ Mem[100+Regs[r2]+Regs[r3]*d]	to index arrays

Επεξεργαστής	Αριθμός καταχωρητών γενικού σκοπού	Αρχιτεκτονική	Έτος
EDSAC	1	accumulator	1949
IBM 701	1	accumulator	1953
CDC 6600	8	load-store	1963
IBM 360	16	register-memory	1964
DEC PDP-8	1	accumulator	1965
DEC PDP-11	8	Register-memory	1970
Intel 8008	1	accumulator	1972
Motorola 6800	2	accumulator	1974
DEC VAX	16	register-memory, memory-memory	1977
Intel 8086	8	extended accumulator	1978
Motorola 68000	16	register-memory	1980
Intel 80386	8	register-memory	1985
MIPS	32	load-store	1985
HP PA-RISC	32	load-store	1986
SPARC	32	load-store	1987
PowerPC	32	load-store	1992
DEC Alpha	32	load-store	1992