Οργάνωση Υπολογιστών

5 “συστατικά” στοιχεία

-Επεξεργαστής:

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

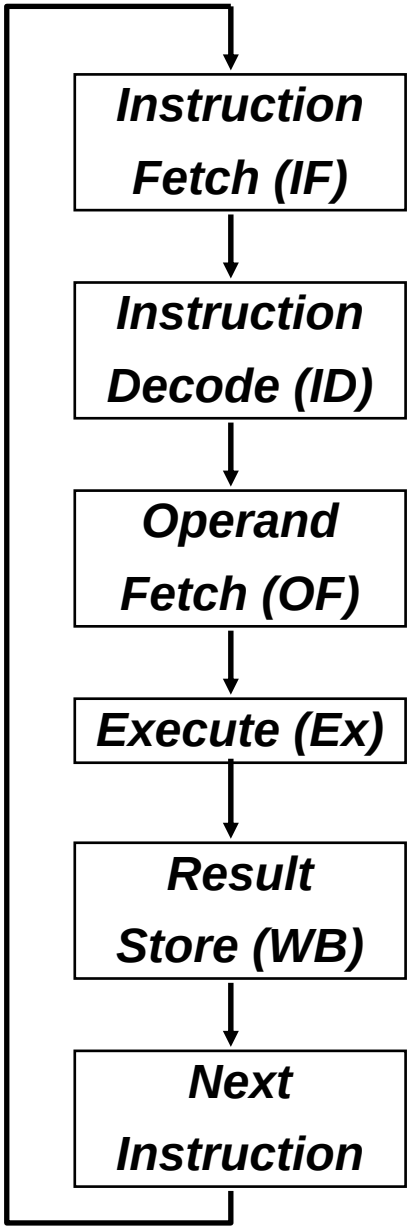
-Μνήμη (3)

-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

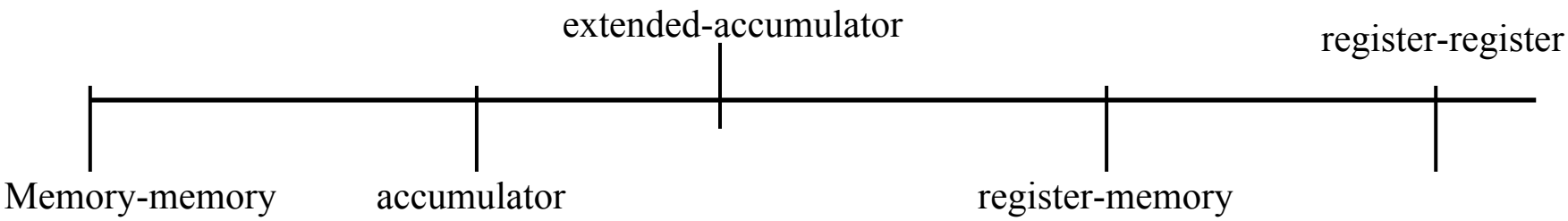
Αφήνουν το ένα όρισμα να είναι στη μνήμη (πχ. 80386)

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*

Σύνολο Εντολών
Instruction Set

- Λέξεις της γλώσσας του υπολογιστή – εντολές
- Γλώσσες των υπολογιστών – όμοιες
- Στο μάθημα, σύνολο εντολών MIPS

