Chapter 3:

Pointers and arrays

Topics:

Layout of code and data in memory
Loads and stores
Registers as pointers
Translating code with arrays
Programming with strings

Reading: Patterson and Hennessy
2.7, 2.14

Layout of Code and Data in Memory

So far, all variables are in registers.
But number of registers is limited!
Too many variables: *spill* to memory.
[Arrays, more complex objects also in memory.]

C++ source code:

```
int x=4;
int y=-1;

int main() {
   // code not shown
}
```

MIPS source code:

```
      .data
x:    .word 4         # int x=4;
y:    .word -1        # int y=-1;

      .text
main: # code not shown
```

x and main are labels; they mark locations in memory.

Layout in memory depends on compiler. For spim:

> main is usually at 0x400024
>
> first address in .data is at 0x10010000

main:	0x400024	# 1^{st} instruction of program
	0x400028	# 2^{nd} instruction of program

X:	0x10010000	0x00000004
y:	0x10010004	0xffffffff

.data, .text, .word are *assembler directives*, not MIPS instructions.
They tell the (spim) assembler to do specific things (manage memory layout), but the MIPS CPU doesn't execute directives.

Def: .word *allocates* a 32-bit word in memory
No type! (Type is managed by the compiler/assembler, or the programmer.)

Def: .data means data allocation section follows

Def: .text means program text follows
(following section is read-only)

Remember: MIPS arithmetic instructions have register operands only.
To work with data in memory, need:

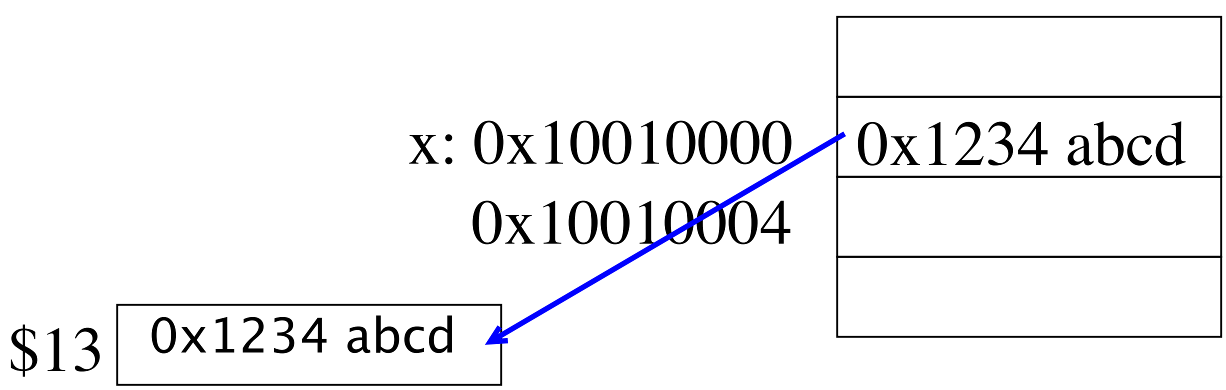
Load instructions	register <- memory
Store instructions	memory <- register

Def: load word (from memory to register)

lw R, ??
[R is any register
?? indicates a memory address ADDR]

contents of R
= contents of aligned word at ADDR (in memory)

