Assembly Language and Computer Architecture Lab

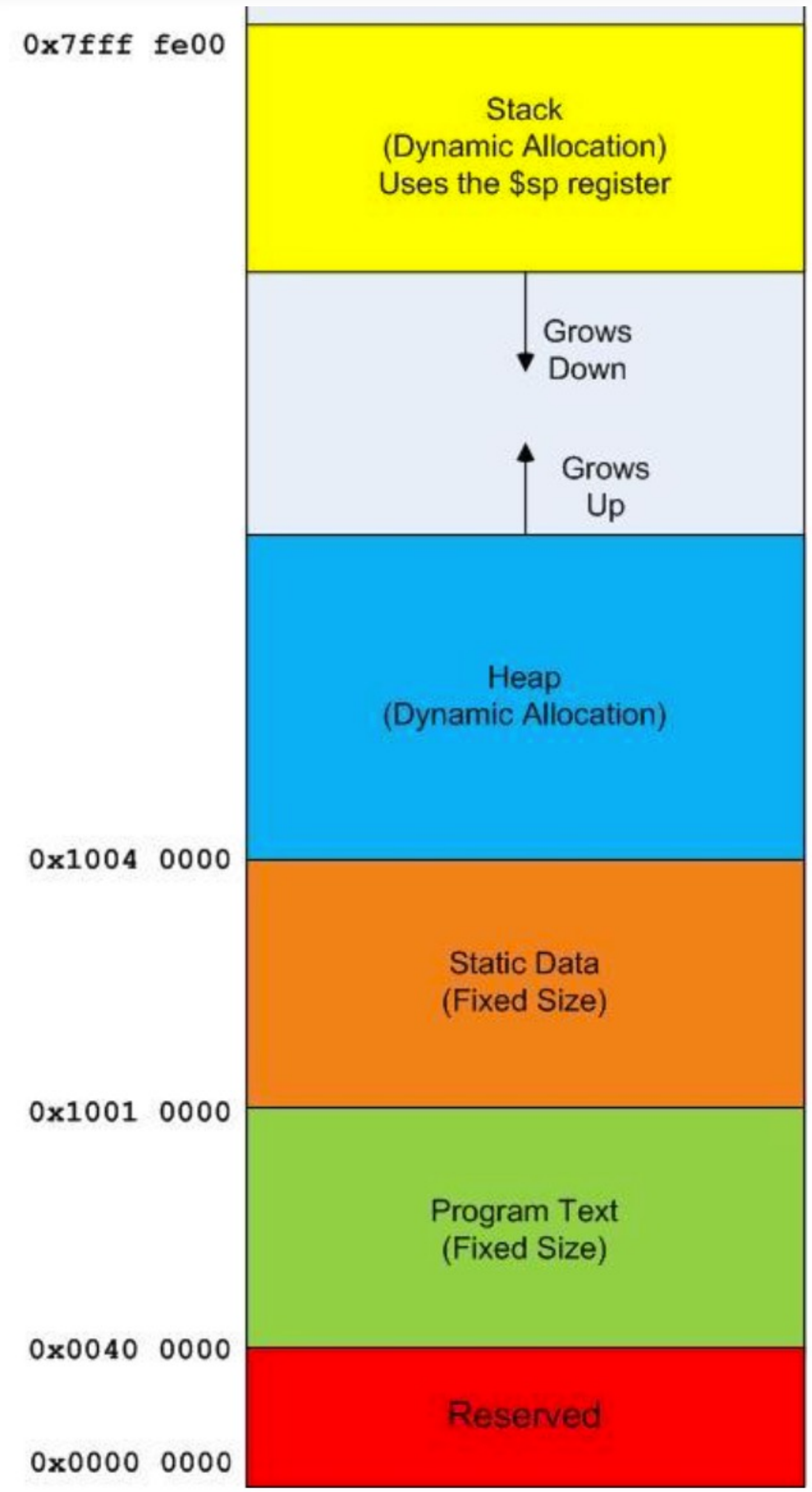
Nguyen Thi Thanh Nga
Dept. Computer engineering
School of Information and Communication Technology

Week 8 Static memory

- Data segment
- Flat memory model
- The static memory
- Accessing memory
- Methods of accessing memory

MIPS memory - the data segment

- 3 main types of memory:
  - static memory
  - stack dynamic memory
  - heap dynamic memory
- Static memory is the simplest as it is defined when the program is assembled and allocated when the program begins execution.
- Dynamic memory is allocated while the program is running and accessed by address offsets. This makes dynamic memory more difficult to access in a program, but much more useful.