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

Memory [0]	32 bits
Memory [4]	32 bits
Memory [8]	32 bits
Memory [12]	32 bits

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

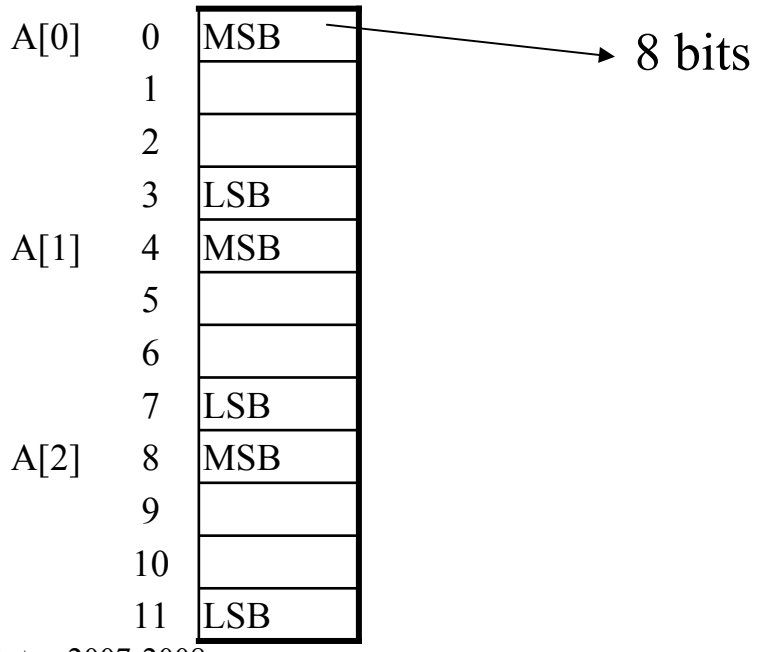
Big Endian vs Little Endian

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

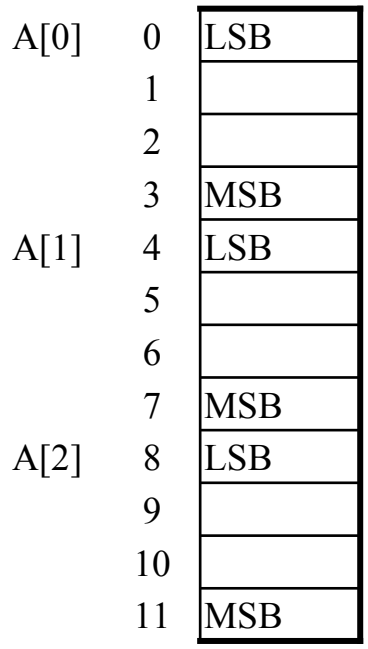
Little Endian: Η δνση του λιγότερο σημαντικού byte (LSB) είναι και **δνση** της λέξης

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

BIG_ENDIAN



LITTLE_ENDIAN



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)
```

Λειτουργίες υλικού υπολογιστών

Αριθμητικές Πράξεις

- `add a, b, c #` $a \leftarrow b + c$

- $a = b + c + d + e;$
  - `add a, b, c`
  - `add a, a, d`
  - `add a, a, e`

  3 εντολές για το άθροισμα 4 μεταβλητών

Λειτουργίες υλικού υπολογιστών

<u>Παράδειγμα</u>:

Κώδικας σε C

a = b + c;

d = a – e;

Μετάφραση σε κώδικα MIPS

```
add a, b, c
sub d, a, e
```

Λειτουργίες υλικού υπολογιστών

<u>Παράδειγμα</u>:

f = (g + h) – (i + j);

Τι παράγει ο compiler?

