

MIPS Assembly Programming

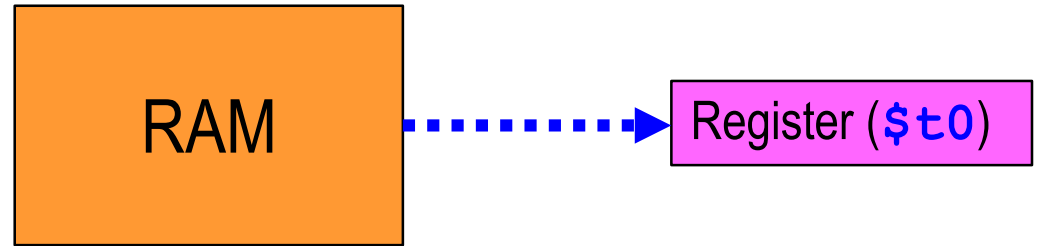
Load/Store and Arrays

MIPS

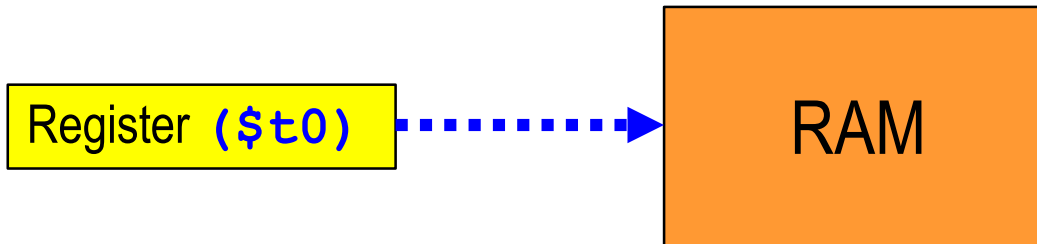
- In MIPS microprocessors ... RAM (Random Access Memory) access is allowed only with **Load** and **Store** instructions
- MIPS microprocessors (MIPS R2000 and ARM) have a «**Load/Store**» architecture.

Accessing the RAM

- **Load instructions:** Read data from RAM and copy it to a register. (**lw** **\$t0**, memory-address)



- **Store instructions:** Write data from a register to RAM. (**sw** **\$t0**, memory-address)



RAM and Registers

- For large data structures (Arrays, Images, ...)
- We have too much data to fit in ... only 32 **registers**
- We need more storage...
- So we have to use system memory (RAM)
 - Memory (RAM) is large
 - But it is very slow with respect to speed of the **registers**
- Commonly used variables are kept in **registers**.

«Load/Store», MIPS

- Use Load and Store instructions:

– **lw** = **l**oad (**w**ord)

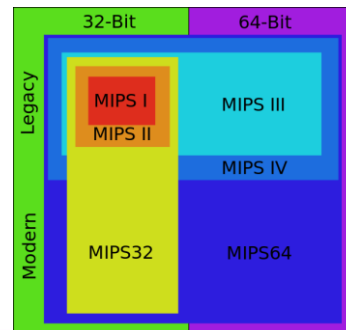
– **sw** = **s**to**re** (**w**ord)

– **lb** = **l**oad (**b**yte)

– **sb** = **s**to**re** (**b**yte).

WORD = 32 bits



BYTE = 8-bits



Load instructions

Op	Operands	Description
○ la	<i>des, addr</i>	Load the address of a label.
→ lb(u)	<i>des, addr</i>	Load the byte at <i>addr</i> into <i>des</i> .
lh(u)	<i>des, addr</i>	Load the halfword at <i>addr</i> into <i>des</i> .
○ li	<i>des, const</i>	Load the constant <i>const</i> into <i>des</i> .
lui	<i>des, const</i>	Load the constant <i>const</i> into the upper halfword of <i>des</i> , and set the lower halfword of <i>des</i> to 0.
→ lw	<i>des, addr</i>	Load the word at <i>addr</i> into <i>des</i> .
lwl	<i>des, addr</i>	
lwr	<i>des, addr</i>	
○ ulh(u)	<i>des, addr</i>	Load the halfword starting at the (possibly unaligned) address <i>addr</i> into <i>des</i> .
○ ulw	<i>des, addr</i>	Load the word starting at the (possibly unaligned) address <i>addr</i> into <i>des</i> .

Store Instructions

Op	Operands	Description
 sb	<i>src1, addr</i>	Store the lower byte of register <i>src1</i> to <i>addr</i> .
sh	<i>src1, addr</i>	Store the lower halfword of register <i>src1</i> to <i>addr</i> .
 sw	<i>src1, addr</i>	Store the word in register <i>src1</i> to <i>addr</i> .
swl	<i>src1, addr</i>	Store the upper halfword in <i>src</i> to the (possibly unaligned) address <i>addr</i> .
swr	<i>src1, addr</i>	Store the lower halfword in <i>src</i> to the (possibly unaligned) address <i>addr</i> .
o ush	<i>src1, addr</i>	Store the lower halfword in <i>src</i> to the (possibly unaligned) address <i>addr</i> .
o usw	<i>src1, addr</i>	Store the word in <i>src</i> to the (possibly unaligned) address <i>addr</i> .

