MIPS lab

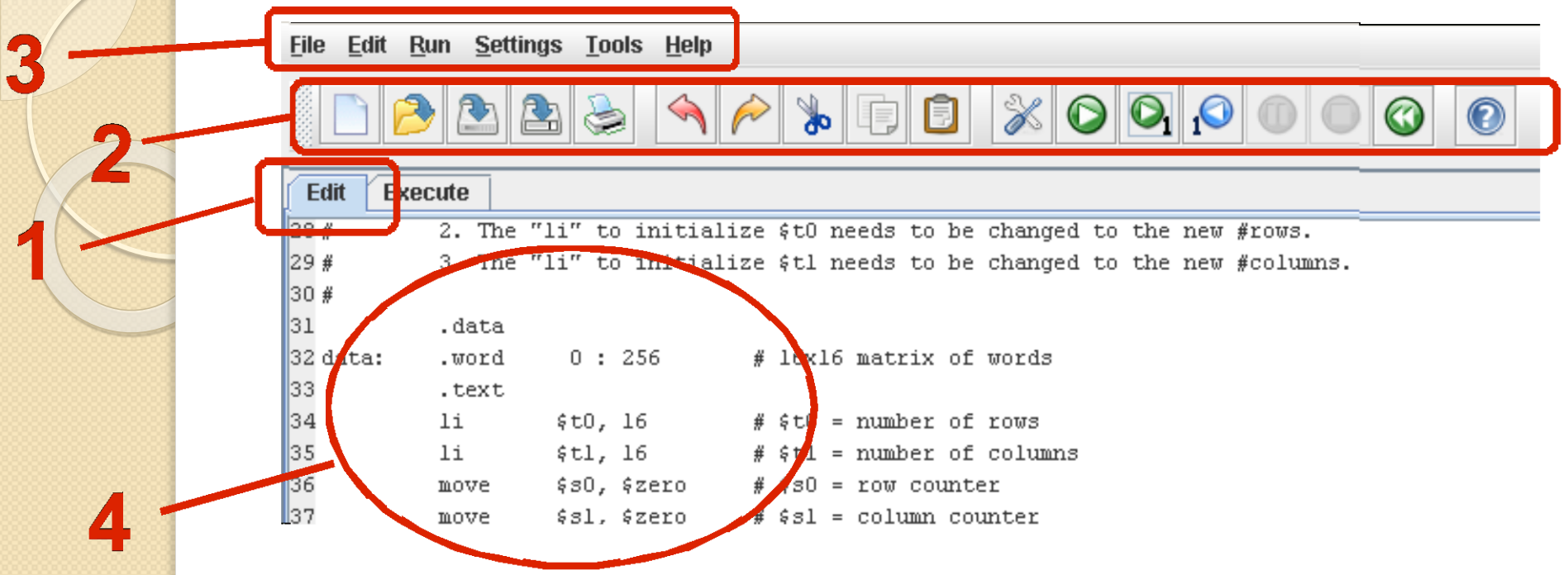
CPU Architecture

Agenda

- Mips Assembler And Runtime Simulator (MARS)
- Building an example code on MARS
- Simulating MIPS CPU using MARS and Modelsim

Mips Assembler And Runtime Simulator (MARS)

- Intro to MARS
- Loading data and
- MIPS Architecture



1. Edit display is indicated by highlighted tab.
2, 3. Typical edit and execute operations are available through icons and menus, dimmed-out when unavailable or not applicable.
4. WYSIWYG editor for MIPS assembly language code.

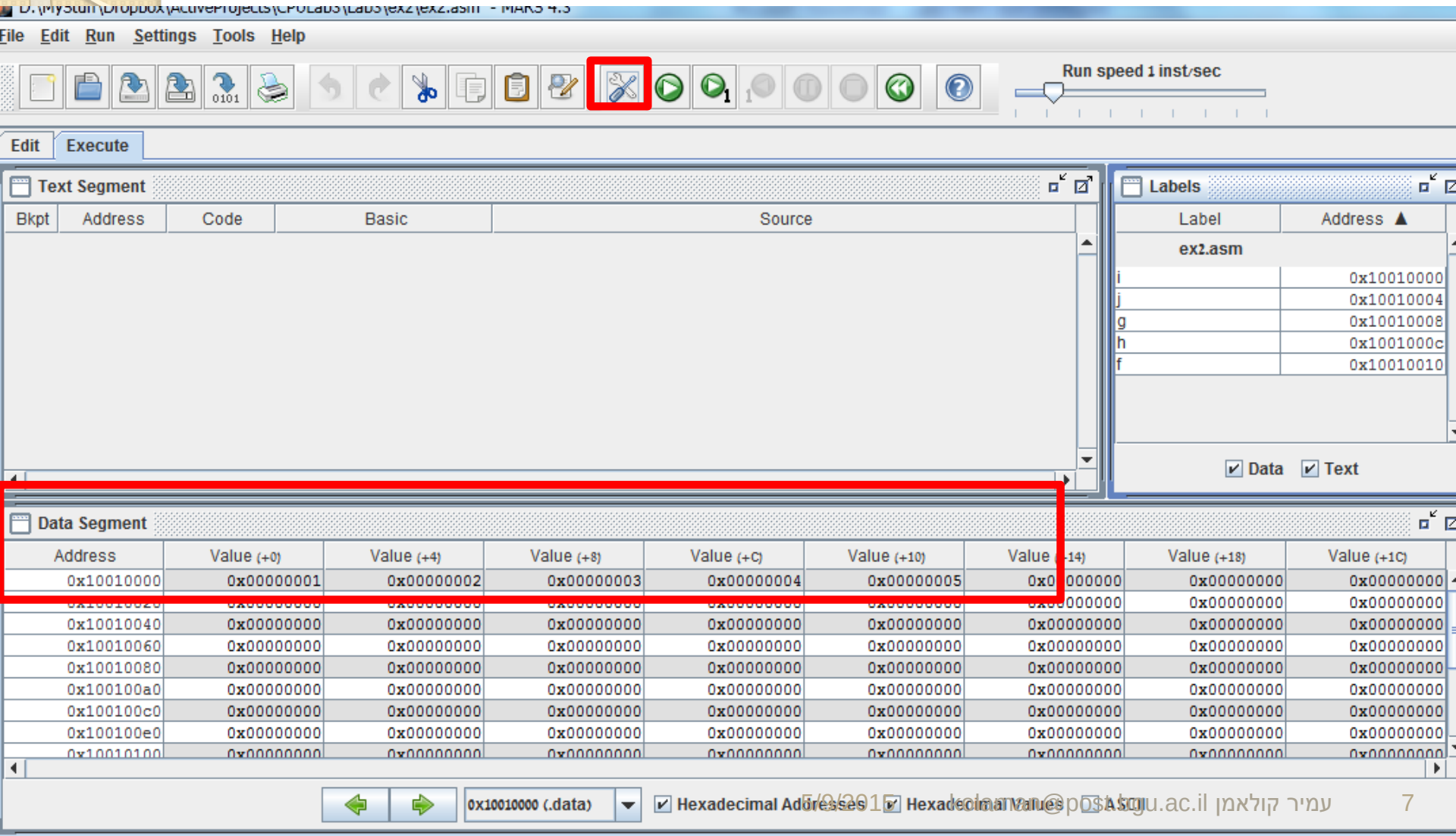
Data segment and comments

- data part starts with reserved word
- **.data**
- each data array has a label, a type reset value and range
- A: .word 0:10
- comments are marked with **#**

example

```
.data          #data part starts here
i: .word 1     #initial value of i is 1
.text                  #code part starts here
.
.
```

After compiling, the address of labels and memory content can be viewed



Instruction overview

Field Size	6-bits	5-bits	5-bits	5-bits	5-bits	6-bits
R- Format	Opcode	Rs	Rt	Rd	Shift	Function
I - Format	Opcode	Rs	Rt	Address/immediate value		
J - Format	Opcode	Branch target address				

- example code

A = B + C.

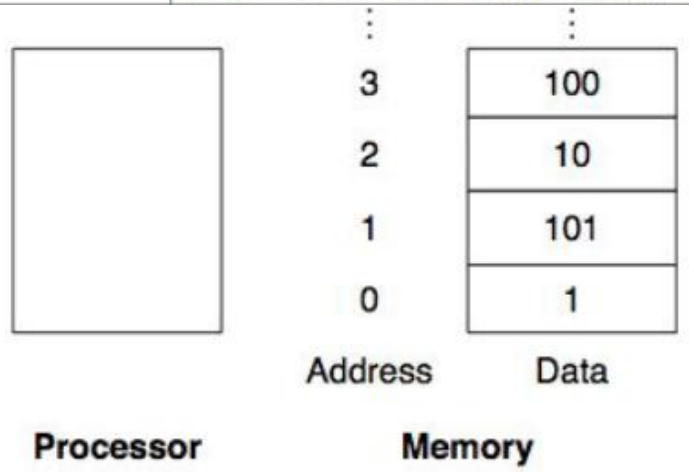
```
LW  $2, B          ;Register 2 = value of memory at address B
LW  $3, C          ;Register 3 = value of memory at address C
ADD $4, $2, $3     ;Register 4 = B + C
SW  $4, A          ;Value of memory at address A = Register 4
```

Instruction overview

Mnemonic	Format	Opcode Field	Function Field	Instruction
Add	R	0	32	Add
Addi	I	8	-	Add Immediate
Addu	R	0	33	Add Unsigned
Sub	R	0	34	Subtract
Subu	R	0	35	Subtract Unsigned
And	R	0	36	Bitwise And
Or	R	0	37	Bitwise OR
Sll	R	0	0	Shift Left Logical
Srl	R	0	2	Shift Right Logical
Slt	R	0	42	Set if Less Than
Lui	I	15	-	Load Upper Immediate
Lw	I	35	-	Load Word
Sw	I	43	-	Store Word
Beq	I	4	-	Branch on Equal
Bne	I	5	-	Branch on Not Equal
J	J	2	-	Jump
Jal	J	3	-	Jump and Link (used for Call)
Jr	R	0	8	Jump Register (used for Return)