```
add t0, g, h # προσωρινή μεταβλητή t0
add t1, i, j # προσωρινή μεταβλητή t1
sub f, t0, t1 # το f περιέχει το t0 – t1
```

Λειτουργίες υλικού υπολογιστών

Κατηγορία	Εντολή	Παράδειγμα	Σημασία	Σχόλια
Αριθμητικές Πράξεις	add	`add a, b, c`	a = b + c	Πάντα 3 τελεστέοι
	subtract	`sub a, b, c`	a = b - c	Πάντα 3 τελεστέοι

Τελεστέοι Υλικού Υπολογιστών

f = (g + h) – (i + j);

(f,g,h,i,j) ανατίθενται σε ($s0,$s1,$s2,$s3,$s4)

$t0, $t1 προσωρινοί καταχωρητές

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Τελεστέοι Υλικού Υπολογιστών

Γλώσσες προγραμματισμού έχουν:

- απλές μεταβλητές
- σύνθετες δομές (π.χ. arrays, structs)

Πώς τις αναπαριστά ένας υπολογιστής;

<u>Πάντα στη μνήμη</u>

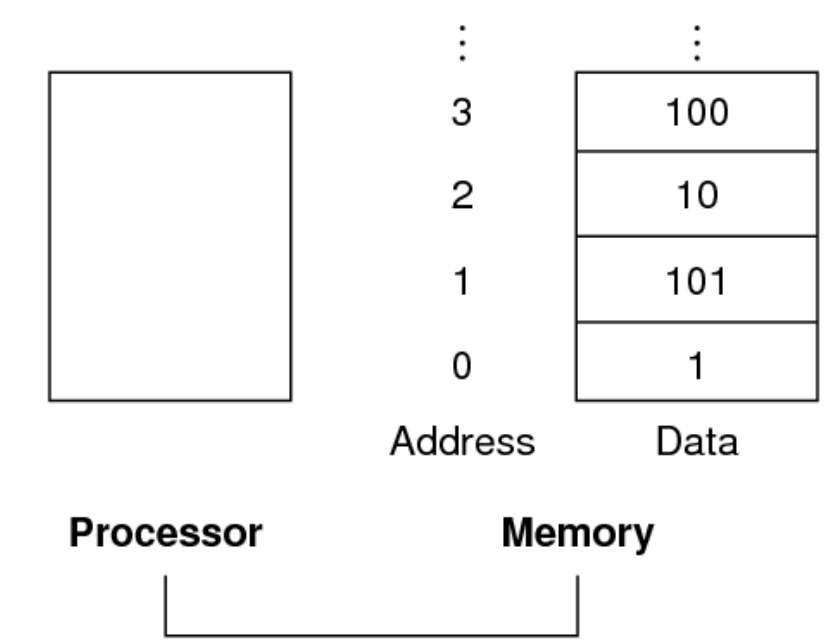
Εντολές μεταφοράς δεδομένων

Τελεστέοι Υλικού Υπολογιστών

Εντολές μεταφοράς δεδομένων

Π.χ. Διεύθυνση του 3 στοιχείου είναι 2

Τιμή στοιχείου Memory[2] είναι 10



Τελεστέοι Υλικού Υπολογιστών

Εντολή μεταφοράς δεδομένων από τη μνήμη

load καταχωρητής, σταθερά(καταχωρητής)

π.χ.

```
lw $t1, 4($s2)
```

φορτώνουμε στον $t1 την τιμή M[$s2+4]

Τελεστέοι Υλικού Υπολογιστών

Παράδειγμα:

Α πίνακας 100 λέξεων (4 bytes η λέξη)

g, h ανατίθενται σε $s1, $s2

αρχική διεύθυνση του Α στον $s3

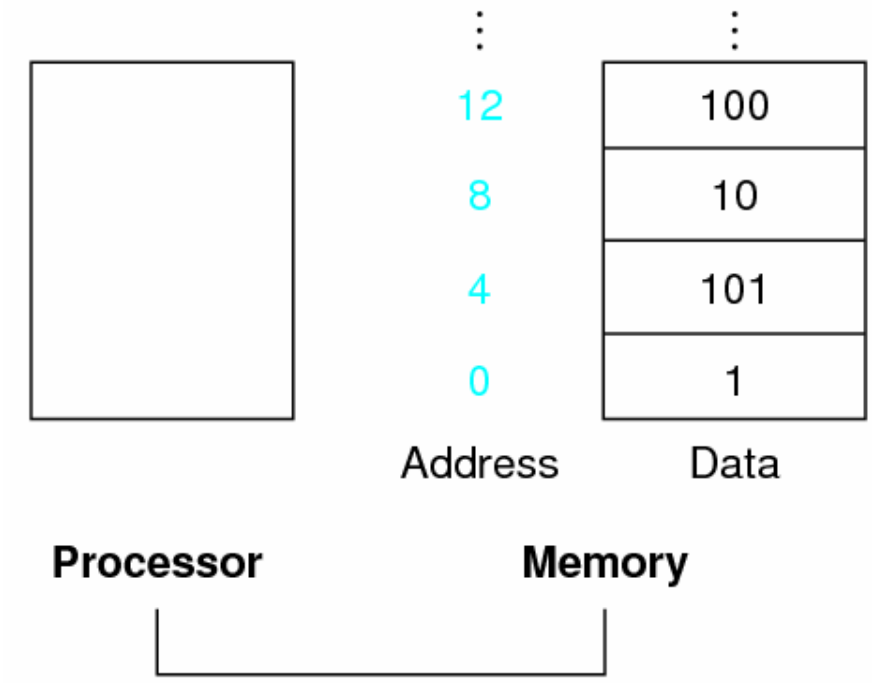
Μεταγλωττίστε g = h + A[8];

offset **base register**