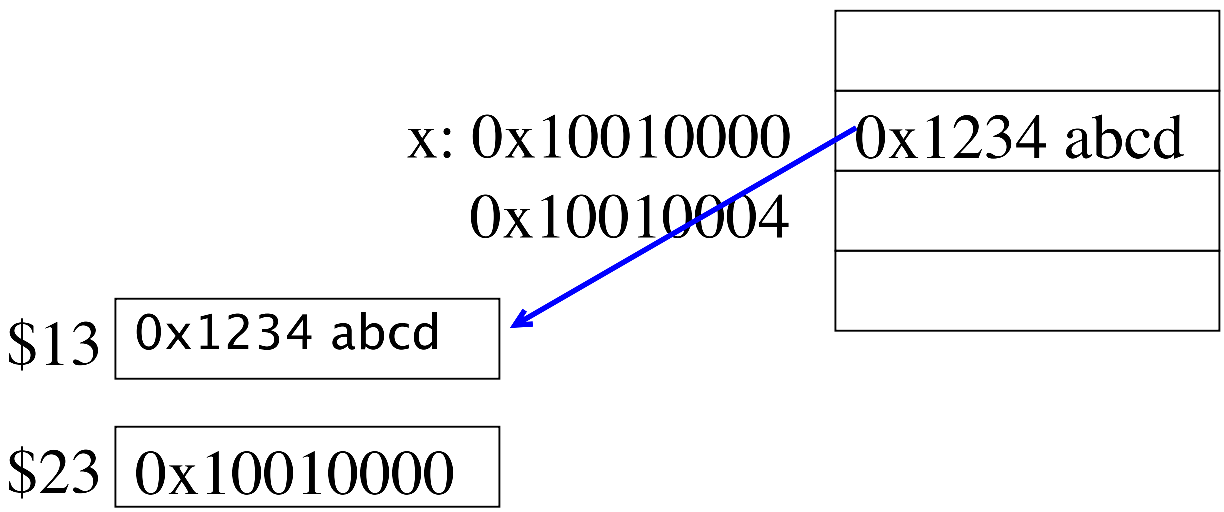
Case 1: ?? is a label
lw $13, x



Case 2: ?? is (Rb)
[Rb is any register]

lw R, (Rb)
ADDR is contents of Rb, i.e.,
R = M[contents of Rb]

Example: lw $13, ($23)
(operation: $13 = M[$23])

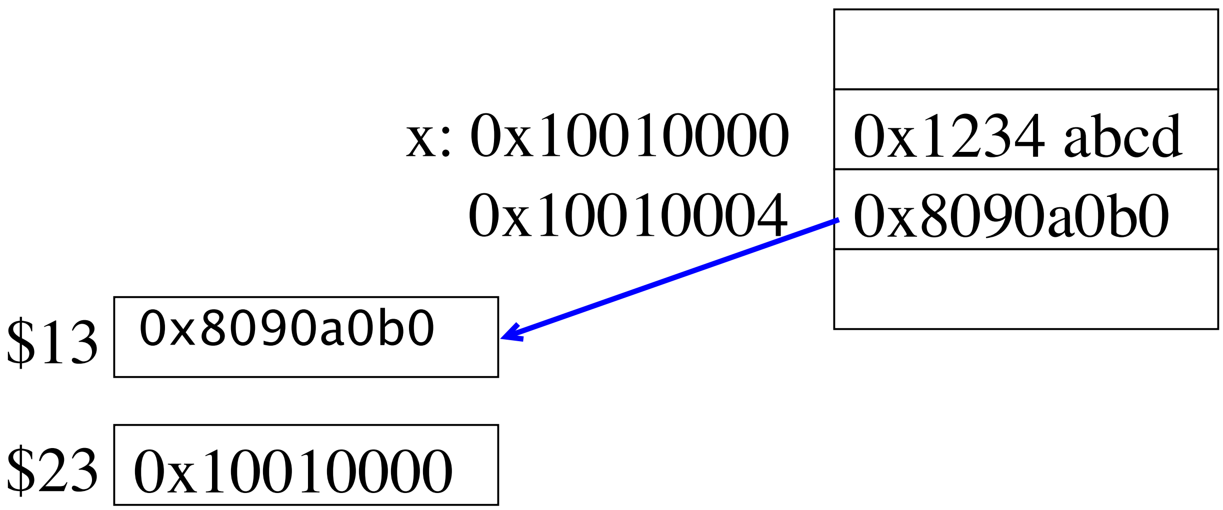


Observation: Rb is reference or pointer

Case 3: ?? is K(Rb)
[Rb is any register, K is constant]

lw R, K(Rb)
ADDR = contents of Rb + K, or
R = M[contents of Rb + K]

Example: lw $13, 4($23)
(operation: $13 = M[4 + $23]
ADDR = 4 + 0x10010000 = 0x10010004