Data transfer instructions examples

Category	Instruction	Example	Meaning	Comments
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits



Code segment and comments

- code part starts with
- **.text**
- comments are marked with **#**

Register types

- $0 or $zero - always contains 0.
- $t0 - $t9 - use for temporary storage of data
- $s0 - $s7 - use to hold address locations in memory
- $a0 - $a3 - use as arguments to system calls
- $v0 and $v1 - use as arguments to system calls

Instruction overview

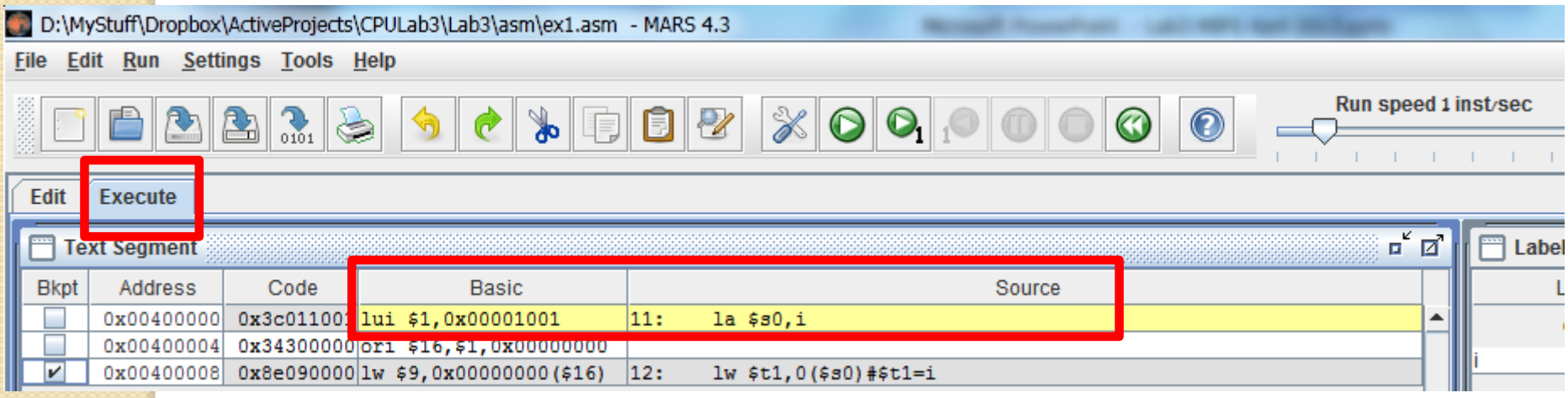
- Registers are used as their purpose and when the user wants their information to be saved when a procedure is called

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	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

Source and basic columns in the execute tab

- Source is the instruction we wrote
- Basic is the real instruction that is encoded to language

Basic	Source
lw $9,0x00000000($0)	6: lw $t1,0 #$t1=i



Addressing types examples

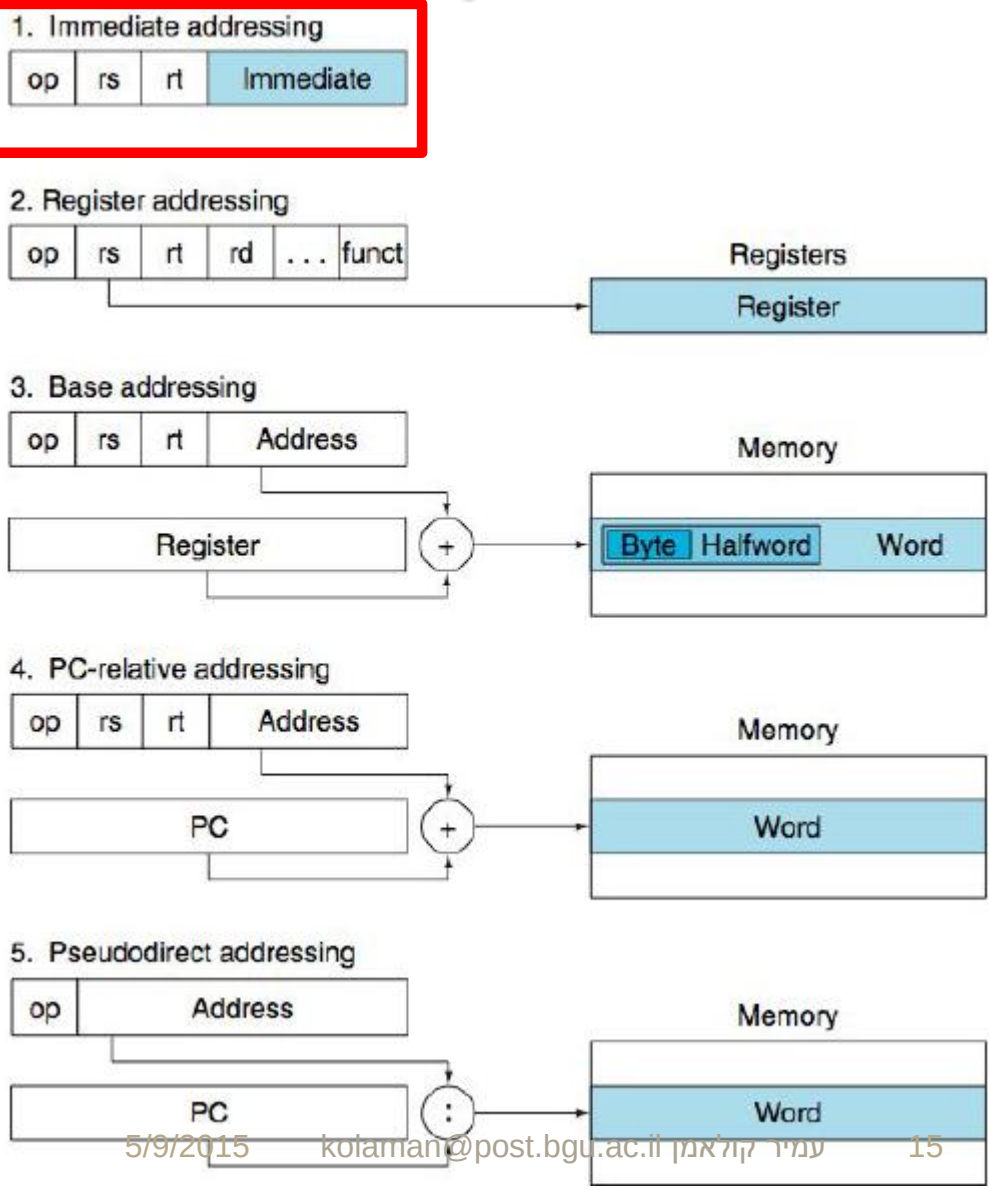
lw $t2,0

add $t5,$t3,$t4

lw $t2,0($s0)

beq $t0,$zero,ELSE

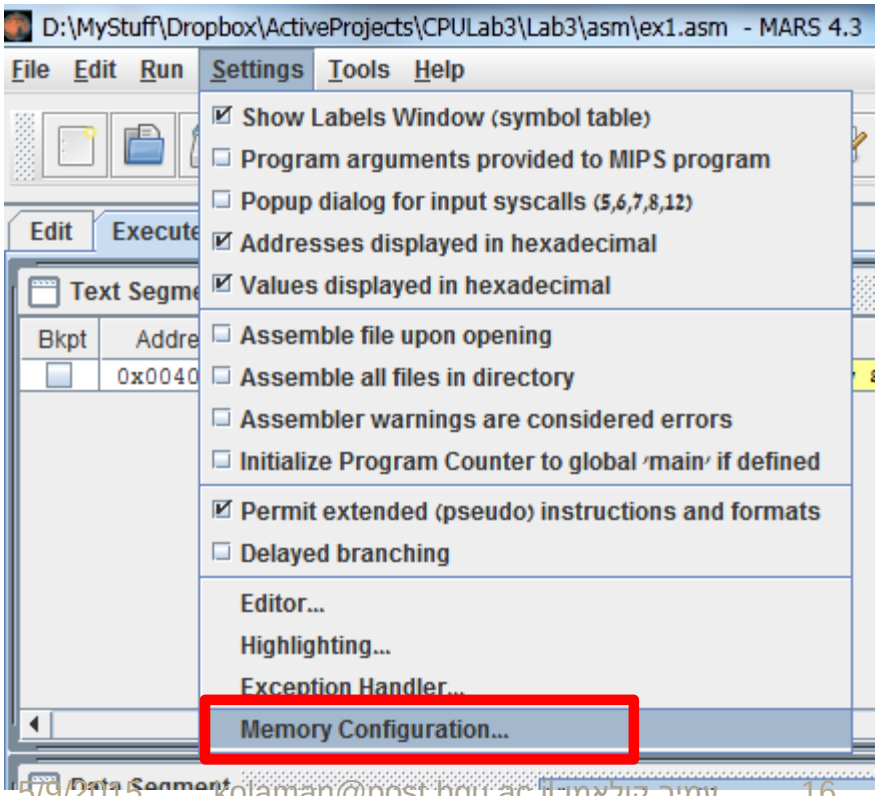
j END



text/data segment location in memory

- If we use immediate addressing we assume that our data starts at 0
- To setup the address of text and data go to
- Settings→ Memory Configuration