```
lw $t0, 32($s3)
add $s1, $s2, $t0
```

Τελεστέοι Υλικού Υπολογιστών

- Μνήμη είναι byte addressable
- Δύο διαδοχικές λέξεις διαφέρουν κατά 4
- alignment restriction (ευθυγράμμιση)
  - λέξεις ξεκινάνε πάντα σε διεύθυνση πολ/σιο του 4



Τελεστέοι Υλικού Υπολογιστών

Σταθερές:

Πρόσθεση της τιμής 4 στον $s3

```
lw $t0, AddrConstant4($s1)
add $s3, $s3, $t0
```

ή

```
addi $s3, $s3, 4
```

Make common case FAST

Τελεστέοι MIPS

32 καταχωρητές $s0, $s1, ...

2^{30} θέσεις λέξεων στη μνήμη

2^{32} θέσεις byte στη μνήμη

Συμβολική γλώσσα MIPS
Παραδείγματα

```
add $s1, $s2, $s3
sub $s1, $s2, $s3
addi $s1, $s2, 100
lw $s1, 100($s2)
sw $s1,100($s2)
```

Τελεστέοι Υλικού Υπολογιστών

Πόσο γρήγορα πληθαίνουν οι καταχωρητές σε έναν επεξεργαστή;

1. Πολύ γρήγορα (Νόμος Moore, 2ος αριθμός τρανζίστορ/18 μήνες)
2. Πολύ αργά (portable binary code)

<u>Π.χ. Θέλουμε τα προγράμματά μας να τρέχουν και στον Pentium III και στον Pentium IV</u>

Αναπαράσταση Εντολών στον Υπολογιστή

Δυαδικά ψηφία, Δυαδικό σύστημα (binary)

Υλικό υπολογιστών, υψηλή-χαμηλή τάση, κλπ.

<u>Καταχωρητές</u>

$s0, ..., $s7 αντιστοιχίζονται στους 16 ως 23

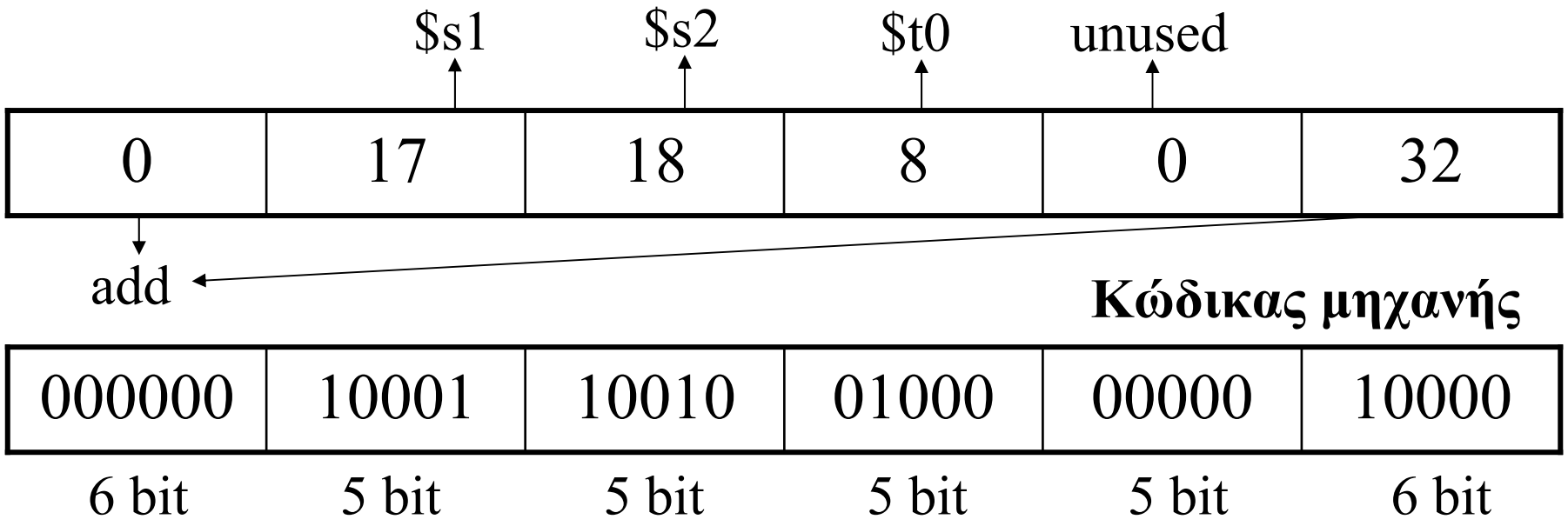
$t0, ..., $t7 αντιστοιχίζονται στους 8 ως 15

Αναπαράσταση Εντολών στον Υπολογιστή

Συμβολική αναπαράσταση:

`add $t0, $s1, $s2` → **Assembly**

Πώς την καταλαβαίνει ο MIPS?



Μορφή Εντολής - Instruction Format

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

R-Type
(register type)

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

`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

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

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

Destination register in rd

Operand register in rt

Operand register in rs

Αναπαράσταση Εντολών στον Υπολογιστή

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

Τι γίνεται με τη load?

Πώς χωράνε οι τελεστές της στα παραπάνω πεδία? Π.χ. η σταθερά της lw.