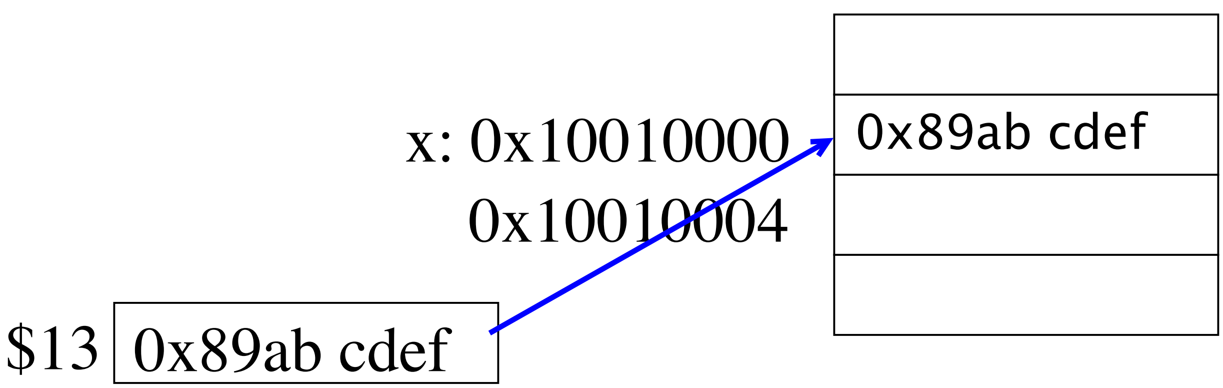
This mode is used in: objects, structs

stack frames (method/function calls)

***Recall**: lw R, ??   # R = M[??]*
sw R, ??   # M[??] = R
[R is any register
?? indicates a memory address ADDR]

contents of aligned word at ADDR (in memory)
= contents of R

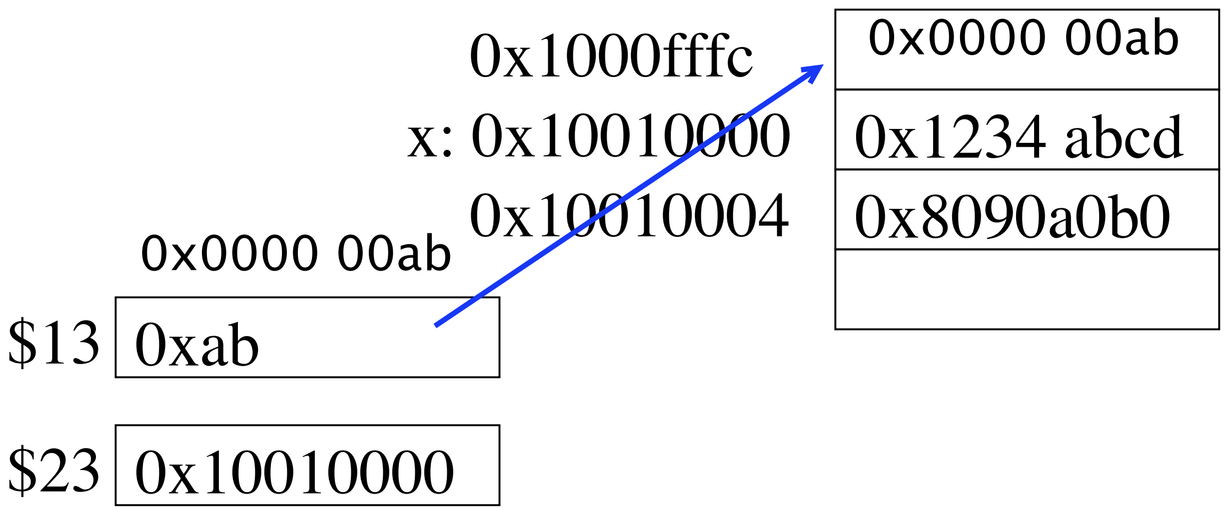
Case 1: ?? is a label
sw $13, x



The 3 ways of specifying the address can also be applied to sw, and other load and store instructions.

Example: sw $13, -4($23)

ADDR = -4 + 0x10010000 = 0x1000fffc



More load and store instructions

load byte: lb R, ??
(R is any register, ?? specifies ADDR, see 3 main options for ??)