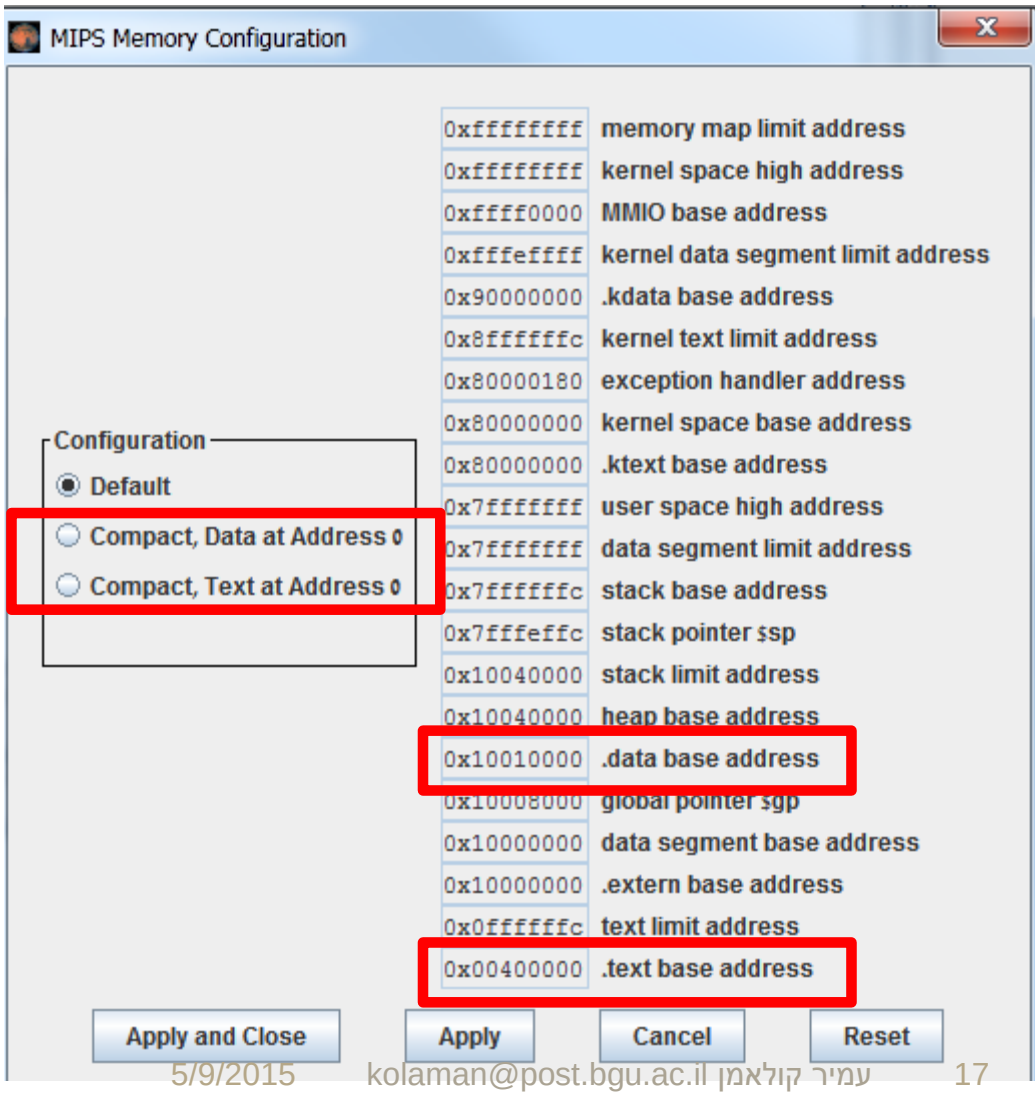
```
lw $t1,0        #$t1=i
```



text/data segment location in memory

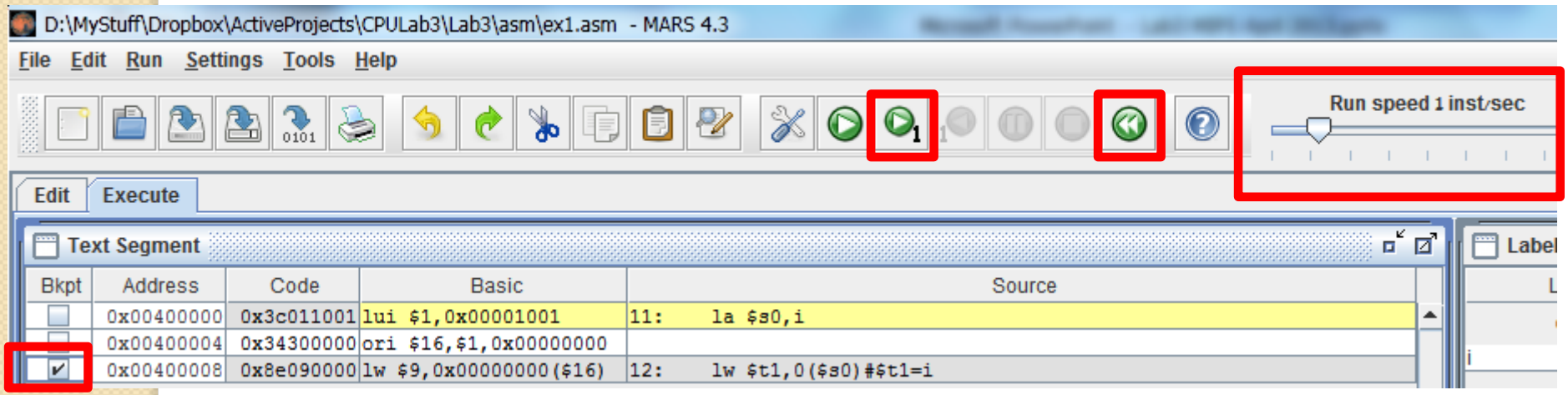
Basic	Source
lw $9,0x00000000($0)	6: lw $t1,0 #$t1=i

- Default base address of .data is 0x10010000
- Default base address of .text is 0x00400000
- We may change the data/text to start at address 0



Debugging

- We can set the run speed to a costume speed and view the register/memory change as the occur.
- We can Reset and run by step.
- We can also set a break point .



Addressing types

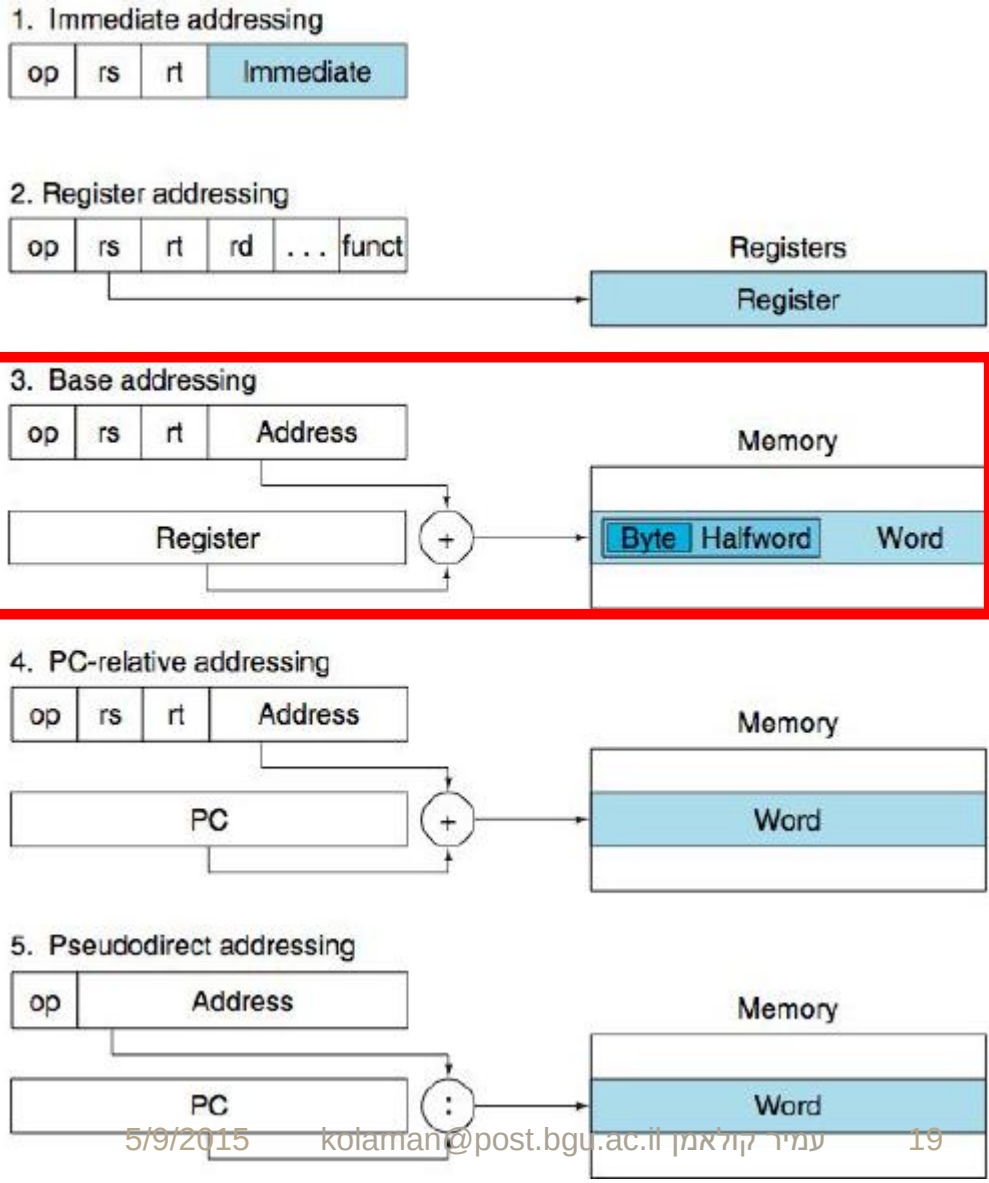
lw $t2,0

add $t5,$t3,$t4

lw $t2,0($s0)