MIPS addressing

MIPS addressing and modes

- Addressing; General methods to access the data in the CPU **or** the RAM:
 - Register
 - Immediate
 - Indexed (based)
 - PC relative
 - Pseudo Direct
- MIPS uses indexed (based) addressing to access the RAM.

MIPS addressing modes

Register

- Operands found in registers
 - **Example:** `add $s0, $t2, $t3`
 - **Example:** `sub $t8, $s1, $0`

Immediate

- 16-bit immediate used as an operand
 - **Example:** `addi $s4, $t5, -73`
 - **Example:** `ori $t3, $t7, 0xFF`

MIPS addressing: Register (direct)



R-type instruction

`add $rd, $rs, $rt`

rs (register source)

MIPS addressing: Immediate



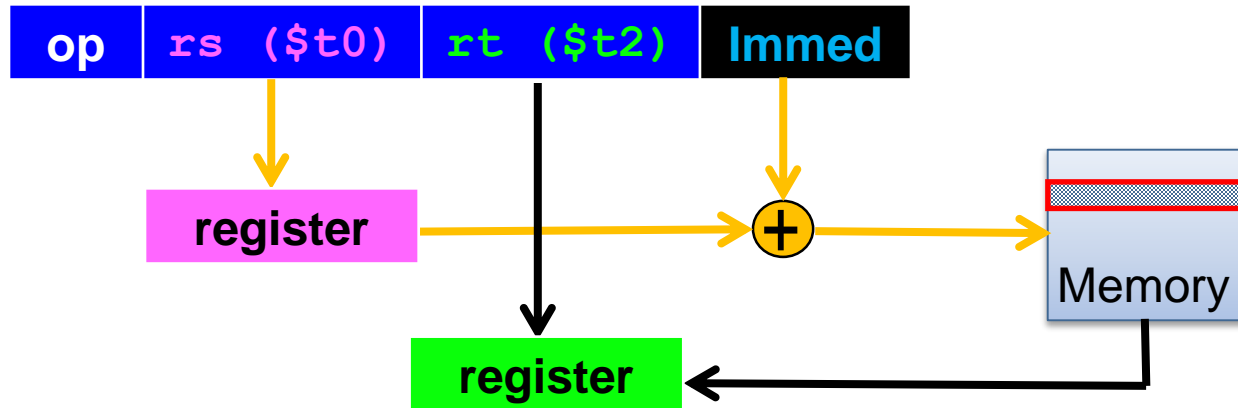
I-type instruction

`addi $rt, $rs, immed`

rt (destination register)

Indexed (Based) Addressing-Load

Store a word from a location in memory to a register.



The address operand specifies an **Immed** (signed constant or Offset) and a **rs** (**r**egister **s**ource) that holds the based-address

```
lw $t2, 4($t0)
```

$\$t2 \leftarrow \text{Mem}[\$t0+4]$

op code	rs (\$t0)	rt (\$t2)	Address/Immediate
100011	01000	01010	0000 0000 0000 0100

Based or Indexed addressing

- The address operand specifies an **Immed** (signed constant or Offset) and a **rs** (register source) that holds the based-address

```
lw $t2, 4($t0)
```

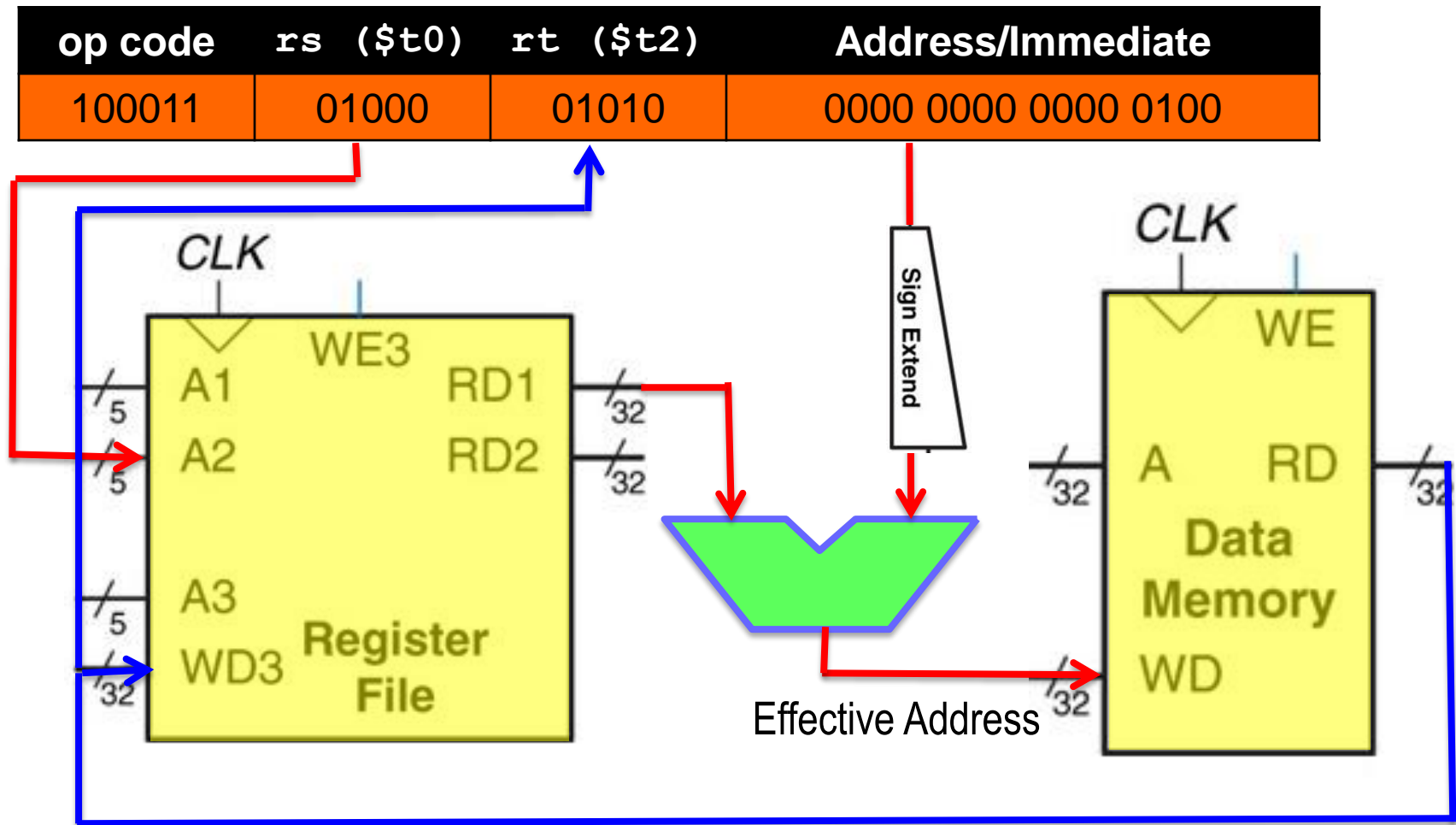
- The actual memory location from where the operand is retrieved for an instruction is the Effective Address (EA).
- The Effective Address (EA) is determined by **adding** the offset to the Register (based-address)