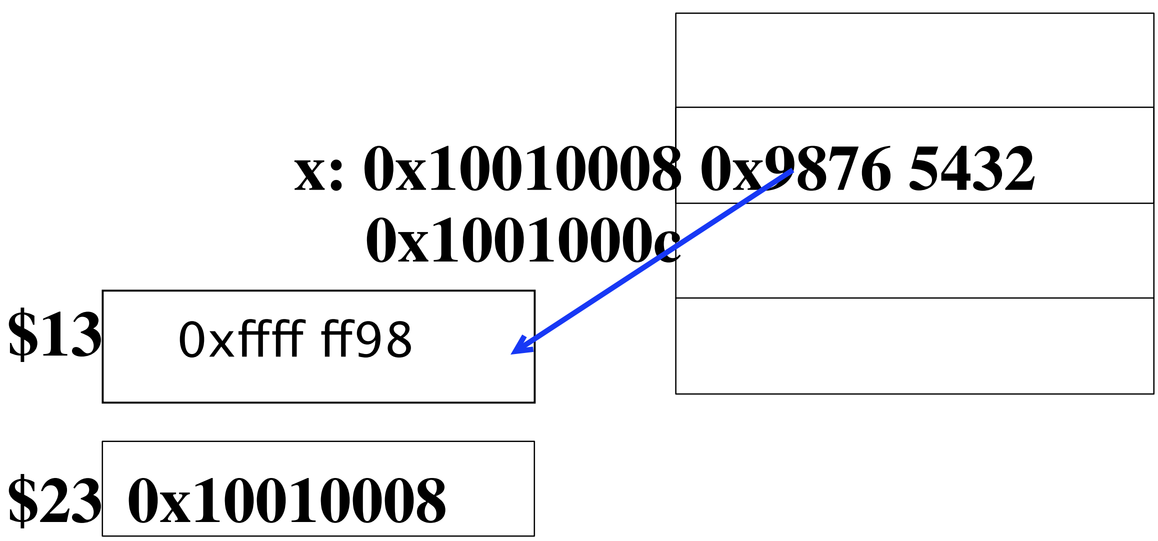
low 8 bits (bits 7-0) of R
= contents of byte from memory at ADDR
other bits of R = sign bit of byte from memory

bit 7 of byte at ADDR, 24 times byte at ADDR

(or, R = $(m[ADDR]_7)^{24} \,||\, m[ADDR]$)

concatenate

Example: lb \$13, (\$23)

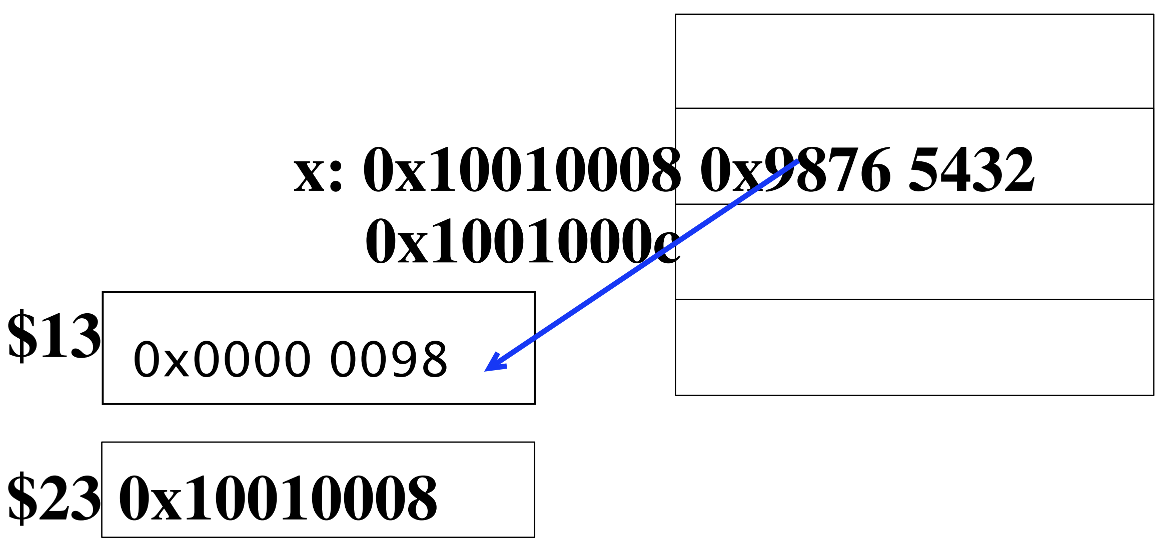


load byte unsigned: lbu R, ??
(R is any register, ?? specifies ADDR, see 3 main options for ??)