Flat memory model

- To a MIPS programmer, memory appears to be flat; there is no structure to it.

- Memory consists of one byte (8 bits) stored after another, and all bytes are equal.

- The MIPS programmer sees a memory where the bytes are stored as one big array, and the index to the array being the byte address.

- The memory is addressable to each byte, and thus called *byte addressable*.

Flat memory model

The bytes in MIPS are organized in groups of:

- A single **byte**
- A group of 2 bytes, called a **half-word**
- A group of 4 bytes, called a **word**
- A group of 8 bytes, called a **double word**

Flat memory model

All groupings start at 0x10010000 and then occur at regular intervals.

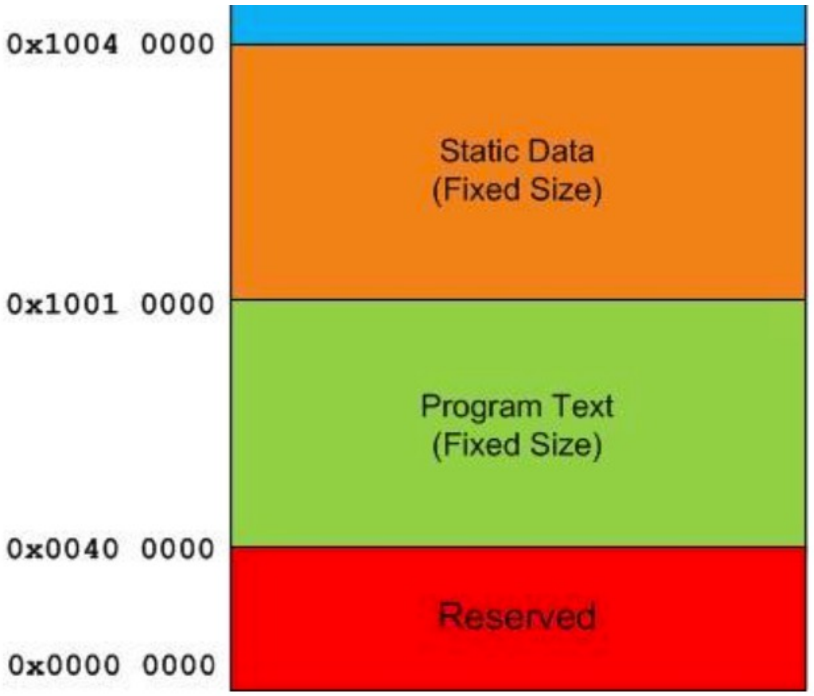
- Memory **half words** would start at addresses 0x10010000, 0x10010002, 0x10010004 and continue in that manner.
- Memory **words** would start at addresses 0x10010000, 0x10010004, 0x10010008, 0x1001000c, and likewise continue.
- Memory **double words** would start at addresses 0x10010000, 0x10010008, 0x10010010, 0x10010018, and continue.

Flat memory model

- The memory groups start is called a ***boundary***.
- Cannot address a group of data except at the boundary for that type.
- For example, a word of memory cannot be loaded at the address 0x10010002 because it is not on a word boundary.
- When discussing data, a word of memory is 4 bytes large (32 bits), but it is also located on a word boundary.
- If 32 bits are not aligned on a word boundary, it is incorrect to refer to it as a word.

Static data

- Static data is data that is defined when the program is assembled and allocated when the program starts to run.
- The size and location of static data is fixed and cannot be changed.
- If a static array is allocated with 10 members, it cannot be resized to have 20 members.
- All variables which will be defined as static must be known before the program is run.



Static data

- Static data is defined using the **.data** assembler directive.

- All memory allocated in the program in a **.data** segment is static data.

- The static data (or simply data) segment of memory is the portion of memory starting at address 0x10010000 and continuing until address 0x10040000.

- The data values can change during the program execution, the data size and address does not change.