beq $t0,$zero,ELSE

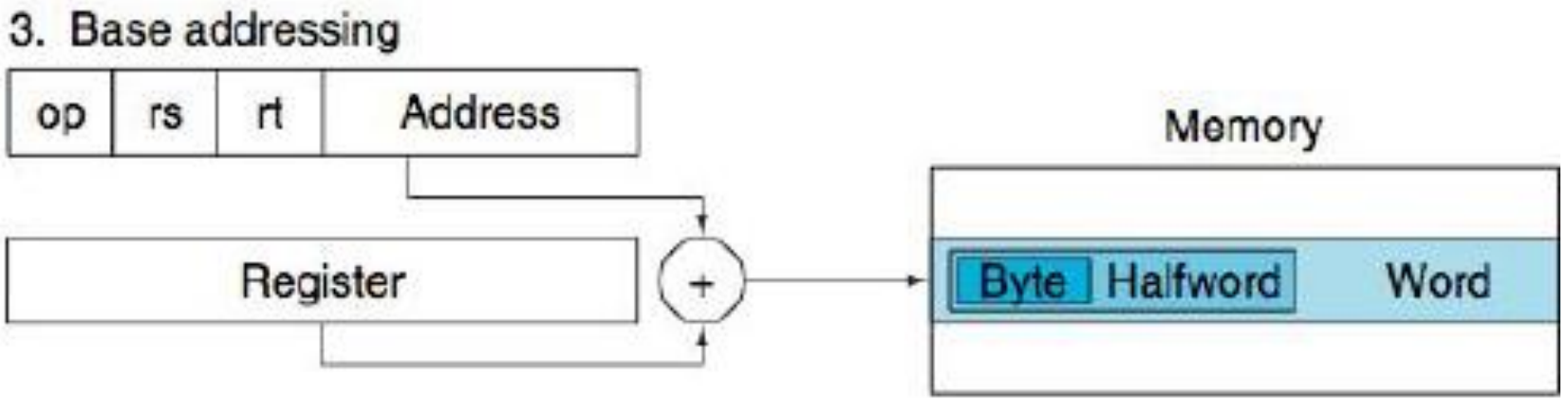
j END



Using base addressing

- IF we use base addressing – there is no need to constraint our data to start at 0.
- No need to change the memory settings.

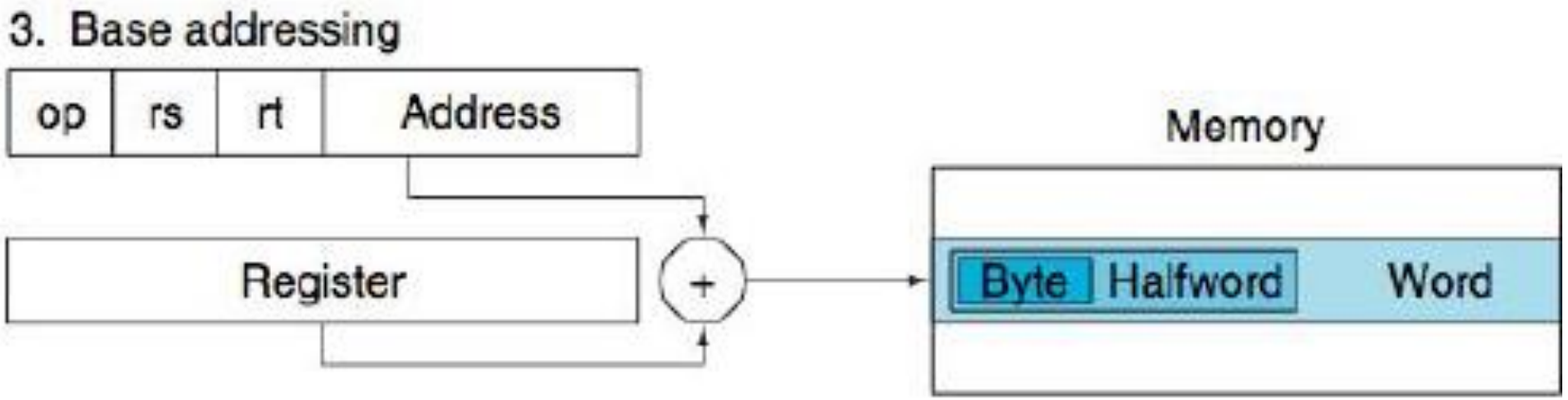
Basic	Source
`lw $9,0x00000000($16)`	`8: lw $t1,0($s0) #$t1=i`



Using base addressing

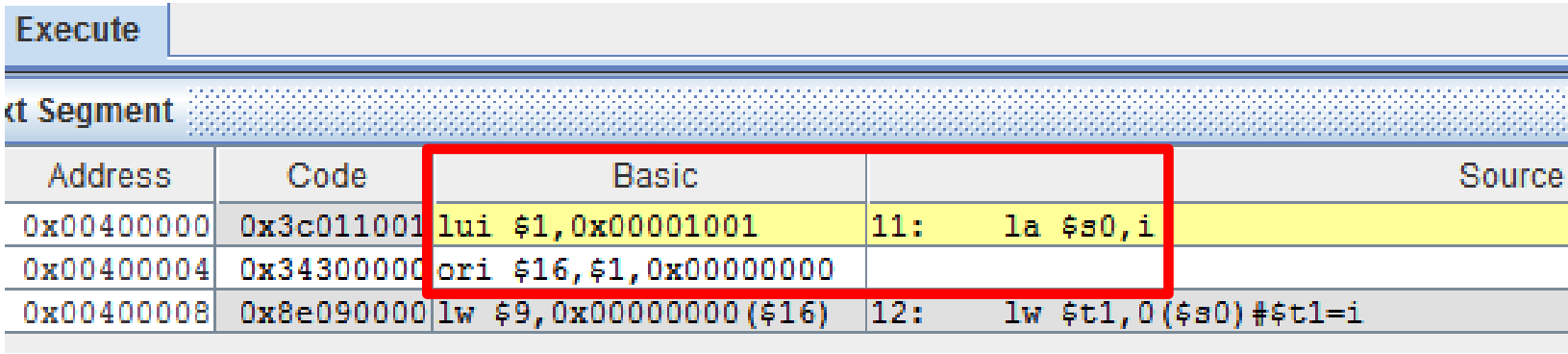
- We need to load the address of label i into a register.
- What instruction should we use?

```
1  #This program implement
2  .data
3  i: .word 1
4  .text
5
6     lw $t1,0($s0)#$t1=i
```



Pseudo Instructions

- la is not real – it is a pseudo instruction
- Used only to make the assembly code shorter and more readable.
- We can view the real instruction in the Execute tab under Basic column



lui

load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits

- Exercise:

1. load immediate number 2^{31} to register $s1

Logical instructions examples

Category	Instruction	Example	Meaning	Comments
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant

- Exercise:

1. load immediate number $2^{30}+1$ to register $s1

Pseudo Instruction examples

li $t1, 0x8000 # $t1
ori $s1,$zero,0x8000 #real instruction
li $t1, 0x80000 # $t1
lui $s1,0x8 #real instruction
ori $s1,$zero,0x0000 #real instruction
la $s1, label # $s1 <- address of *label same as*
lui $s1,0x8 #real instruction
ori $s1,$zero,0x0000 #real instruction
move $t1,$t2 #move contents of $t2 to $t1 same as
addu $t1,$zero,$t2 #real instruction

Addressing types

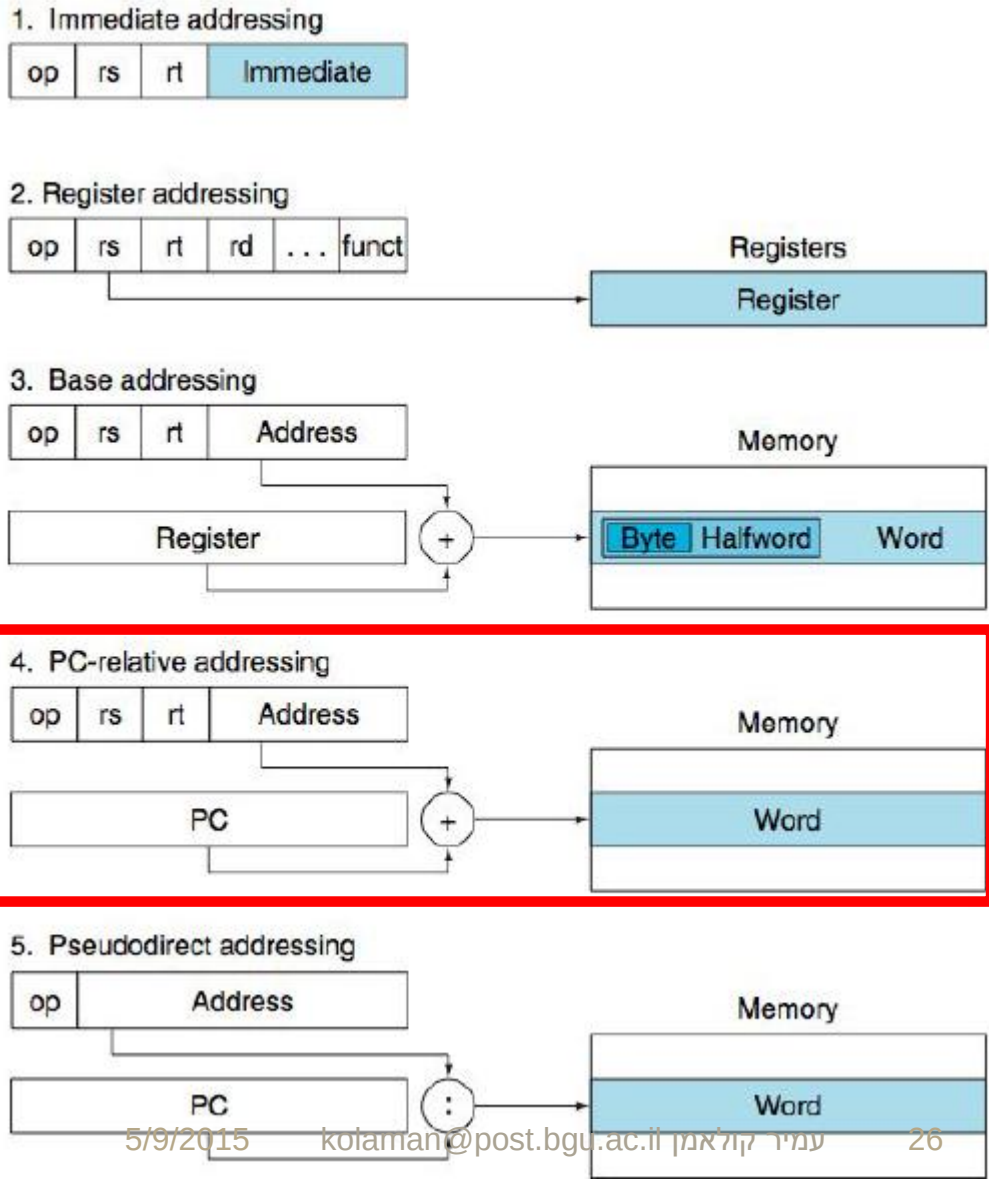
lw $t2,0

add $t5,$t3,$t4

lw $t2,0($s0)

beq $t0,$zero,ELSE

j END



Conditional branch instructions examples

Category	Instruction	Example	Meaning	Comments
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap

- In assembly it is more convenient to mark jump locations by label.