```
lw $t1, 8000($s3)
```

σε ποιο πεδίο χωράει;

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:

op	rs	rt	address_offset
6 bits	*5 bits*	*5 bits*	*16 bits*

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

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

Αναπαράσταση Εντολών στον Υπολογιστή

Παράδειγμα:

```
lw $t0, 32($s3)
```

Καταχωρητές (σκονάκι ☺)
$s0, ..., $s7 αντιστοιχίζονται στους 16 ως 23
$t0, ..., $t7 αντιστοιχίζονται στους 8 ως 15

I-format

op	rs	rt	σταθερά ή διεύθυνση
6 bit	5 bit	5 bit	16 bit
xxxxxx	19	8	32

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

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

MIPS I-Type : Load/Store

OP	rs	rt	address
6 bits	5 bits	5 bits	16 bits

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

- Παραδείγματα :

– Store word: `sw 500($4), $3`

Offset — base register in rs — source register in rt

– Load word: `lw $1, 30($2)`

Destination register in rt — Offset — base register in rs

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.

OP	rs	rt	immediate
6 bits	**5 bits**	**5 bits**	**16 bits**

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

- Παραδείγματα :

  – add immediate: `addi $1,$2,100`

  – and immediate `andi $1,$2,10`

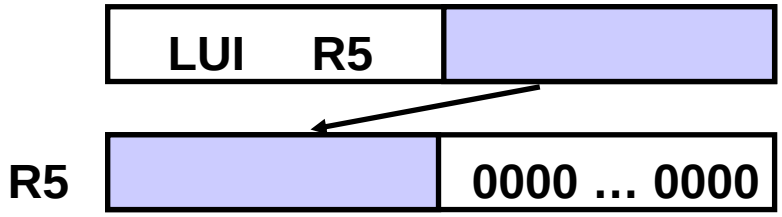
Result register in rt

Source operand register in rs

Constant operand in immediate

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)



Αναπαράσταση Εντολών στον Υπολογιστή

εντολή	μορφή	op	rs	rt	rd	shamt	funct	address
`add`	R	0	reg	reg	reg	0	32_{ten}	δ.ε.
`sub`	R	0	reg	reg	reg	0	34_{ten}	δ.ε.
`addi`	I	8_{ten}	reg	reg	δ.ε.	δ.ε.	δ.ε.	σταθ.
`lw`	I	35_{ten}	reg	reg	δ.ε.	δ.ε.	δ.ε.	διευθ.
`sw`	I	43_{ten}	reg	reg	δ.ε.	δ.ε.	δ.ε.	διευθ.

Αναπαράσταση Εντολών στον Υπολογιστή

Παράδειγμα:

$t1 περιέχει base address πίνακα A (32 bit/στοιχείο A[i])

$s2 αντιστοιχίζεται στη μεταβλητή h

Μεταγλωττίστε το A[300] = h + A[300];

```
lw $t0, 1200($t1)
add $t0, $s2, $t0
sw $t0, 1200($t1)
```

Αναπαράσταση Εντολών στον Υπολογιστή

Παράδειγμα (συνέχεια):