low 8 bits (bits 7-0) of R
= contents of byte from memory at ADDR
other bits of R = 0's

(or, R = 0^{24} || m[ADDR])

Example: lbu $13, ($23)

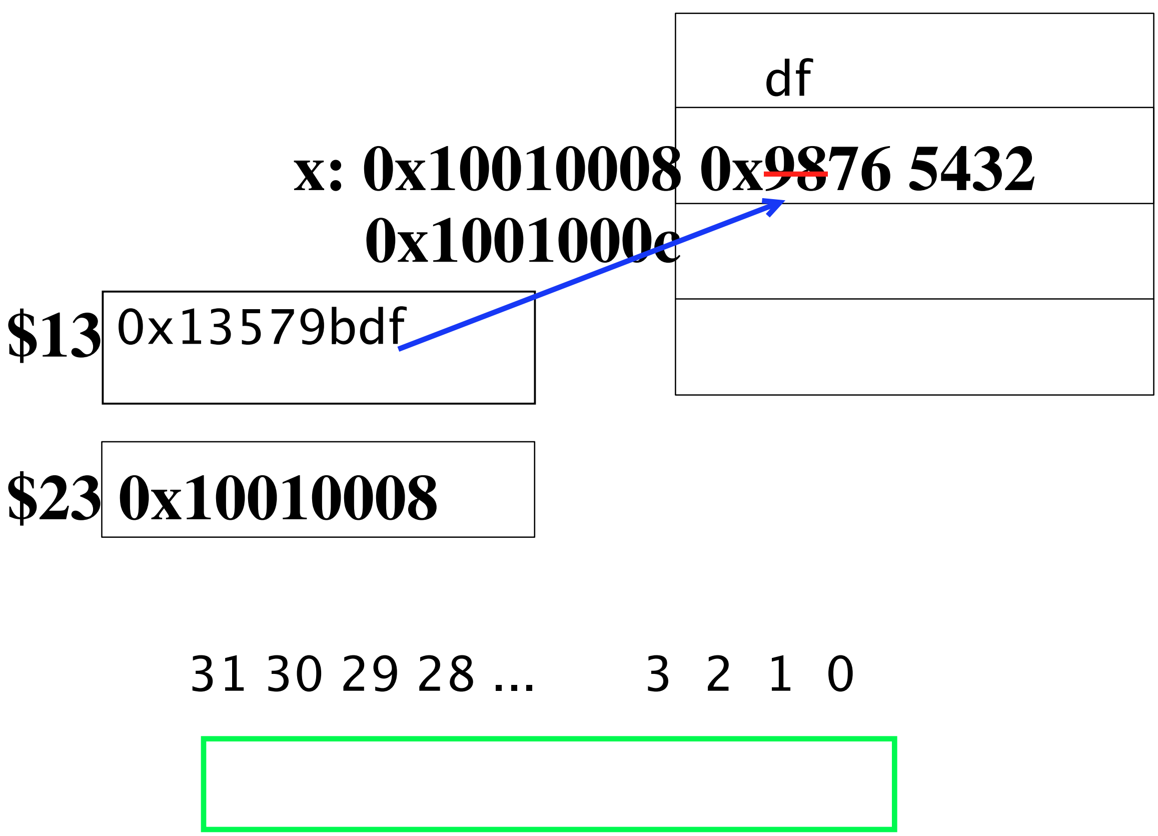


store byte: sb R, ??
(R is any register, ?? specifies ADDR, see 3 main options for ??)

contents of byte from memory at ADDR
= bits 7-0 of R

(or, $m[ADDR] = [R]_{7..0}$

Example: sb $13, ($23)



Pointers

A pointer is a variable that contains the address of another variable (in memory).