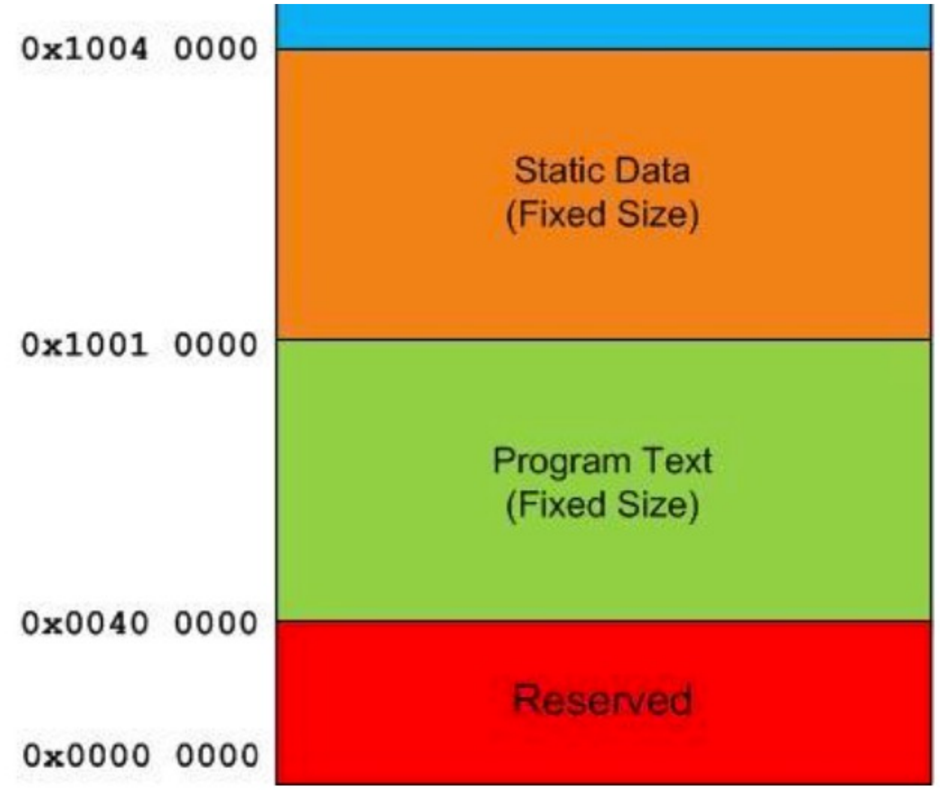
Static data

- When the assembler starts to execute, it keeps track of the next address available in the data segment.

- Initially the value of the next available slot in the data segment is set to 0x10010000.

- As space is allocated in the data segment, the next available slot is incremented by the amount of space requested.

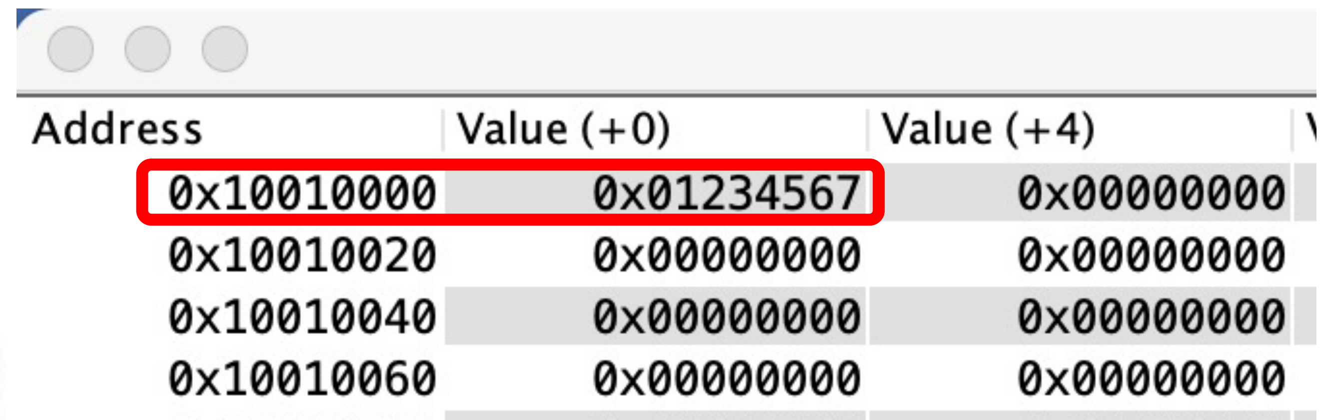
- This allows the assembler to keep track of where to store the next data item.



Static data

- Consider the following MIPS code fragment.

```
.data
a: .word   0x1234567
   .space 14
b: .word   0xFEDCBA98
```



- If this is the first **.data** directive found, the address to start placing data is 0x10010000.

- A word is 4 bytes of memory, so the label *a:* points to a 4 bytes allocation of memory at address 0x10010000 and extending to 0x10010003, and the next free address in the data segment is updated to be 0x10010004.