$$EA \leftarrow \text{Mem}\{ \$(\text{?}) + \text{sign-ext}_{32}(\text{offset}) \}$$

```
$t2 ← Mem{ $t0 + 00000000000000000000000000000000100 }
```

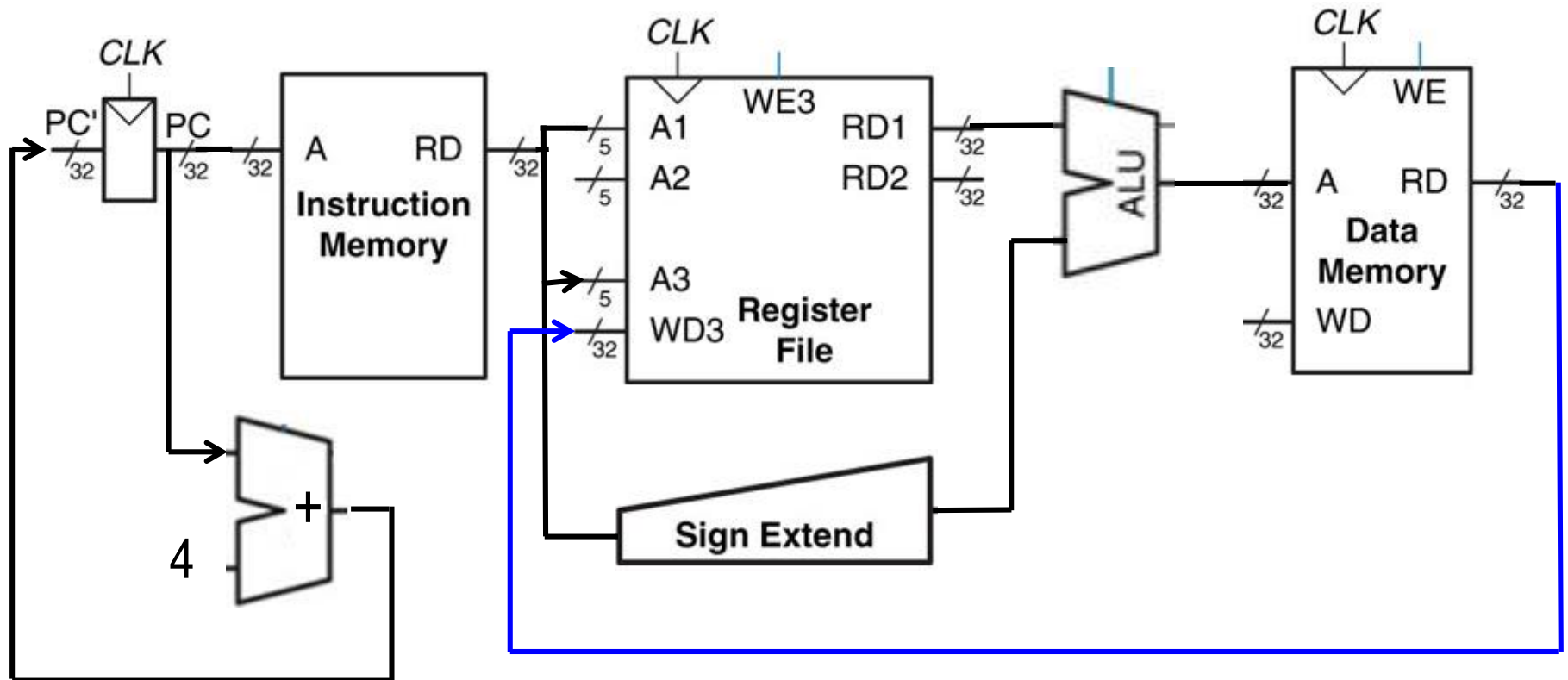
MIPS uses INDEXED ADDRESSING

lw \$t2, 4(\$t0)