Addressing types

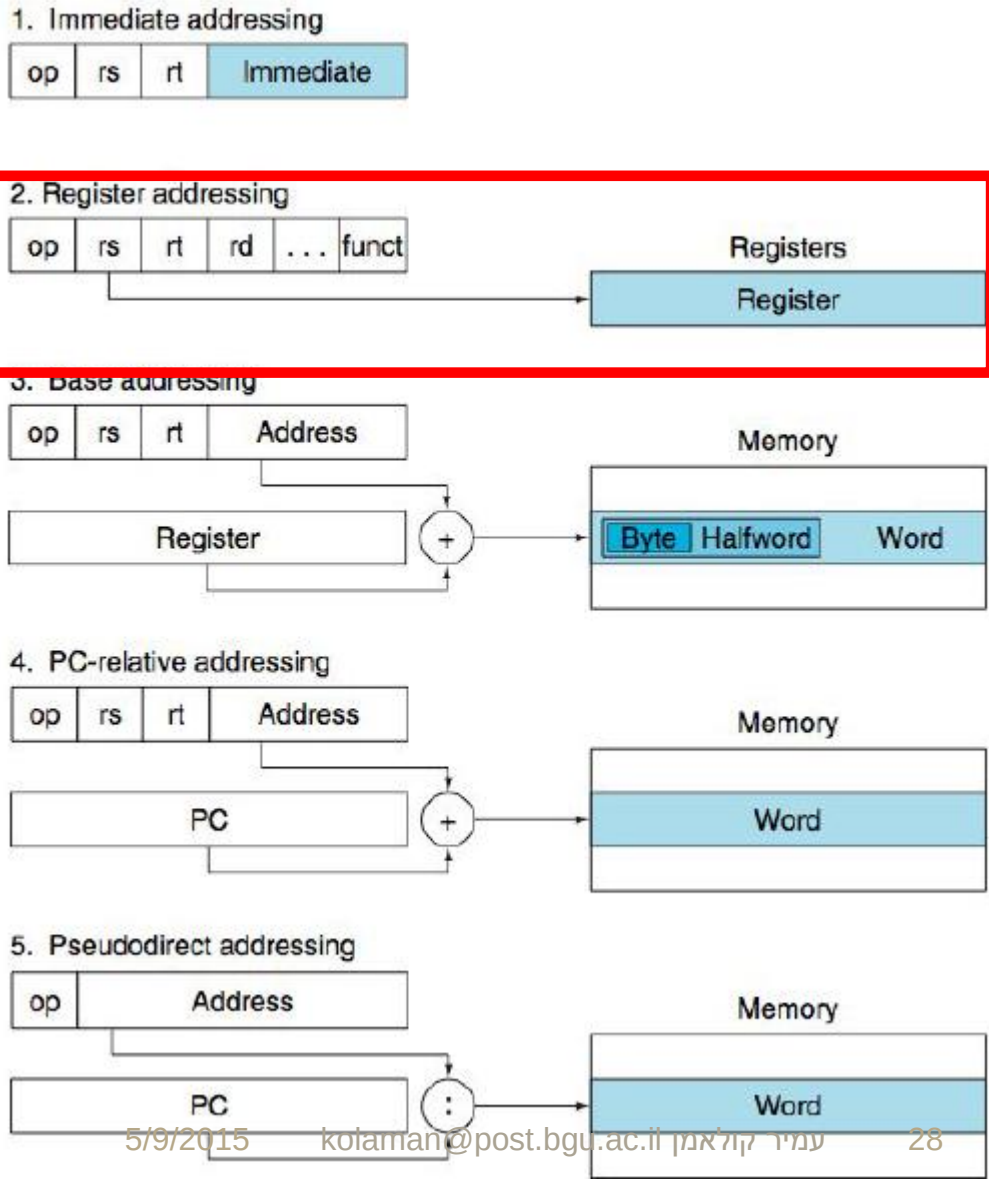
lw $t2,0

add $t5,$t3,$t4

lw $t2,0($s0)

beq $t0,$zero,ELSE

j END



Arithmetic instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants

- Exercise:
- Write down the assembly code of

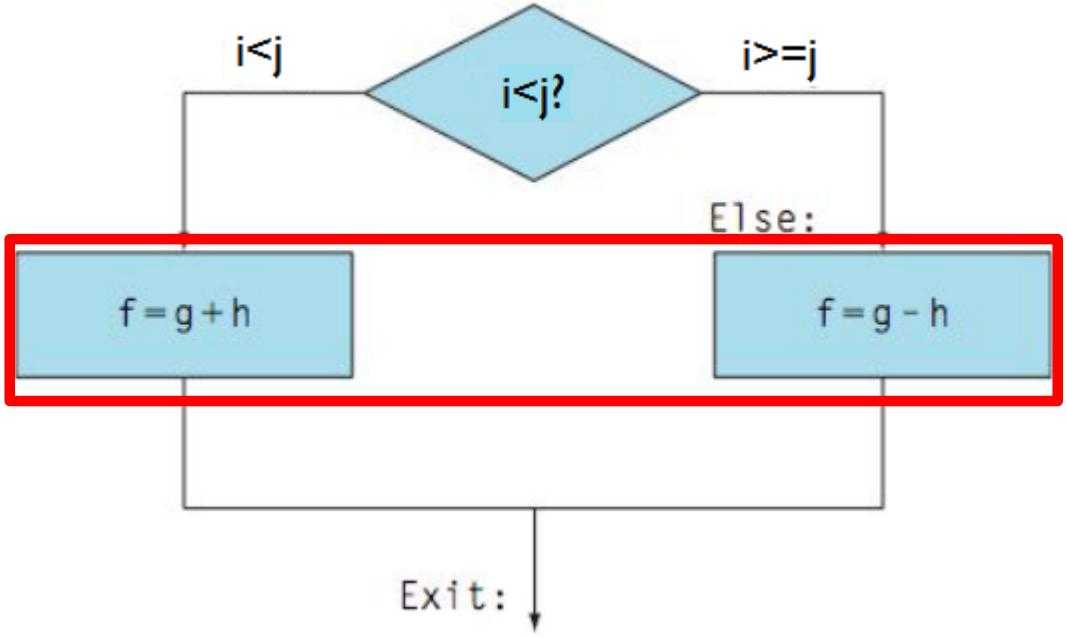
  f=g-h

  f=g+h

Assembly code example

- $s0 = f , $s1 = g , $s2 = h , $s3 = i , $s4 = j

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



Addressing types

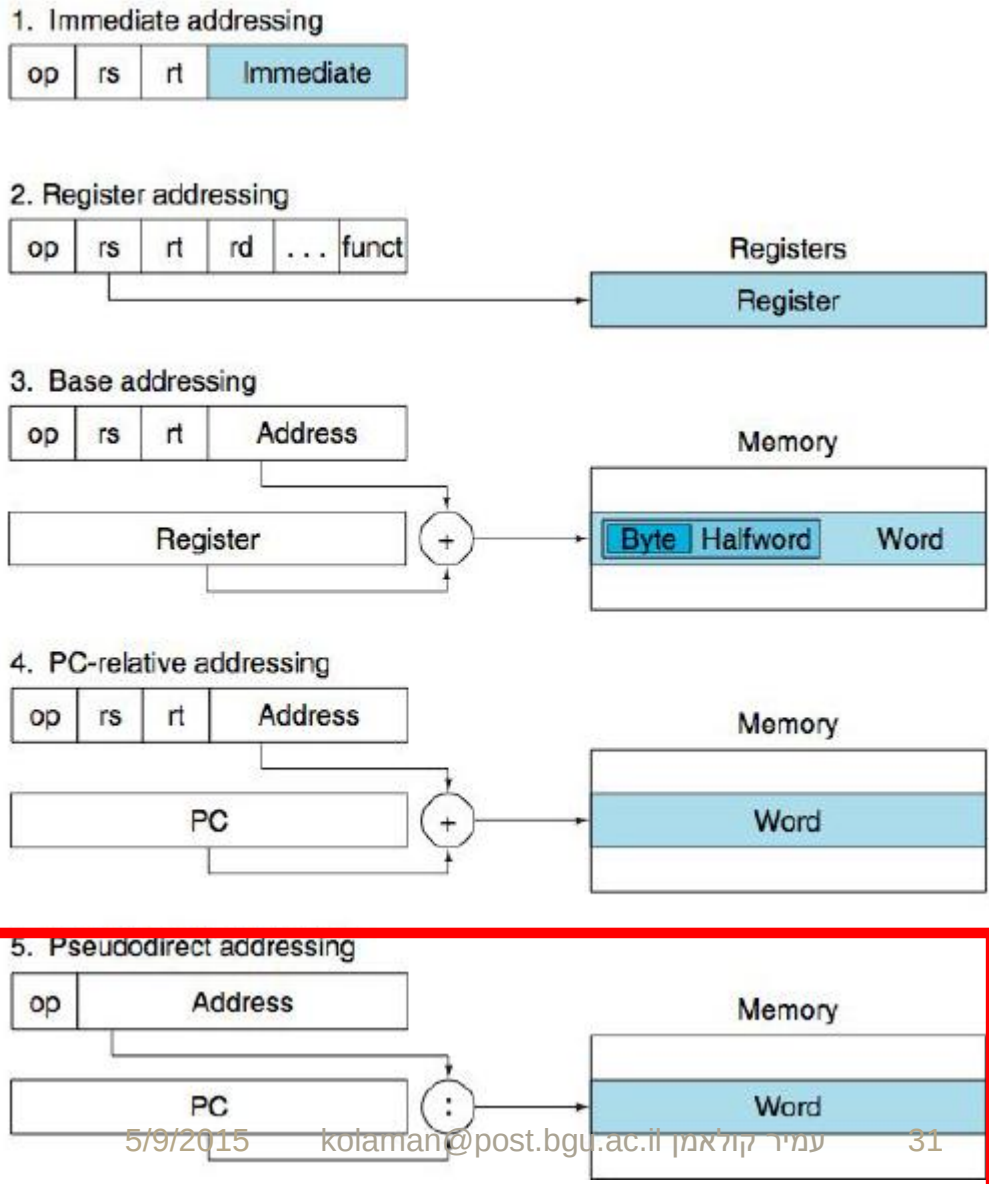
lw $t2,0

add $t5,$t3,$t4

lw $t2,0($s0)