Static data

- Consider the following MIPS code fragment.

```
.data
a:  .word   0x1234567
    .space  14
b:  .word   0xFEDCBA98
```

Data Segment			
Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x00000000	0x00000000	0x00000000	0x0000000
0x00000000	0x00000000	0x00000000	0x0000000
0x00000000	0x00000000	0x00000000	0x0000000
0x00000000	0x00000000	0x00000000	0x0000000

- Next an area of memory is allocated that using the **.space 14** assembly directive. The **.space** directive sets aside 14 bytes of memory, starting at 0x10010004 and extending to 0x10010011.
- There is no label on this part of the data segment, which means that the programmer must access it directly through an address.
- Generally, there will be a label present for variables in the data segment.

Static data

- Consider the following MIPS code fragment.

```
.data
a:  .word   0x1234567
    .space  14
b:  .word   0xFEDCBA98
```

(+10)	Value (+14)	Value (+18)
0x00000000	0xfedcba98	0x00000000
0x00000000	0x00000000	0x00000000
0x00000000	0x00000000	0x00000000
0x00000000	0x00000000	0x00000000

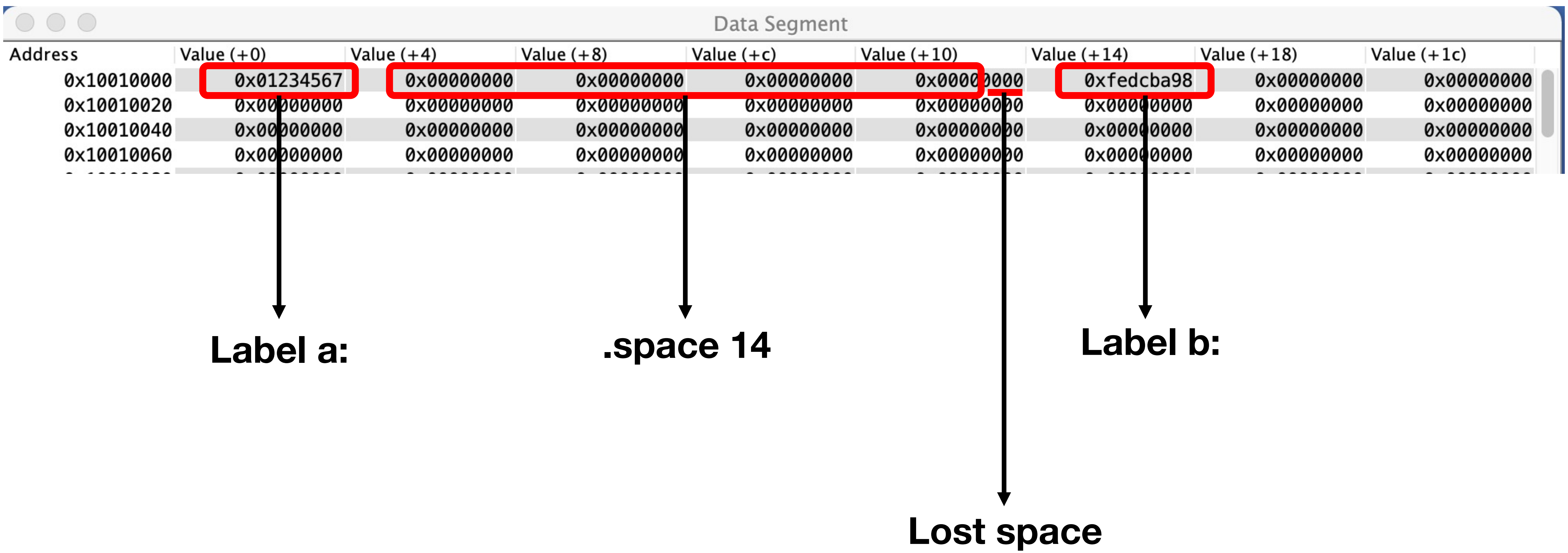
- Finally, another word of memory is allocated at the label **b:**

- This memory could have been placed at 0x10010012, as this is the next available byte. However, specifying that this data item is a word means that it must be placed on a word boundary.

- If the next available address is not on a word boundary when a word allocation is asked for, the assembler moves to the next word boundary, and the space between is simply lost to the program.

Static data

- What the memory looks like after assembling this code fragment.



Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x01234567	0x00000000	0x00000000	0x00000000	0x00000000	0xfedcba98	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Static data

- The column address gives the base address for a grouping of 32 (0x20) byte addresses.