```
lw $t0, 1200($t1)
add $t0, $s2, $t0
sw $t0, 1200($t1)
```

Κώδικας Μηχανής?

op	rs	rt	rd	shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Αναπαράσταση Εντολών στον Υπολογιστή

Παράδειγμα (συνέχεια):

op	rs	rt	rd	shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

op	rs	rt	rd	shamt	funct
10**0**011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	8	0	32
10**1**011	01001	01000	0000 0100 1011 0000		

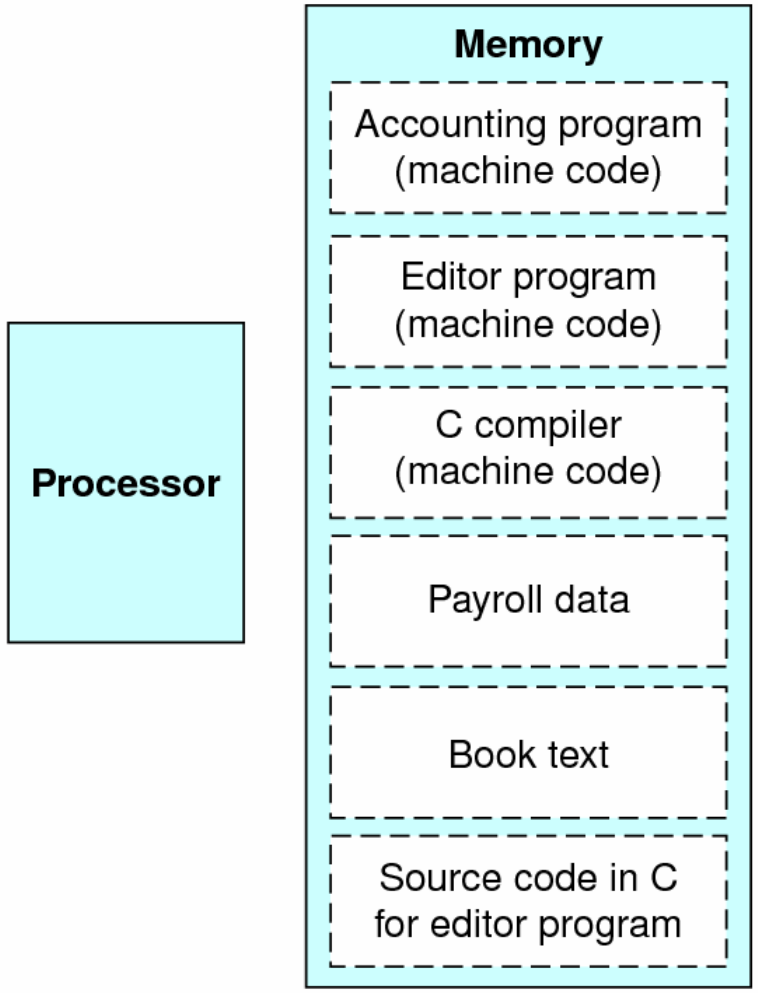
Αναπαράσταση Εντολών στον Υπολογιστή

Έννοια αποθηκευμένου προγράμματος

Ο υπολογιστής κάνει πολλές εργασίες φορτώνοντας δεδομένα στο μνήμη.

Δεδομένα και εντολές είναι στοιχεία στη μνήμη.

Π.χ. compilers μεταφράζουν στοιχεία σε κάποια άλλα στοιχεία.



Λογικές Λειτουργίες (Πράξεις)

Λογικές Λειτουργίες	Τελεστές C	Εντολές MIPS
Shift left	<<	`Sll (shift left logical)`
Shift right	>>	`Srl (shift right logical)`
AND	&	`and, andi`
OR	|	`or, ori`
NOT	~	`nor`

Λογικές Λειτουργίες (Πράξεις)

<u>SHIFT</u>

$s0: 0000 0000 0000 0000 0000 0000 0000 1001 = 9_{ten} ☺

```
sll $t2, $s0, 4
```

Κάνουμε shift αριστερά το περιεχόμενο του $s0 κατά 4 θέσεις

0000 0000 0000 0000 0000 0000 1001 0000 = 144_{ten}

και τοποθετούμε το αποτέλεσμα στον $t2.

!!Το περιεχόμενο του $s0 μένει αμετάβλητο!!

Λογικές Λειτουργίες (Πράξεις)

SHIFT