(Since addresses are integers, pointers "look" like integers.)

Example:

x is an integer variable (in memory).
ptr_x is a pointer to an integer.
$s0 is also a pointer to an integer.

$s0	**0x1001 0000**

x: 0x1001 0000	**7**
ptr_x: 0x1001 0004	**0x1001 0000**

ptr_x and $s0 both point to x
ptr_x and $s0 contain the address of x

1. Declaring pointers

To declare a pointer called ptr_x of type [type]:

```
[type]  *ptr_x;
```

For example,

```
int *ptr0; // ptr0 is a pointer to an int
char *ptr1; // ptr1 is a pointer to a char
```

2. Initializing pointers

Pointers must be initialized before they are used. We initialize the contents of pointers to the addresses of variables.

&x means "address of variable x"

To initialize ptr_x to point to the variable x (Or, initialize ptr_x to contain the address of the variable x):

```
int  *ptr_x; int  x;      ptr_x = &x;
```

3. Deferencing pointers

To reference a variable that a pointer points to, we dereference the pointer.

*ptr_x means
"the variable that ptr_x points to"
"dereference ptr_x"

Example code (same effect as x=0):

```
int  *ptr_x; int  x;

ptr_x = &x;
*ptr_x = 0;
```

(*ptr_x = 0 means "the variable that ptr_x points to is set equal to 0")

4. Pointer operations

Always think of pointers as variables that contain addresses.

Example:

```
int  *ptr_x, *ptr_y;
int  x=7, y=13;

ptr_x = &x;
ptr_y = &y;
```

Suppose
address of x is 0x1001 0000
address of y is 0x1001 0004

variable	address	contents
x	0x10010000	7
y	0x10010004	13
ptr_x	??	0x10010000
ptr_y	??	0x10010004

Using the same initial conditions as above, mark the changes.

Example 1: ptr_x = &y;
This sets ptr_x to the address of y; now ptr_x points to y, instead of x.

variable	address	contents
x	0x10010000	7
y	0x10010004	13
ptr_x	??	~~0x10010000~~ 0x10010004
ptr_y	??	0x10010004

Example 2: ptr_y = ptr_x;

This sets ptr_y equal to ptr_x; hence, they both point to what ptr_x points to, which is x.

variable	address	contents
x	0x10010000	7
y	0x10010004	13
ptr_x	??	0x10010000
ptr_y	??	~~0x10010004~~ 0x10010000

Example 3: *ptr_x = y;   // x = y;

This sets what ptr_x points to, which is x, to the contents of the variable y.

variable	address	contents
x	0x10010000	~~7~~ 13
y	0x10010004	13
ptr_x	??	0x10010000
ptr_y	??	0x10010004

Example 4: *ptr_x = *ptr_x + *ptr_y;

This sets what ptr_x points to equal to the sum of what ptr_x points to and what ptr_y points to.

variable	address	contents
x	0x10010000	~~7~~ 20
y	0x10010004	13
ptr_x	??	0x10010000
ptr_y	??	0x10010004

Character arrays

```
char str[] = "Gysin";
char *ptr_ch;
```

	8	
str: 0x10010000	**'G'**	**str[0]**
0x10010001	**'y'**	**str[1]**
0x10010002	**'s'**	**str[2]**
0x10010003	**'i'**	**str[3]**
0x10010004	**'n'**	**str[4]**
0x10010005	**'\0'**	**str[5]**

In C/C++, characters are encoded in *ASCII*.
(See Patterson and Hennessy, p. 122)
Each character is 8 bits.
'G' is 0x47 (or 71), 'y' is 0x79 (or 121), etc

Java uses *Unicode*. (See P&H, p. 127)
Each character is 16 bits (2 bytes).
Many alphabets are encoded, each organized into a block.

base address of str[] = &str[0] = str
= 0x10010000

address of str[i] = &str[0] + i
(for char arrays only!)