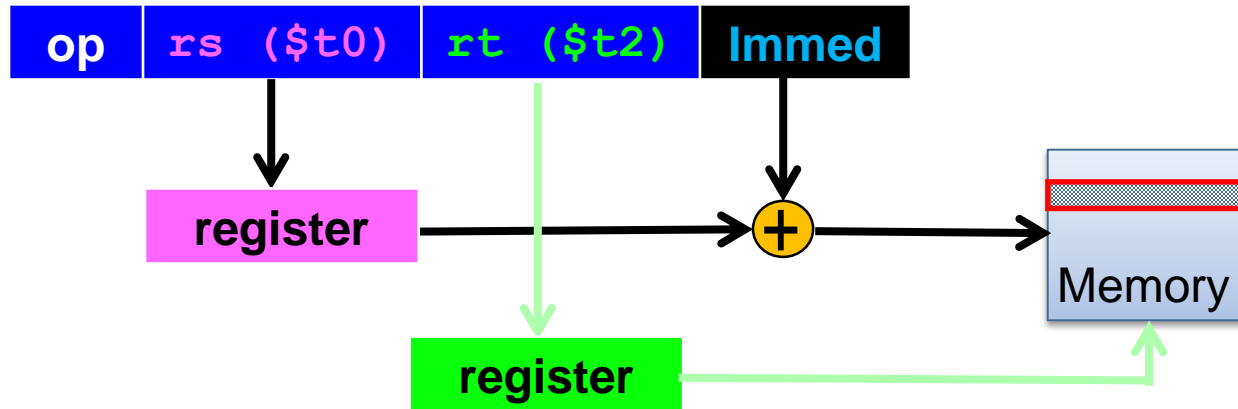
rs is the first source register
rt is the second source register

`lw $t2, 4($t0)`



Indexed (Based) Addressing; **store**

Store a word from a register to a location in memory.



sw **\$t2**, **4 (\$t0)**

$\$t2 \rightarrow \text{Mem}[\$t0+4]$

op code	rs (\$t0)	rt (\$t2)	Address/Immediate
100011	01000	01010	0000 0000 0000 0100

MIPS memory access instructions

- **WORD** [32 (64-bits)]
- **BYTE** [8-bits].

MIPS memory access instructions

- **WORD** (**Address in memory must be word-aligned**)
 - **lw**; Loads a word from a location in memory to a register
 - **sw**; Store a word from a register to a location in memory
- **BYTE** (**Address not aligned-Only one byte is loaded from memory**)
 - **lb**; Loads a byte from a location in memory to a register. Sign extends this result in the register.
 - **sb**; Store the least significant byte of a register to a location in memory.

Memory alignment

MIPS requires that all words start at byte addresses and are multiples of 4 bytes ($4 \times 8 = 32$ -bits)

4	3	2	1
C	8	4	0

Aligned

Address in
0,4,8, C in Hexadecimal

Memory alignment

MIPS requires that all words start at byte addresses are multiples of 4 bytes ($4 \times 8 = 32$ -bits)

Not Aligned

4	3	2	1
C	8	4	0
1C	18	14	10
C	8	4	0

`.align #` directive the next datum on a 2^n byte boundary.

Load Word (Memory Read)

lw

Word addressable memory

- Each 32-bit data word has a unique address

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
. 00000000	A B C D E F 7 8	Word 0

Reading Word-Addressable Memory

- Memory read is called **load**
- **Mnemonic:** load word (**lw**)
- **Format:**
lw \$t2, 1(\$t0)
- Effective Address calculation: $EA \leftarrow \text{Mem}\{ \$(\text{s}) + \text{sign-ext}_{32}(\text{offset}) \}$
 - **add** base-address (**\$t0**) to the *offset* (**1**)
 - Effective-address: **\$t2** \leftarrow **Mem**[**\$t0 + 1**]
- **Result:**
 - **\$t2** holds the value at effective-address (**\$t0 + 1**)

Any register may be used as base address

Reading Word-Addressable Memory

- **Example:** read a word of data at memory address 0x00000001 into register: **\$t2**
 - Effective Address:
\$t2 \leftarrow **Mem**[**\$t0** + 1] = **0x00000001**
 - **\$t2** holds the value: **0xF2F1AC07** after load

Assembly code

\$t0 = 0x00000000 (base address)

lw \$t2, 1(\$t0) #read memory word 1 into \$t2

Register File	
Reg	value
\$t0	
...	
\$t2	0xF2F1AC07 ←

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Store Word (Memory Write)

SW

Writing Word-Addressable Memory

- Memory write are called **store**
- Mnemonic: **s**store **w**ord (**sw**)
- Format:

sw **\$s0**, **3(\$t0)**

- Effective address: **\$s0** → **Mem****[\$t0 + 3]**

Writing Word-Addressable Memory

- Example: Write (store) the value in **\$s0** into memory address 0x00000003
 - Effective-address:
Mem[\$t0 + 3] → 0x00000000+3=0x00000003
 - To the above address load the word: **0x40F30788**