- Each subsequent column is the 4 bytes (or word) offset from the base address.

- The first column is the base address + 0 bytes, so it is addresses 0x10010000 - 0x10010003, the second column is addresses 0x10010004 - 0x10010007, and so on.

- The memory at label **a:** stores 0x01234567, then 14 bytes of uninitialized memory are allocated, the next two bytes of memory are unused and lost, and finally a word at label **b:** which stores 0xfedcba98.

Accessing memory

- All memory access in MIPS in done through some forms of a **load** or **store** operator.
- These operators include loading/storing a byte (**lb**, **sb**); half word (**lh**, **sh**); word (**lw**, **sw**); or double word (**ld**, **sd**).
- In this content, only words of memory will be considered, so only the **lw** and **sw** will be introduced.

lw operator

- The only real format: an address is stored in $\mathbf{R_s}$, and an offset from that address is stored in the **Immediate** value. The value of memory at [$\mathbf{R_s}$ + **Immediate**] is stored into $\mathbf{R_t}$.

- The format and meaning are:

format: **lw $\mathbf{R_t}$, Immediate ($\mathbf{R_s}$)**

meaning: $\mathbf{R_t} \leftarrow$ **Memory** [$\mathbf{R_s}$ + **Immediate**]

Copy from memory to register

lw operator

- The pseudo-operator, allows the address of a label to be stored in $\mathbf{R_s}$ and then the real **lw** operator is called to load the value.
- The format and meaning are:

format: **lw $\mathbf{R_s}$, label**

meaning: $\mathbf{R_s} \leftarrow$ **Memory[Label]**

translation: **lui $at 0x00001001**
lw $\mathbf{R_s}$, offset($at)

offset is displacement of value

in the data segement

lw operator

- Consider the following example:

```
.data
a: .word   0x1234567
   .space 14
b: .word   0xFEDCBA98

.text
lw $s0,b
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$t7	15	0x00000000
$s0	16	0xfedcba98

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,0x00001001	7: lw $s0,b
	0x00400004	0x8c300014	lw $16,0x00000014($1)	

sw operator

- The only real format: an address is stored in $\mathbf{R_s}$, and an offset from that address is stored in the **Immediate** value. The value of $\mathbf{R_t}$ is stored in memory at [$\mathbf{R_s}$ **+ Immediate**].

- The format and meaning are:

format: **sw** $\mathbf{R_t}$, **Immediate (**$\mathbf{R_s}$**)**

meaning: **Memory** [$\mathbf{R_s}$ + **Immediate**] $\leftarrow \mathbf{R_t}$

Copy value from register to memory

sw operator

- The pseudo-operator, allows the content in register $\mathbf{R_s}$ to be stored in the address of the label.
- The format and meaning are:

format: **sw R_s, label**

meaning: **$R_s \leftarrow$ Memory[Label]**

translation: **lui $at 0x00001001**
sw R_s, offset($at)

\# offset is displacement of value

\# in the data segement

sw operator

- Consider the following example:

```
.data
a: .word   0x1234567
   .space 14
b: .word   0xFEDCBA98

.text
lw $s0,b
sw $s0,a
```

$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xfedcba98
$s1	17	0x00000000

Address	Value (+0)	Value (+4)
0x10010000	0x01234567	0x00000000
0x10010020	0x00000000	0x00000000

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,0x00001001	7: lw $s0,b
	0x00400004	0x8c300014	lw $16,0x00000014($1)	
	0x00400008	0x3c011001	lui $1,0x00001001	8: sw $s0,a
	0x0040000c	0xac300000	sw $16,0x00000000($1)	

sw operator

- Consider the following example:

```
.data
a: .word   0x1234567
   .space 14
b: .word   0xFEDCBA98

.text
lw $s0,b
sw $s0,a
```

$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xfedcba98
$s1	17	0x00000000

Address	Value (+0)	Value (+4)
0x10010000	0xfedcba98	0x00000000
0x10010020	0x00000000	0x00000000

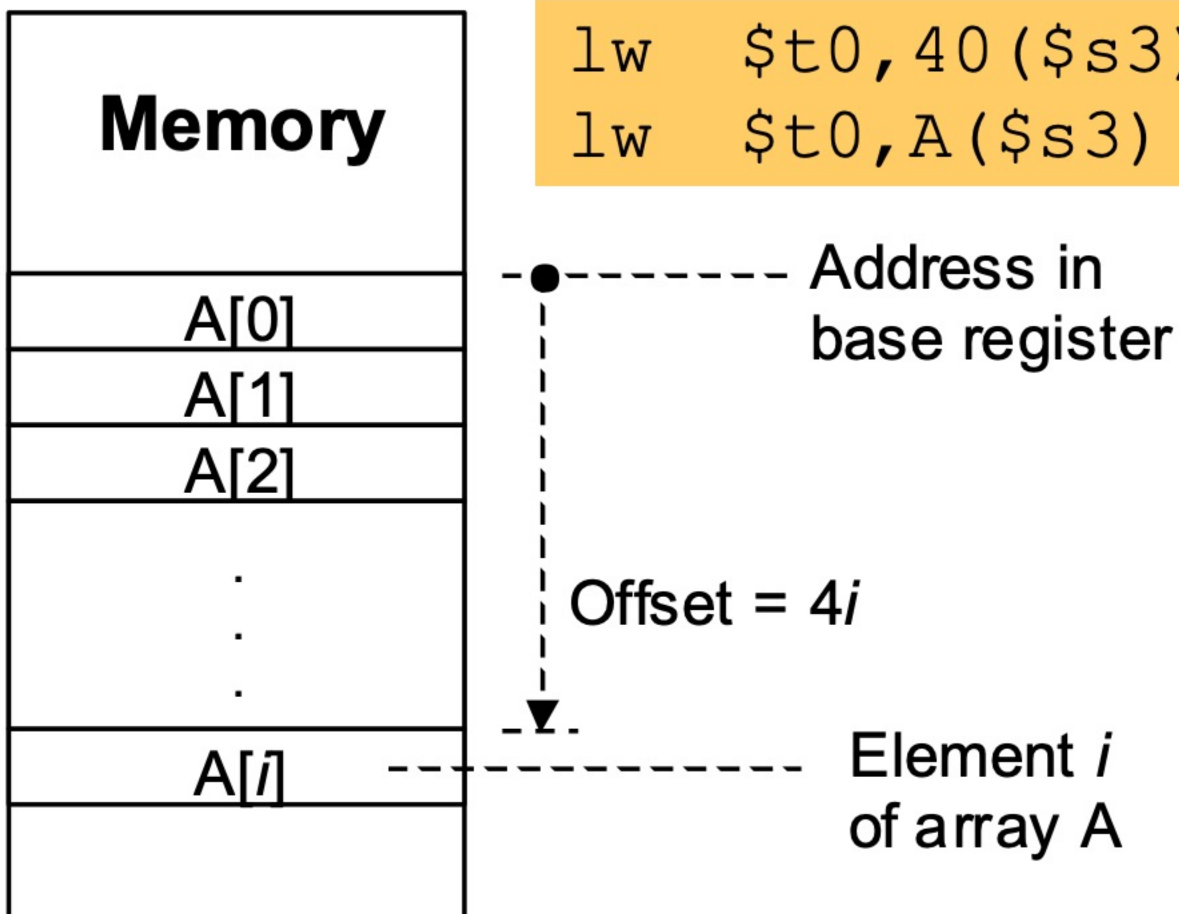
Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,0x00001001	7: lw $s0,b
	0x00400004	0x8c300014	lw $16,0x00000014($1)	
	0x00400008	0x3c011001	lui $1,0x00001001	8: sw $s0,a
	0x0040000c	0xac300000	sw $16,0x00000000($1)	

load and store instructions

	op (31–26)	rs (25–21)	rt (20–16)	operand / offset (15–0)
I	1 0 x 0 1 1	1 0 0 1 1	0 1 0 0 0	0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0
	lw = 35 sw = 43	Base register	Data register	Offset relative to base

```
lw   $t0,40($s3)
lw   $t0,A($s3)
```



Note on base and offset:

The memory address is the sum of (`rs`) and an immediate value. Calling one of these the base and the other the offset is quite arbitrary. It would make perfect sense to interpret the address `A($s3)` as having the base `A` and the offset `($s3)`. However, a 16-bit base confines us to a small portion of memory space.

lw and **sw** in MiniMIPS and memory addressing mechanism allow simple access to the elements of the string over the base address and offset (offset = 4 / i.e., to the ith element (word)).

load, store machine code

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

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

35	$s3	$t0	32
35	19	8	32
100011	10011	01000	0000 0000 0010 0000

(0x8E680020)