```
sll $t2, $s0, 4
```

Καταχωρητές (σκονάκι ☺)
$s0, ..., $s7 αντιστοιχίζονται στους 16 ως 23
$t0, ..., $t7 αντιστοιχίζονται στους 8 ως 15

6 bit	5 bit	5 bit	5 bit	5 bit	6 bit
op	rs	rt	rd	shamt	funct
0	0	16	10	4	0
000000	00000	10000	01010	00100	000000

sll: opcode=0, funct=0

Λογικές Λειτουργίες (Πράξεις)

AND, OR

$t2: 0000 0000 0000 0000 0000 1101 0000 0000

$t1: 0000 0000 0000 0000 0011 1100 0000 0000

```
and $t0, $t1, $t2          # Μάσκα
```

$t0: 0000 0000 0000 0000 0000 1100 0000 0000

```
or $t0, $t1, $t2
```

$t0: 0000 0000 0000 0000 0011 1101 0000 0000

Λογικές Λειτουργίες (Πράξεις)

NOT, NOR

$t1: 0000 0000 0000 0000 0011 1100 0000 0000

$t3: 0000 0000 0000 0000 0000 0000 0000 0000

`not $t0, $t1` **δεν υπάρχει** γιατί θέλουμε πάντα 2 καταχωρητές source. Άρα χρησιμοποιούμε τη **nor:**

A NOR 0 = NOT (A OR 0) = NOT A

`nor $t0, $t1, $t3`

$t0: 1111 1111 1111 1111 1100 0011 1111 1111

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

Εντολές Λήψης Αποφάσεων

<u>beq, bne</u>

```
beq reg1, reg2, L1  #branch if equal
```

Αν οι καταχωρητές reg1 και reg2 **είναι ίσοι**, πήγαινε στην ετικέτα L1

```
bne reg1, reg2, L1  #branch if not equal
```

Αν οι καταχωρητές reg1 και reg2 **δεν είναι ίσοι**, πήγαινε στην ετικέτα L1

Εντολές Λήψης Αποφάσεων

Παράδειγμα:

if(i == j) f = g + h; else f = g – h;

με f, g, h, i, j αντιστοιχούνται σε $s0, ..., $s4

version 1

```
     bne $s3, $s4, Else
     add $s0, $s1, $s2
     j Exit
Else:sub $s0, $s1, $s2
Exit:
```

version 2

```
     beq $s3, $s4, Then
     sub $s0, $s1, $s2
     j Exit
Then: add $s0, $s1, $s2
Exit:
```

Εντολές Λήψης Αποφάσεων

Βρόχοι (Loops)

while (save[i] == k) i += 1;

με i = $s3, k = $s5, save base addr = $s6

```
Loop:     sll  $t1, $s3, 2 #πολ/ζω i επί 4
          add  $t1, $t1, $s6
          lw   $t0, 0($t1)
          bne  $t0, $s5, Exit
          addi $s3, $s3, 1
          j    Loop
Exit:
```

Εντολές Λήψης Αποφάσεων

<u>Συγκρίσεις</u>

```
slt $t0, $s3, $s4   # set on less than
```

Ο καταχωρητής $t0 τίθεται με 1 αν η τιμή στον $s3 είναι μικρότερη από την τιμή στο $s4.

Σταθερές ως τελεστέοι είναι δημοφιλείς στις συγκρίσεις

```
slti $t0, $s2, 10   # set on less than
                    # immediate
```

Ο καταχωρητής $t0 τίθεται με 1 αν η τιμή στον $s2 είναι μικρότερη από την τιμή 10.

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*

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

OP	rs	rt	address
6 bits	5 bits	5 bits	16 bits

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

- Παραδείγματα :

Final offset is calculated in bytes, equals to {instruction field address} x 4, e.g. new PC = PC + 400

Register in rt

Register in rs

- 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

OP	jump target
6 bits	**26 bits**

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

final jump memory address in bytes is calculated from {jump target} x 4

- Παραδείγματα :
  - 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:

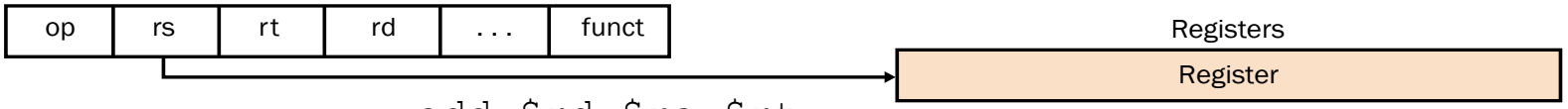
1. *Register* Addressing
2. *Base* or *Displacement* Addressing
3. *Immediate* Addressing
4. *PC-relative* addressing (address is the sum of the PC and a constant in the instruction)
5. *Pseudodirect* addressing (the jump address is the 26 bits of the instruction, concatenated with the upper bits of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

1. Immediate addressing