beq $t0,$zero,ELSE

j END



Unconditional jump instructions examples

Category	Instruction	Example	Meaning	Comments
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

is there a difference between

```
loop:   beq $t0,$t0,loop
loop2:  j loop
```

MIPS Instruction Set

- Instruction overview
- MIPS assembly practice
- Instruction encoding

Instruction encoding

Field Size	6-bits	5-bits	5-bits	5-bits	5-bits	6-bits
R- Format	Opcode	Rs	Rt	Rd	Shift	Function
I - Format	Opcode	Rs	Rt	Address/immediate value		
J - Format	Opcode	Branch target address				

- 3 instruction types
- R – register
- I – immediate/address types
- J – jump type

aRithmetic instruction format

Mnemonic	Format	Opcode Field	Function Field	Instruction
Add	R	0	32	Add
Addu	R	0	33	Add Unsigned
Sub	R	0	34	Subtract
Subu	R	0	35	Subtract Unsigned
And	R	0	36	Bitwise And
Or	R	0	37	Bitwise OR
Sll	R	0	0	Shift Left Logical
Srl	R	0	2	Shift Right Logical
Slt	R	0	42	Set if Less Than
Jr	R	0	8	Jump Register (used for Return)

Field Size	6-bits	5-bits	5-bits	5-bits	5-bits	6-bits
R- Format	Opcode	Rs	Rt	Rd	Shift	Function

- Rs, Rt – source registers
- Rd – destination register

Example

- Encode the following commands
- add $t0 to $s2 and save it in $t0

op	rs	rt	rd	address/ shamt	funct
0	18	8	8	0	32
000000	10010	01000	01000	00000	100000

Data transfer format

Mnemonic	Format	Opcode Field	Function Field	Instruction
Addi	I	8	-	Add Immediate
Lui	I	15	-	Load Upper Immediate
Lw	I	35	-	Load Word
Sw	I	43	-	Store Word
Beq	I	4	-	Branch on Equal
Bne	I	5	-	Branch on Not Equal

Field Size	6-bits	5-bits	5-bits	5-bits	5-bits	6-bits
I - Format	Opcode	Rs	Rt	Address/immediate value		

- Rs – source register
- Rt – destination register
- Branch – jump to address 2^{16} bytes from current PC+4

Jump instruction format

Mnemonic	Format	Opcode Field	Function Field	Instruction
J	J	2	-	Jump
Jal	J	3	-	Jump and Link (used for Call)

Field Size	6-bits	5-bits	5-bits	5-bits	5-bits	6-bits
J - Format	Opcode	Branch target address				

- Unconditional jump for address up to 2^{26} bytes from current PC+4

MIPS op-code encoding matrix

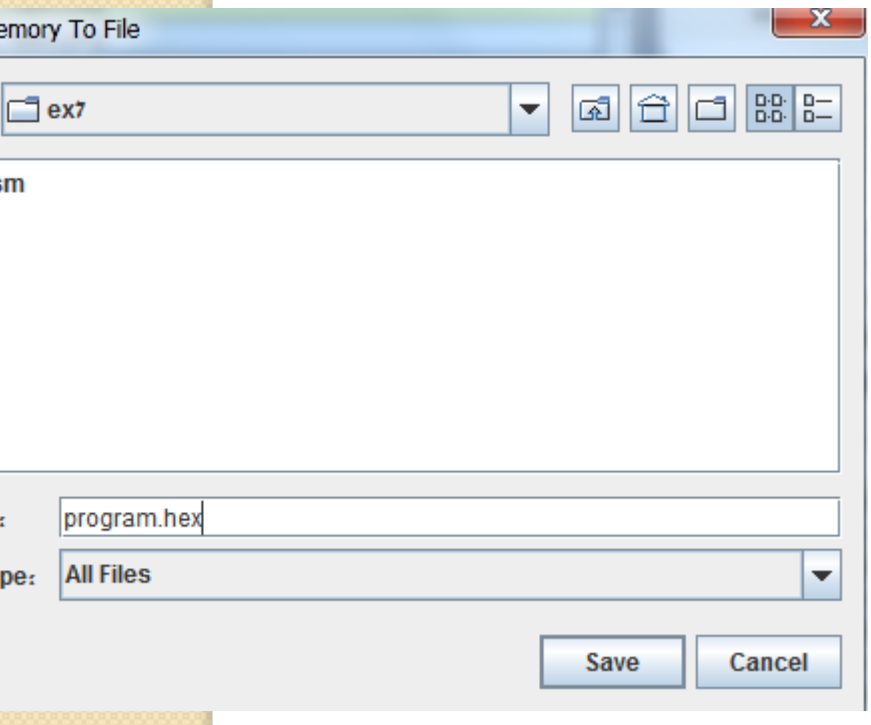
op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

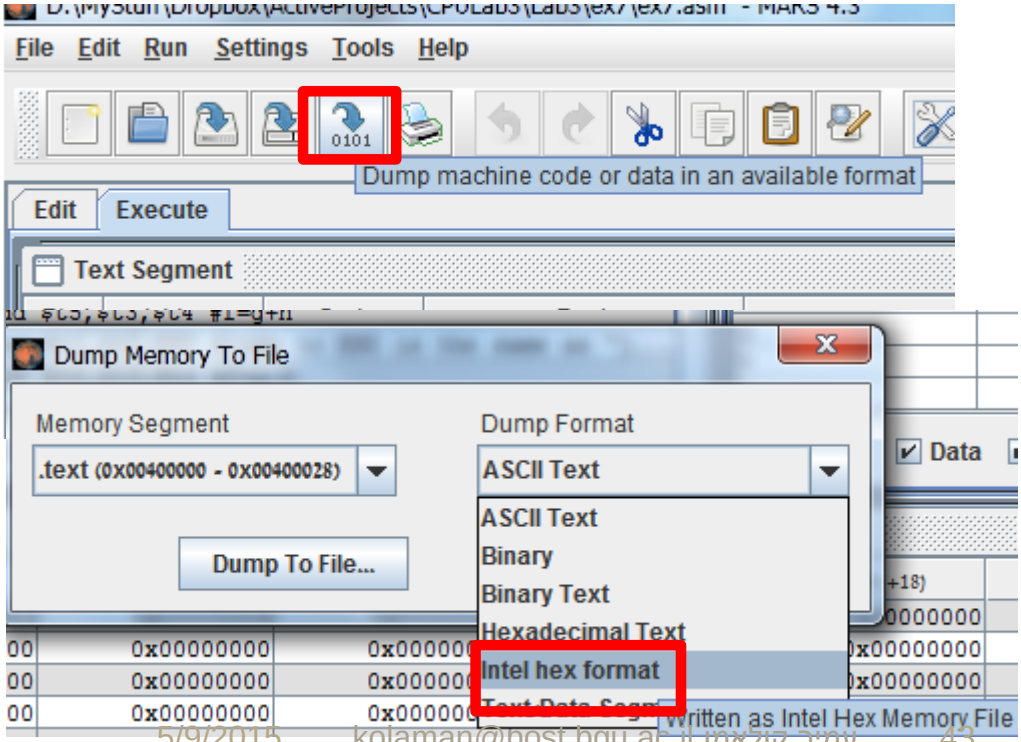
MIPS R-type function encoding matrix

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

Exporting text and code segments

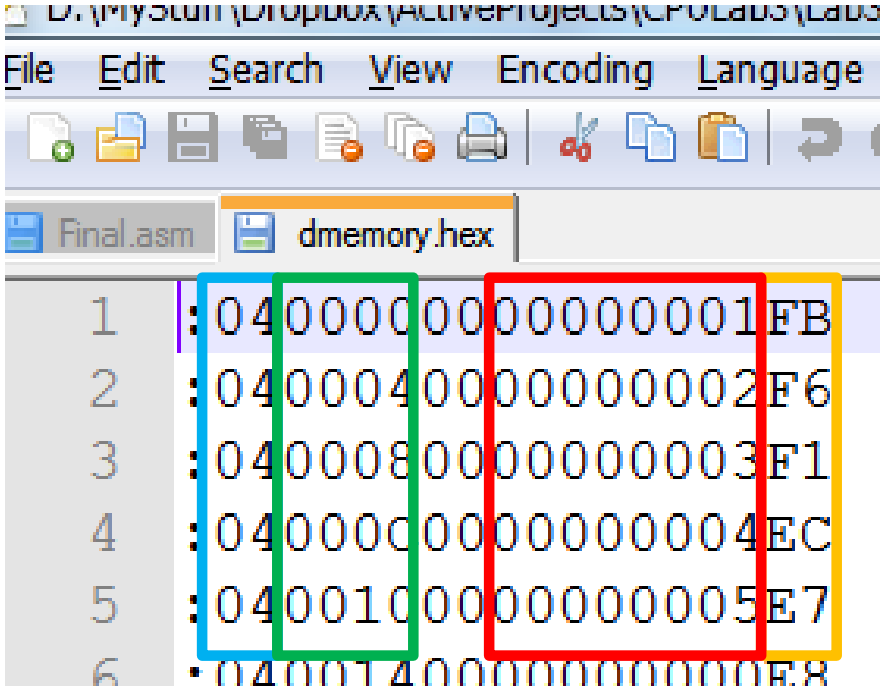
- Exporting code and data using Dump Memory option.
- Make sure **Intel hex format**