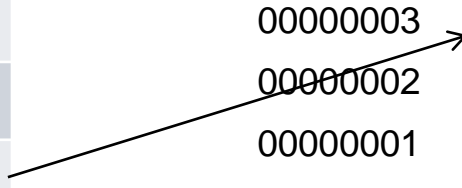
Assembly code

\$t0 = 0x00000000 (base address)

sw \$s0, 3(\$t0) #write the value in \$s0 to memory word 3

Register File	
Reg	value
\$t0	
...	
\$s0	0x40F30788

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

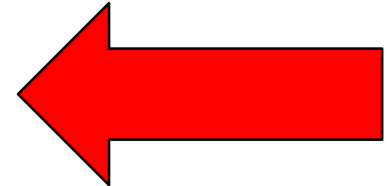


Load **W**ord and Load **B**yte

- **lw** \$rt, offset(\$rs)

$$EA \leftarrow \$(s) + \text{sign-ext}_{32}(\text{offset})$$

- **lb** \$rt, offset(\$rs)



Load ... Store >> Byte

- `lb $rt, offset($rs)`
 - `lb register(destination) , RAM(source)`
 - copy byte at source RAM location to low-order byte of destination register
- `sb $rs, offset($rt)`
 - `sb register(source) , RAM(destination)`
 - store byte (low-order) in source register into RAM destination.

Byte-Addressable Memory

- Next lecture

Load immediate, Load RAM address

- `li $t0, value`
 - Load the immediate `value` into destination register `$t0`
- `la $t1, var`
 - Load RAM address `var` (label defined in the program) into register `$t1`

REVIEW

Immediate addressing

- `lw $t2, var`
 - load word, from RAM address `var`, into `$t2`
- `sw $t2, var`
 - store word, from register `$t2`, into RAM address `var`

Example-1: `lw` ...

```
        .data
var:     .word    9

        .text

start:

        lw      $t0, var
        addi    $t0, $t0, 3
        sw      $t0, var
```

declare storage for `var`; initial value is 9

load contents of RAM location into register `$t0`: `$t0 ← var`

add 3 to it

store contents of register `$t0` into RAM: `$t0 → var`

`$t0 = ?`

Assemble ... GO

\$zero	0	0
\$at	1	268500992
\$v0	2	0
\$v1	3	0
\$a0	4	0
\$a1	5	0
\$a2	6	0
\$a3	7	0
\$t0	8	12
\$t1	9	0
\$t2	10	0
\$t3	11	0
\$t4	12	0

start	0x00400000
var	0x10010000

Example-2: `lw` ...

```
        .data
var:     .word    9

        .text

start:

lw      $t0, var

li      $t1, 5

sw      $t1, var
```

declare storage for `var`; initial value is 9

load contents of RAM location into register `$t0`: $\$t0 \leftarrow \text{var}$

$\$t1 = 5$ (“load immediate”)

store contents of register `$t1` into RAM: $\$t1 \rightarrow \text{var}$

$\$t0 = ?$
 $\$t1 = ?$

Assemble ... GO

\$zero	0	0
\$at	1	268500992
\$v0	2	0
\$v1	3	0
\$a0	4	0
\$a1	5	0
\$a2	6	0
\$a3	7	0
\$t0	8	9
\$t1	9	5
\$t2	10	0
\$t3	11	0
\$t4	12	0

Based or Indexed addressing

- `lw $t2, 4($t0) # $t2 ← Mem[$t0 + 4]`
 - load word at RAM address `($t0+4)` into register `$t2`
 - `$t0` contains the base address
 - "4" gives offset from address in register `$t0`
- `sw $t2, 4($t0) # $t2 → Mem[$t0 + 4]`
 - store word in register `$t2` into RAM at address `($t0 + 4)`
 - `$t0` contains the base address
 - negative offsets are fine
- Note: based addressing is especially useful for:
 - **arrays**; access elements as offset from base address
 - **stacks**; easy to access elements at offset from stack pointer or frame pointer

Based or Indexed addressing

- `lw $t2, 4($t0) # $t2 ← Mem[$t0 + 4]`

Base register = pointer to the array

Offset = the location in the array

- `sw $t2, 4($t0) # $t2 → Mem[$t0 + 4]`

`$t2 ← Memory[$t0 + 4] ... Pseudocode`

- Pointer to the array/Location in the array

Example ... 0 (\$x) ... 4 (\$x) ...

```
.text
```

```
.globl main
```

```
main:
```

```
la    $t0, x
```

copy RAM address of 'x' into \$t0

```
lw    $t1, 0($t0)
```

load word at RAM address 0(\$t0) into \$t1

```
la    $t2, y
```

copy RAM address of 'y' into \$t2

```
lw    $t2, 4($t0)
```

load word at RAM address 4(\$t0) into \$t2

```
add   $t3, $t1, $t2
```

add \$t1 and \$t2 into \$t3

```
sw    $t3, 8($t0)
```

store word \$t3 into RAM address 8(\$t0)

```
.data
```

```
.align 2
```

align it on 4-byte (word) boundary

```
x:    .word 3
```

Our data for x

```
y:    .word 5
```

Our data for y

Note that: `.align x`

- `.align x`

- `x` = 0 (byte)