```
      addi $rt,$rs,immediate
π.χ.  lui  $t0, 255
      slti $t0, $s1, 10
```

I-Type

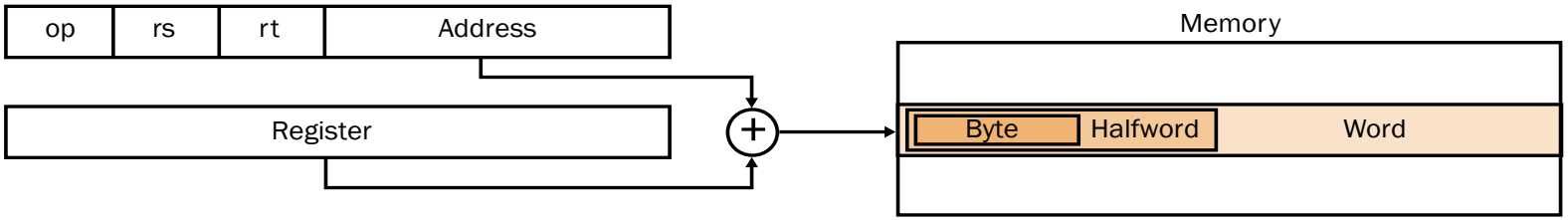
2. Register addressing



R-Type

```
add $rd,$rs,$rt
π.χ. add $t0,$s1,$s2
```

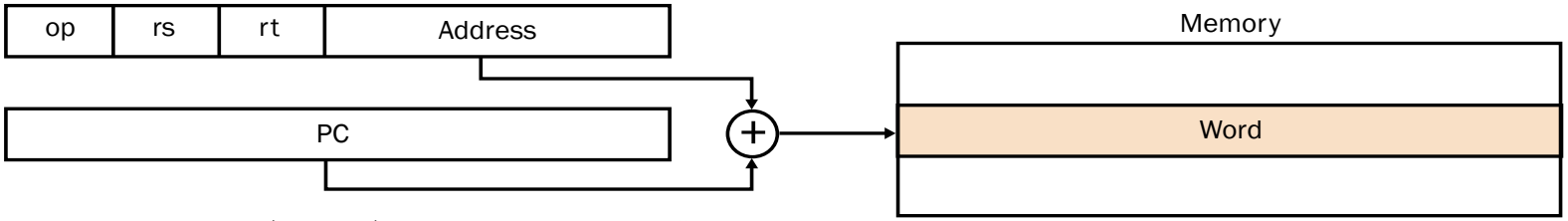
3. Base addressing



I-Type

```
lw $rt,address($rs)
π.χ.  lw $t1,100($s2)
```

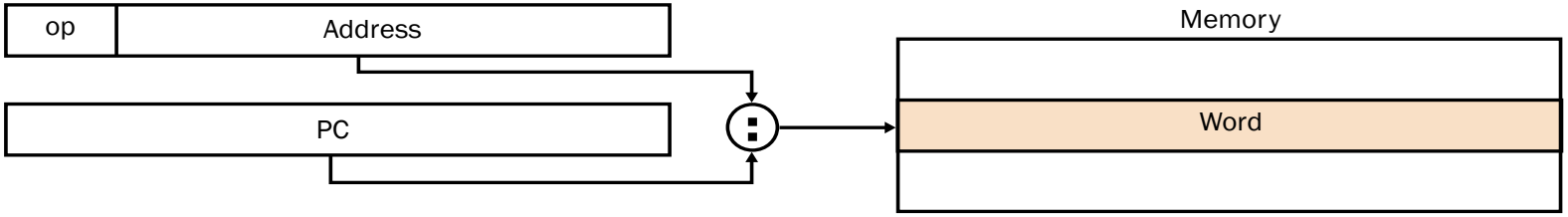
4. PC-relative addressing



I-Type

```
bne $rs,$rt,address # goto {4 x address (0:15)}
π.χ. bne $s0,$s1,L2
```

5. Pseudodirect addressing



J-Type

```
j address # goto {4 x address(0:25)}(0:27)-(28:31)(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