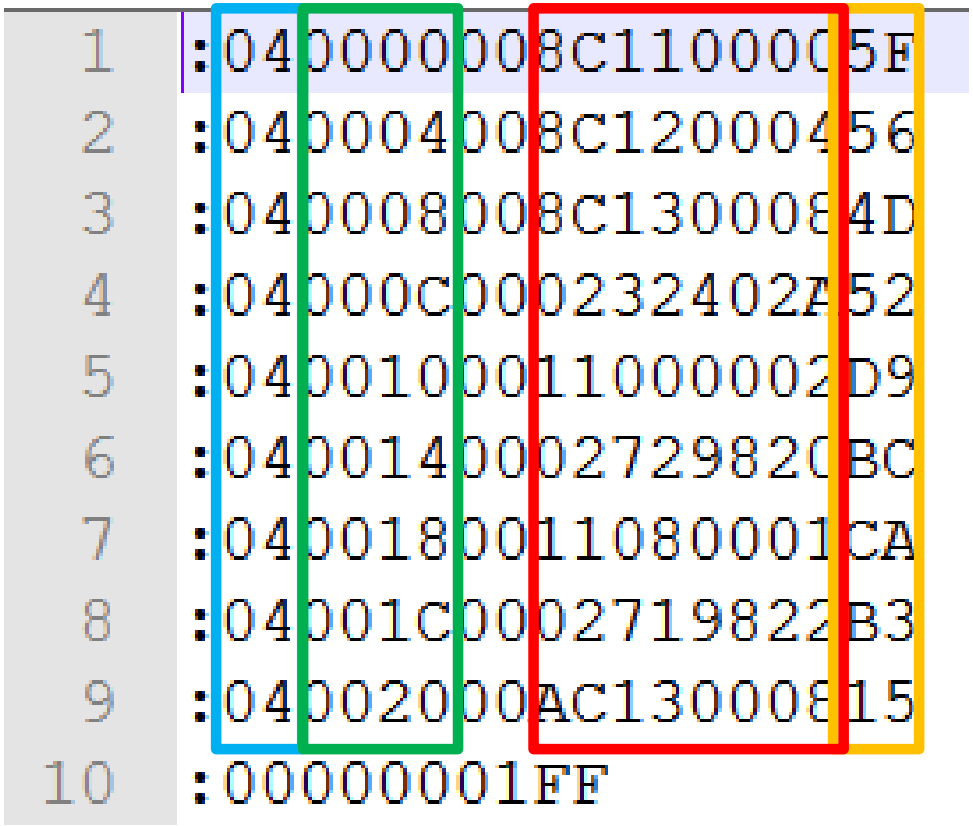
Exported data file in a text editor

- Important columns are
- Address and Data
- Less important
- number of bytes in data.
- checksum
- hex record type



Exported files can be opened in a text editor

- text is encoded in regular hex format



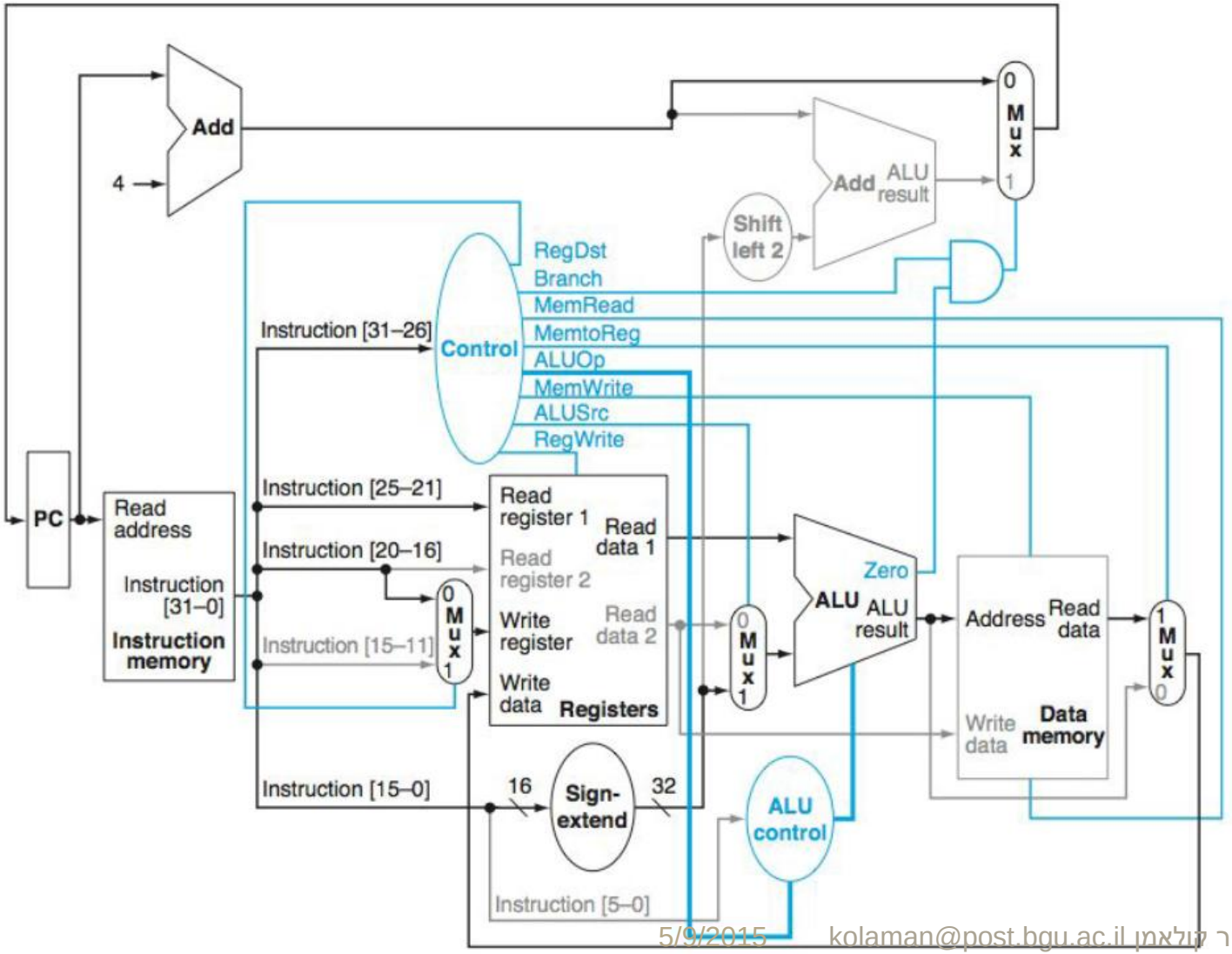
Location of the exported files

- Dump files should be saved in the quartus directory.
- They are referenced in Mega-Function entity called
- altsyncram

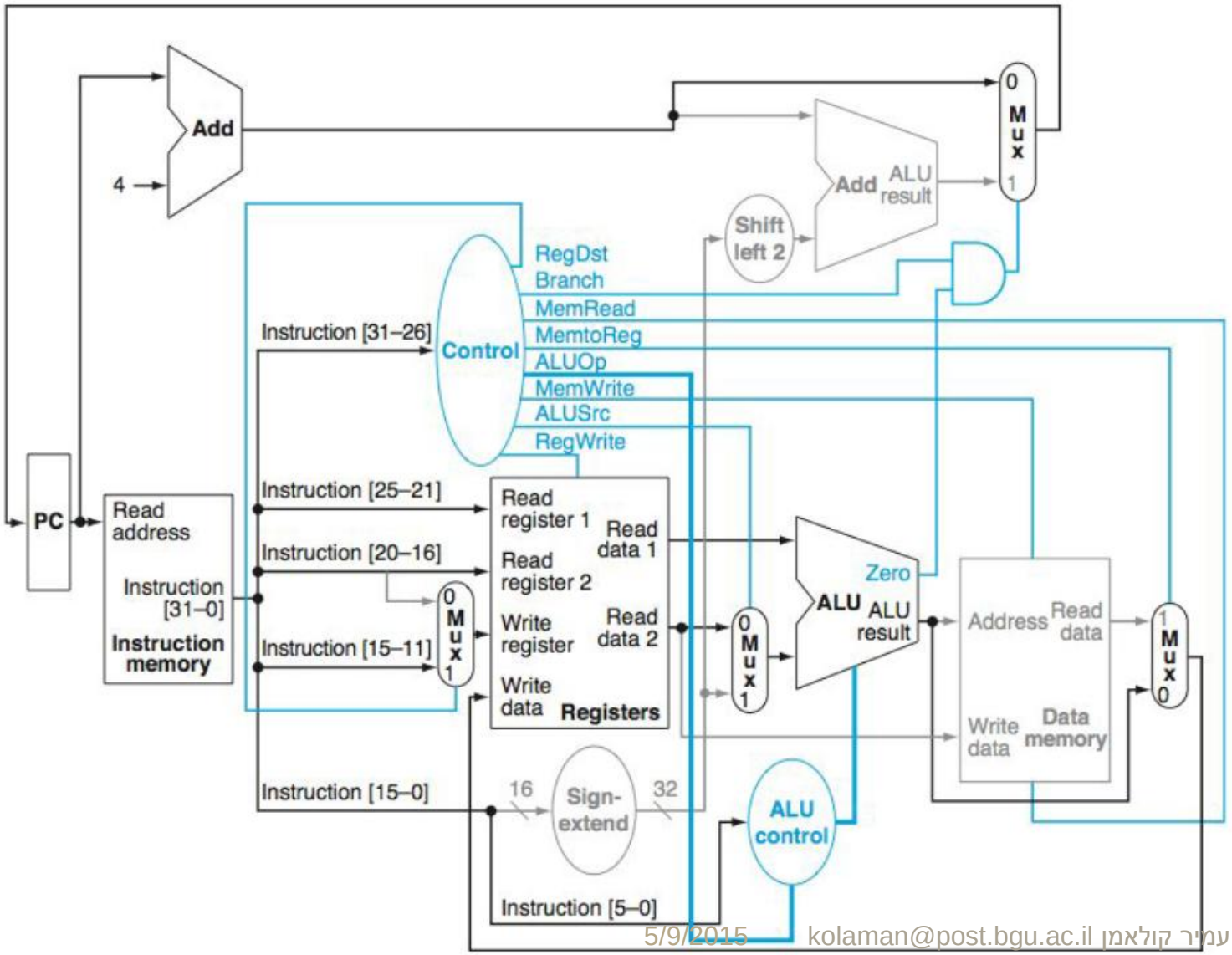
```
inst_memory: altsyncram

    GENERIC MAP (
        operation_mode => "ROM",
        width_a => 32,
        widthad_a => 8,
        lpm_type => "altsyncram",
        outdata_reg_a => "UNREGISTERED",
        init_file => "program.hex",
        intended_device_family => "Cyclone"
    )
```

```
data_memory : altsyncram
GENERIC MAP  (
    operation_mode => "SINGLE_PORT",
    width_a => 32,
    widthad_a => 8,
    lpm_type => "altsyncram",
    outdata_reg_a => "UNREGISTERED",
    init_file => 'dmemory.hex',
    intended_device_family => "Cyclone"
)
PORT MAP (
    wren_a => memwrite,
    clock0 => write_clock,
    address_a => address,
    data_a => write_data,
    q_a => read_data    );
```