- `x` = 1 (half)

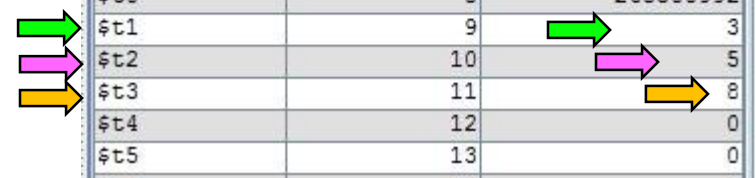
- `x` = 2 (Word)

- `x` = 3 (double)

Assemble ... GO

```
2
3      .text
4      .globl main
5
6  main:
7      la    $t0, x
8      lw    $t1, 0($t0)
9      la    $t2, y
10     lw    $t2, 4($t0)
11     add   $t3, $t1, $t2
12     sw    $t3, 8($t0)
13
14     .data
15     .align 2
16     x: .word 3
17     y: .word 5
18
```

Registers		Coproc 1	Coproc 0
Name	Number	Value	
\$zero	0	0	
\$at	1	268500992	
\$v0	2	0	
\$v1	3	0	
\$a0	4	0	
\$a1	5	0	
\$a2	6	0	
\$a3	7	0	
\$t0	8	268500992	
\$t1	9	3	
\$t2	10	5	
\$t3	11	8	
\$t4	12	0	
\$t5	13	0	



\$t1 = ?
\$t2 = ?
\$t3 = ?

Address

Address	Code	Basic	
4194304	0x3c011001	lui \$1,4097	6: main: la \$t0, x
4194308	0x34280000	ori \$8,\$1,0	
4194312	0x8d090000	lw \$9,0(\$8)	7: lw \$t1, 0(\$t0)
4194316	0x3c011001	lui \$1,4097	8: la \$t2, y
4194320	0x342a0004	ori \$10,\$1,4	
4194324	0x8d0a0004	lw \$10,4(\$8)	9: lw \$t2, 4(\$t0)
4194328	0x012a5820	add \$11,\$9,\$10	10: add \$t3, \$t1, \$t2
4194332	0xad0b0008	sw \$11,8(\$8)	11: sw \$t3, 8(\$t0)

0, 0+4=4, 4+4=8, 8+4=12, 12+4=16

```
add-memory-align5.asm*
1  # lw
2
3      .text
4      .globl main
5
6  main:
7      la $t0, x
8      lw $t1, 0($t0)
9      la $t2, y
10     lw $t2, 4($t0)
11     la $t3, w
12     lw $t3, 8($t0)
13     la $t4, z
14     lw $t4, 12($t0)
15     la $t5, k
16     lw $t5, 16($t0)
17
18     .data
19     .align 2
20     x: .word 3
21     y: .word 5
22     w: .word 7
23     z: .word 9
24     k: .word 11
```

Registers		
Name	Number	Value
\$zero	0	0
\$at	1	268500992
\$v0	2	0
\$v1	3	0
\$a0	4	0
\$a1	5	0
\$a2	6	0
\$a3	7	0
\$t0	8	268500992
\$t1	9	3
\$t2	10	5
\$t3	11	7
\$t4	12	9
\$t5	13	11
\$t6	14	0
\$t7	15	0
\$a0	16	0

MIPS Arrays

Arrays

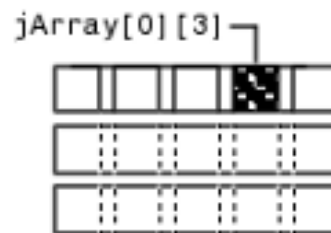
- Access large amounts of similar data
 - **Index:** access each element
 - **Size:** number of elements



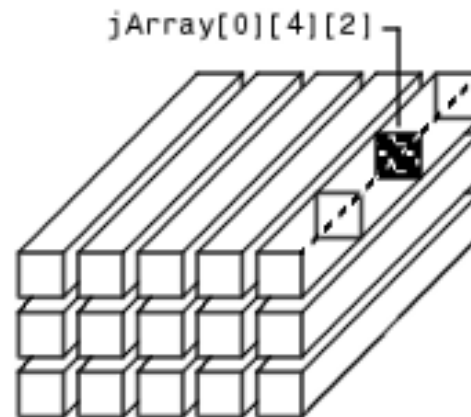
Array Access from Java



Simple Array

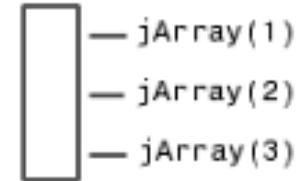


Array of Arrays

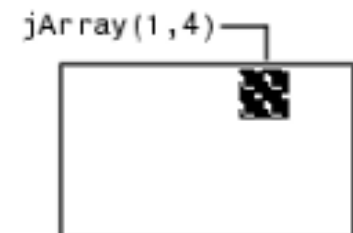


Array of Arrays of Arrays

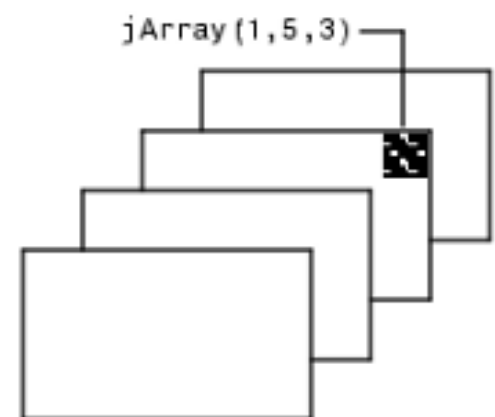
Array Access from MATLAB



One-dimensional Array



Two-Dimensional Array



Three-Dimensional Array

Arrays

- 5-element array
- **Base address** = 0x12348000 (address of first element, **array[0]**)
- First step in accessing an array: load base address into a register