```
sw $s1, 4($t1)
```

43	$t1	$s1	4
43	9	17	4
101011	01001	10001	0000 0000 0000 0100

(0xAD310004)

Methods of accessing memory

Four methods of addressing data are shown below:

- Addressing by label
- Register direct access
- Register indirect access
- Register offset access

Methods of accessing memory

- Consider the following pseudo code:

```
main
{
        static volatile int a = 5;
        static volatile int b = 2;
        static volatile int c = 3;
        int x = prompt("Enter a value for x: ");
        int y = a * x * x + b * x + c;
        print("The result is: " + y);
}
```

Quadratic program pseudo code

Addressing by label

- A label can be defined for the address of a variable.

- This type of data can only exist in the **.data** segment of the program, which means that this data cannot move or change size.

- When the variable is stored in the **.data** segment, it can generally be addressed directly using a label.

Addressing by label

```
#Program 8.1 Quadratic program
#Date 2/4/2020
#Purpose: Addressing by label
.text
.globl main

        # Get input value and store it in $s0
main:   la $a0, prompt
        jal PromptInt
        move $s0, $v0

        # Load constants a, b, and c into registers
        lw $t5, a
        lw $t6, b
        lw $t7, c

        # Calculate the result of y=a*x*x + b*x + c
        #and store it
        mul $t0, $s0, $s0
        mul $t0, $t0, $t5
        mul $t1, $s0, $t6
        add $t0, $t0, $t1
        add $s1, $t0, $t7

        # Store the result from $s1 to y
        sw $s1, y

        # Print output from memory y
        la $a0, result
        lw $a1, y
        jal PrintInt
        jal PrintNewLine
        nop

        #Exit program
        jal Exit
.data
    a: .word 5
    b: .word 2
    c: .word 3
    y: .word 0
    prompt: .asciiz "Enter a value for x: "
    result: .asciiz "The result is: "

.include "utils.asm"
```

a, b, and c are loaded from memory using the lw operator with labels.

Register direct access

```
#Program 8.2 Quadratic program
#Date 2/4/2020
#Purpose: register direct access
.text
.globl main
        # Get input value and store it in $s0
main:   la $a0, prompt
        jal PromptInt
        move $s0, $v0

        # Load constants a, b, and
        li $t5, 5
        li $t6, 2
        li $t7, 3

        # Calculate the result of
        # and store it
        mul $t0, $s0, $s0
        mul $t0, $t0, $t5
        mul $t1, $s0, $t6
        add $t0, $t0, $t1
        add $s1, $t0, $t7
        # Print output from memory y
        la $a0, result
        move $a1, $s1
        jal PrintInt
        jal PrintNewLine
        nop

        #Exit program
        jal Exit
.data
    y: .word 0
    prompt: .asciiz "Enter a value for x: "
    result: .asciiz "The result is: "

.include "utils.asm"
```

The values are stored directly in the registers, and so memory is not accessed at all.

Register indirect access

- Register indirect access differs from register direct access in that the register does not contain the value to use in the calculation but **contains the address** in memory of the value to be used.

- For example:

```
.data
    .word 5
    .word 2
    .word 3
y:  .word 0
```

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x10010000	0x00000005	0x00000002	0x00000003	0x00000000	0x65746e45

Register indirect access

```
#Program 8.3 Quadratic program
#Date 2/4/2020
#Purpose: register indirect access
.text
.globl main

        # Get input value and store it in $s0
main:   la $a0, prompt
        jal PromptInt
        move $s0, $v0

        # Load constants a, b, and c into reg
        lui $t0, 0x1001
        lw $t5, 0($t0)
        addi $t0, $t0, 4
        lw $t6, 0($t0)
        addi $t0, $t0, 4
        lw $t7, 0($t0)

        # Calculate the result of y=a*x*x + b
        # and store it.
        mul $t0, $s0, $s0
        mul $t0, $t0, $t5
        mul $t1, $s0, $t6
        add $t0, $t0, $t1
        add $s1, $t0, $t7

        # Print output from memory y
        la $a0, result
        move $a1, $s1
        jal PrintInt
        jal PrintNewLine

        #Exit program
        jal Exit

.data
    .word 5
    .word 2
    .word 3
y:  .word 0
prompt: .asciiz "Enter a value for x: "
result: .asciiz "The result is: "