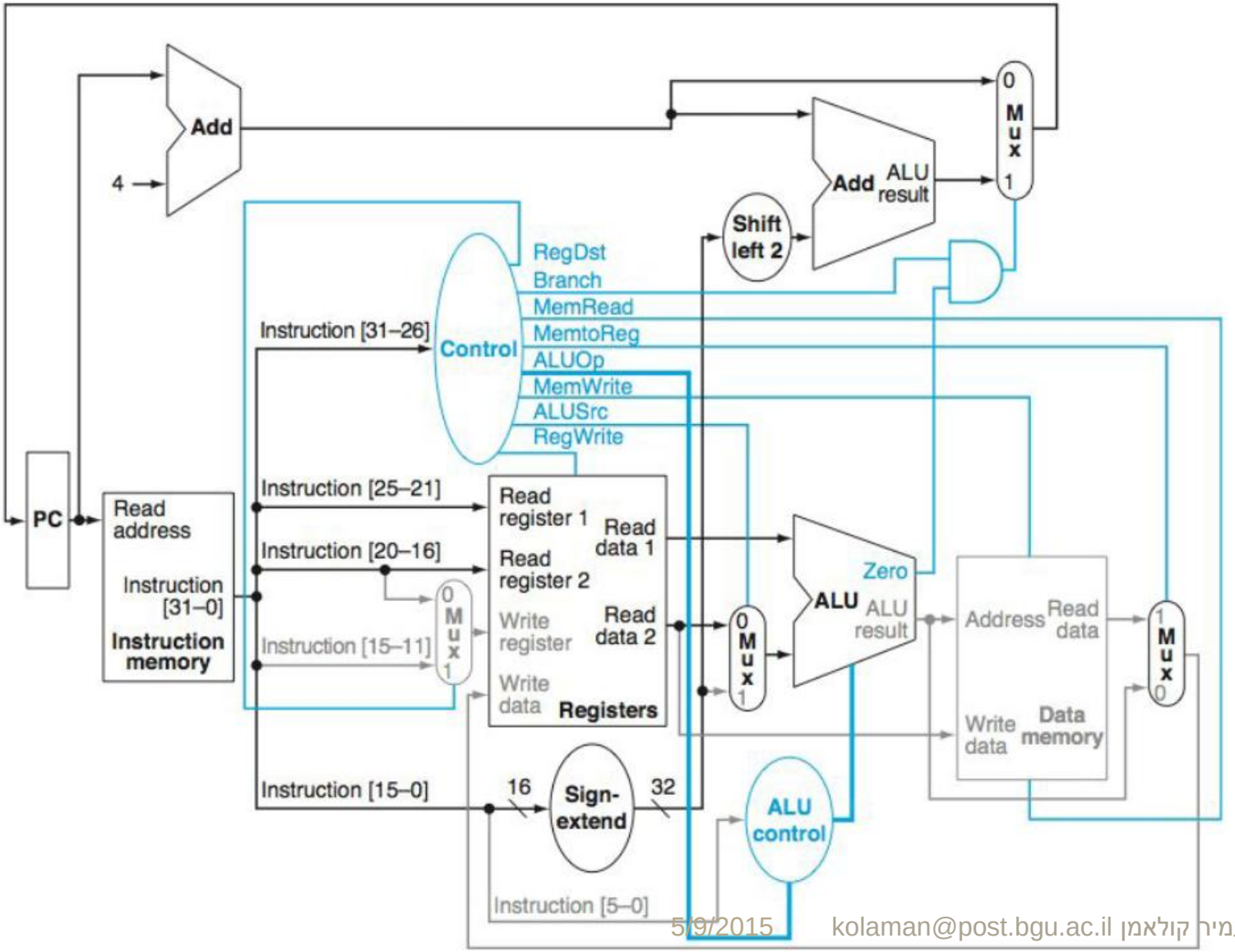
Load Instruction



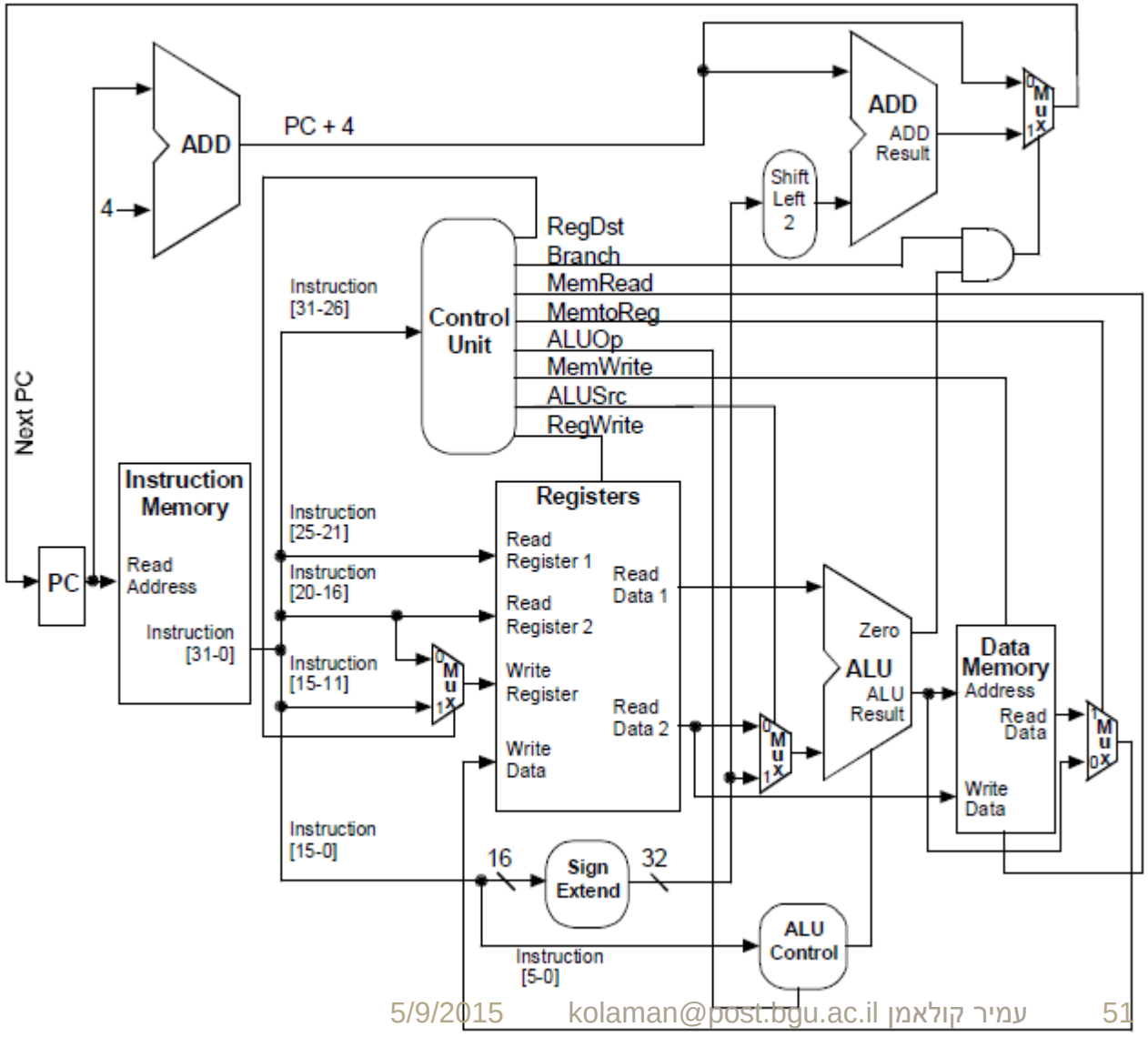
R-type instruction



Branch equal Instruction



Full diagram of our MIPS



PROJECT PART 3 OVERVIEW