0x12340010	array[4]
0x1234800C	array[3]
0x12348008	array[2]
0x12348004	array[1]
0x12348000	array[0]

“Array” example

```
.data
Array: .space 4
.text
la    $t0, Array
li    $t1, 5
sw    $t1, 0($t0)
lw    $t2, 0($t0)
move  $a0, $t2
li    $v0, 10
syscall
```

Reserves a [free] space of 4-bytes or 32-bits
(space for just one integer 32 bits = 4-bytes)

write address of array into register **\$t0**

\$t1 = 5

store the first element into the array cell;
(indirect addressing)

get the first element from the array cell

\$a0 must hold address of integer to print

Resulting values in the registers: **\$v0**, **\$a0**, **\$t1**, **\$t2**?

```
1  # Arrays-fine/MIPS-0
2  # create 1-integer array; example
3  #-----
4      .data
5  Array: .space 4
6      .text
7
8      la $t0, Array
9      li $t1, 5
10
11     sw $t1, 0($t0)
12     lw $t2, 0($t0)
13     move $a0, $t2
14
15     li $v0, 10
16     syscall
17
```

\$v0 = ?

\$a0 = ?

\$t1 = ?

\$t2 = ?

Assemble ... GO

```
1  # Arrays-fine/MIPS-0
2  # create 1-integer array; example
3  #-----
4      .data
5  Array: .space 4
6      .text
7
8      la $t0, Array
9      li $t1, 5
10
11     sw $t1, 0($t0)
12     lw $t2, 0($t0)
13     move $a0, $t2
14
15     li $v0, 10
16     syscall
17
```

Registers			Coproc 1	Coproc 0
Name	Number	Value		
\$zero	0	0		
\$at	1	268500992		
\$v0	2	10		
\$v1	3	0		
\$a0	4	5		
\$a1	5	0		
\$a2	6	0		
\$a3	7	0		
\$t0	8	268500992		
\$t1	9	5		
\$t2	10	5		
\$t3	11	0		
\$t4	12	0		
\$t5	13	0		
\$t6	14	0		
\$t7	15	0		
\$s0	16	0		
\$s1	17	0		
\$s2	18	0		
\$s3	19	0		
\$s4	20	0		
\$s5	21	0		
\$s6	22	0		
\$s7	23	0		
\$t8	24	0		
\$t9	25	0		
\$k0	26	0		
\$k1	27	0		
\$gp	28	268468224		
\$sp	29	2147479548		
\$fp	30	0		
\$ra	31	0		
pc		4194336		
hi		0		
lo		0		

The same example

What is the difference?

```
4      .data
5  Array: .space 4
6      .text
7
8      la $t0, Array
9      li $t1, 5
10
11     sw $t1, 0($t0)
12     lw $t2, 0($t0)
13     move $a0, $t2
14
15     li $v0, 10
16     syscall
17
```

```
4      .data
5  Array: .space 4
6      .text
7      #-----
8      la $t0, Array
9      li $t1, 5
10     #-----
11     sw $t1, 0($t0)
12     lw $t2, 0($t0)
13     #-----
14     move $a0, $t2
15     li $v0, 1
16     syscall
17     #-----
18     li $v0, 10
19     syscall
```

Create an one-integer array. Print its contents to the console

```
        .data
Array:  .space 4

        .text

la      $t0, Array
li      $t1, 5
sw      $t1, 0($t0)
lw      $t2, 0($t0)
move    $a0, $t2
li      $v0, 1
syscall

li      $v0, 10
syscall
```

load `syscall 1` → `$v0` register;

Assemble ... GO

```
4      .data
5  Array: .space 4
6      .text
7      #-----
8      la $t0, Array
9      li $t1, 5
10     #-----
11     sw $t1, 0($t0)
12     lw $t2, 0($t0)
13     #-----
14     move $a0, $t2
15     li $v0, 1
16     syscall
17     #-----
18     li $v0, 10
19     syscall
```

Print to console

```
5
-- program is finished running --
```

Another “Array” example

```
3      .data
4  Array: .space 8
5      .text
6      #-----
7      la $t0, Array
8      #-----
9      li $t1, 5
10     sw $t1, 0($t0)
11     li $t1, 6
12     sw $t1, 4($t0)
13     #-----
14     lw $t2, 0($t0)
15     move $a0, $t2
16     li $v0, 1
17     syscall
18
19     lw $t3, 4($t0)
20     move $a0, $t3
21     li $v0, 1
22     syscall
23     #-----
24     li $v0, 10
25     syscall
26
```