.include "utils.asm"
```

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x10010000	0x00000005	0x00000002	0x00000003	0x00000000	0x65746e45

Register offset access

- In the **lw** instruction, the immediate value is a distance from the address in the register to the value to be loaded.

- In the register indirect access, this immediate was always zero as the register contained the actual address of the memory value to be loaded.

- In the following example, the value will be used to specify how far in memory the value to be loaded is from the address in the register.

Register offset access

```
#Program 8.4 Quadratic program
#Date 2/4/2020
#Purpose: register offset address
.text
      .globl main

        # Get input value and store
main:   la $a0, prompt
        jal PromptInt
        move $s0, $v0

        # Load constants a, b, and
        lui $t0, 0x1001
        lw $t5, 0($t0)
        lw $t6, 4($t0)
        lw $t7, 8($t0)

        # Calculate the result of y
        # and store it
        mul $t0, $s0, $s0
        mul $t0, $t0, $t5
        mul $t1, $s0, $t6
        add $t0, $t0, $t1
        add $s1, $t0, $t7
        # Print output from memory y
        la $a0, result
        move $a1, $s1
        jal PrintInt
        jal PrintNewLine

        #Exit program
        jal Exit
.data
    .word 5
    .word 2
    .word 3
y: .word 0
prompt: .asciiz "Enter a value for x: "
result: .asciiz "The result is: "

.include "utils.asm"
```

Pointer

- If a register can contain the address of a variable in memory, then a memory value can contain a reference to another variable at another spot in memory.

- These variables are called pointer variables.

- The following program shows the use of memory indirect (pointer) variables.

Pointer

The memory at the start of the .data segment contains an address to the actual storage location for the constants a, b, and c.

```
#Program 8.5 Quadratic program
#Date 2/4/2020
#Purpose: memory indirect (pointer)
.text
      .globl main

        # Get input value and store
main:   la $a0, prompt
        jal PromptInt
        nop
        move $s0, $v0

        # Load constants a, b, an
        lui $t0, 0x1001
        lw $t0, 0($t0)
        lw $t5, 0($t0)
        lw $t6, 4($t0)
        lw $t7, 8($t0)

        # Calculate the result of
        #y=a*x*x + b * x + c and
        mul $t0, $s0, $s0
        mul $t0, $t0, $t5
        mul $t1, $s0, $t6
        add $t0, $t0, $t1
        add $s1, $t0, $t7
```

```
        # Print output from memory y
        la $a0, result
        move $a1, $s1
        jal PrintInt
        jal PrintNewLine
        nop

        #Exit program
        jal Exit
.data
   .word constants
y: .word 0
prompt: .asciiz "Enter a value for x: "
result: .asciiz "The result is: "
constants:
        .word 5
        .word 2
        .word 3
.include "utils.asm"
```

Pointer

- The memory at the start of the .data segment contains an address to the actual storage location for the constants a, b, and c.
- These variables are then accessed by loading the address in memory into a register, and using that address to locate the constants.

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)
0x10010000	0x10010030	0x00000000	0x65746e45	0x20612072	0x756c6176	0x6f662065	0x3a782072
0x10010020	0x65722065	0x746c7573	0x3a736920	0x00000020	0x00000005	0x00000002	0x00000003

Exercise 8.1

- The following table has memory addresses in each row, and columns which represent each of the MIPS boundary types, byte, half word, word, and double word.

- Put a check mark in the column if the address for that row falls on the boundary type for the column.

Exercise 8.1

Address	Boundary Type			
	Byte	Half	Word	Double
0x10010011				
0x10010100				
0x10050108				
0x1005010c				
0x1005010d				
0x1005010e				
0x1005010f				
0x10070104				

Exercise 8.2

- Why do "**la label**" instructions always need to be translated into 2 lines of pseudo code?
- What about "**lw label**" instructions?
- Explain the similarities and differences in how they are implemented in MARS.

Exercise 8.3

- The following program fails to load the value 8 into $t0. In fact it creates an exception. Why?

```
.text
        lui $t0, 1001
        lw $a0, 0($t0)
        li $v0, 1
        syscall

        li $v0, 10
        syscall

.data
        .word 8
```

Exercise 8.4

- Translate the following pseudo code into MIPS assembly to show each of the addressing modes covered in this chapter.
- Note that variables x and y are static and volatile, so should be stored in data memory. When using register direct access, you do not need to store the variables in memory.

```
main() {
    static volatile int miles =
        prompt("Enter the number of miles driven: ");
    static volatile int gallons =
        prompt("Enter the number of gallons used: ");
    static volatile int mpg = miles / gallons;
    output("Your mpg = " + mpg);
}
```

End of week 8