Two-Integer Array example;

ArraysMIPS-2.asm*

```
1  # Create a 2-integer array and print its contents to the console
2  #-----
3  .data
4  Array: .space 8          # 8 bytes of an integer-array-4 bits for each integer
5  .text
6  #-----
7  la $t0, Array            # write address of array into register $t0
8  #-----
9  li $t1, 5                # $t1 = 5
10 sw $t1, 0($t0)           # store the first element into the array cell; indirect addressing
11 li $t1, 6                # $t1 = 6
12 sw $t1, 4($t0)           # store the second element into the array cell; indirect addressing
13 #-----
14 lw $t2, 0($t0)            # get the first element from the array cell
15 move $a0, $t2             # $a0 must hold address of integer to print
16 li $v0, 1                 # load syscall into syscall register; $v0 must hold integer command (1)
17 syscall
18
19 lw $t3, 4($t0)            # get the second element from the array cell
20 move $a0, $t3             # $a0 must hold address of integer to print
21 li $v0, 1                 # load syscall into syscall register; $v0 must hold integer command (1)
22 syscall
23 #-----
24 li $v0, 10                # exit to OS
25 syscall
```


Assemble ... GO

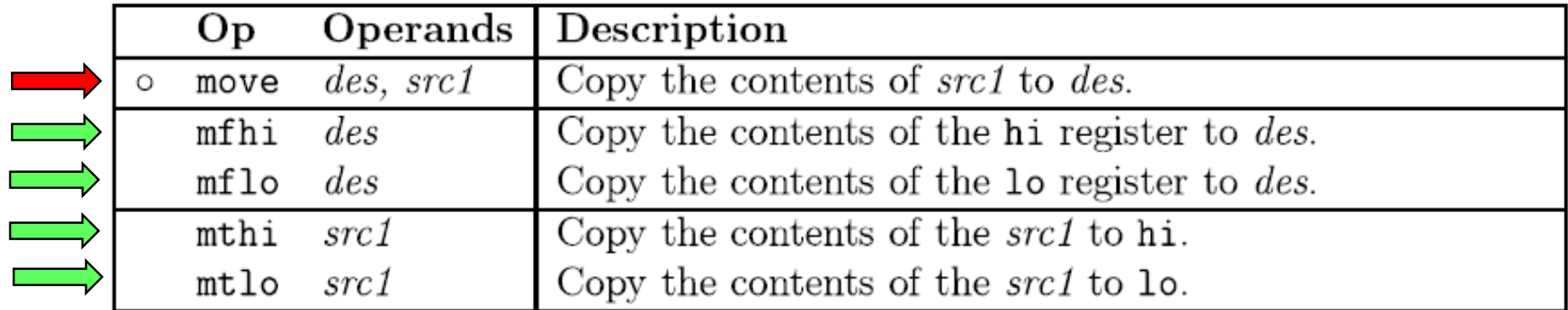
```
3      .data
4  Array: .space 8
5      .text
6      #-----
7      la $t0, Array
8      #-----
9      li $t1, 5
10     sw $t1, 0($t0)
11     li $t1, 6
12     sw $t1, 4($t0)
13     #-----
14     lw $t2, 0($t0)
15     move $a0, $t2
16     li $v0, 1
17     syscall
18
19     lw $t3, 4($t0)
20     move $a0, $t3
21     li $v0, 1
22     syscall
23     #-----
24     li $v0, 10
25     syscall
26
```

Print to console

```
56
-- program is finished running --
```

Registers		
Name	Number	Value
\$zero	0	0
\$at	1	268500992
\$v0	2	10
\$v1	3	0
\$a0	4	6
\$a1	5	0
\$a2	6	0
\$a3	7	0
\$t0	8	268500992
\$t1	9	6
\$t2	10	5
\$t3	11	6
\$t4	12	0
\$t5	13	0
\$t6	14	0
\$t7	15	0
\$s0	16	0
\$s1	17	0
\$s2	18	0
\$s3	19	0
\$s4	20	0
\$s5	21	0
\$s6	22	0
\$s7	23	0
\$t8	24	0
\$t9	25	0
\$k0	26	0
\$k1	27	0
\$gp	28	268468224
\$sp	29	2147479548
\$fp	30	0
\$ra	31	0
pc		4194368
hi		0
lo		0

Data Movement





Op	Operands	Description
○ move	<i>des, src1</i>	Copy the contents of <i>src1</i> to <i>des</i> .
mfhi	<i>des</i>	Copy the contents of the hi register to <i>des</i> .
mflo	<i>des</i>	Copy the contents of the lo register to <i>des</i> .
mthi	<i>src1</i>	Copy the contents of the <i>src1</i> to hi.
mtlo	<i>src1</i>	Copy the contents of the <i>src1</i> to lo.

`mfhi/mflo` → move from hi/lo

`mthi/mtlo` → move to hi/lo

Exception Handling



Op	Operands	Description
rfe		Return from exception.
syscall		Makes a system call. See 4.6.1 for a list of the SPIM system calls.
break	<i>const</i>	Used by the debugger.
nop		An instruction which has no effect (other than taking a cycle to execute).

nop are used to overcome data-hazards in MIPS pipelined-processors



Valid Class Project