Using pointers to access char arrays

1) To initialize ptr_ch to point to str[0]:

ptr_ch = &str[0];
(or, ptr_ch = str;)

2) Simple pointer arithmetic

char *ptr_ch;
ptr_ch++ or ptr_ch = ptr_ch + 1
means "add one to ptr_ch so that ptr_ch contains the address of the next char"
(true for char arrays and char pointers only! int arrays and int pointers slightly different)

Example:

```
char str[] = "Gysin";
char *ptr_ch = &str[0];
```

Code	ptr_ch	output
`ptr_ch = &str[0];`	0x1001 0000	
`cout << *ptr_ch;`		'G'
`ptr_ch++;`	0x1001 0001	
`cout << *ptr_ch;`		'y'
`ptr_ch++;`	0x1001 0002	
`cout << *ptr_ch;`		's'

Similarly, ptr_ch = ptr_ch + K means "add K to contents of ptr_ch, so that ptr_ch contains the address of the char that is K chars after the original char that ptr_ch pointed to"

(true for char arrays and char pointers only! int arrays and int pointers slightly different)

Example: what is printed?

Code	ptr_ch	output
`ptr_ch = &str[0];`	0x10010000	
`ptr_ch = ptr_ch + 3;`	0x10010003	
`cout << *ptr_ch;` // print		i
`cout << *(ptr_ch - 1);`		s

Working with char arrays in C/C++ and MIPS:

C/C++ (sequential array access or stepping through an array):

```
  char str[6];

  for (i=0;i<6;i++)
    str[i] = 0xa;
```

Rewrite in MIPS:

* Need to calculate address of str[i]
    use formula: &str[i] = &str[0] + i
* Need instruction to get address of label

MIPS load address instruction: la R, label
means R = address of label

Choose some registers:
i is $i
&str[0] (or address of str) is in $base
$temp is a temporary

Rewrite in MIPS:

```
str:  .byte 0:6
# at label str, allocate 6 bytes
# initialize to 0

       li     $i,0
       la     $base,str
       li     $temp, 0xa
loop:
       add    $t1, $base, $i      # $t1 = &str[i]
       sb     $temp, ( $t1 )      # str[i] = 0xa
       addi   $i, $i, 1
       blt    $i, 6, loop
```

[Example 3.1:
http://unixlab.sfsu.edu/~whsu/csc256/PROGS/3.1]

Trace:

label	address	contents	Array element
str:	0x10010000	0x~~00~~0a	str[0]
	0x10010001	0x~~00~~0a	str[1]
	0x10010002	0x00	str[2]
	0x10010003	0x00	str[3]
	0x100100~~1~~4	0x00	str[4]
	0x100100~~1~~5	0x00	str[5]

$base = 0x1001 0000 $temp = 0xa

	$i	$t1
	0	
add $t1, $base, $i		0x1001 0000
sb $temp, ($t1)		
addi $i, $i, 1	1	
add $t1, $base, $i		0x1001 0001
sb $temp, ($t1)		
addi $i, $i, 1		
	2	

Rewrite C/C++ sequential array access code using pointers:

```
char str[6];
char *ptr;

ptr = str; // ptr = &str[0]

for (i=0;i<6;i++) {
    *ptr = 0xa;
    ptr++;
}
```

Rewrite in MIPS:

Choose some registers
$ptr is ptr
$i is i

```
str:    .byte   0:6

        li      $i,0
        la      $ptr, str
        li      $t0, 0xa
loop:   sb      $t0, ($ptr)     // *ptr = 0xa;
        addi    $ptr, $ptr, 1
        addi    $i, $i, 1
        blt     $i, 6, loop
```

[Example 3.2:
http://unixlab.sfsu.edu/~whsu/csc256/PROGS/3.2]

Trace:

label	address	contents	Array element
str:	0x10010000	0x~~00~~ 0a	str[0]
	0x10010001	0x~~00~~ 0a	str[1]
	0x10010002	0x00	str[2]
	0x10010003	0x00	str[3]
	0x10010014	0x00	str[4]
	0x10010015	0x00	str[5]

$t0 = 0xa

	$ptr	$i
	0x1001 0000	0
sb $t0, ($ptr)		
addi $ptr, $ptr, 1		
	0x1001 0001	
addi $i, $i, 1		
		1
sb $t0, ($ptr)		
addi $ptr, $ptr, 1		
	0x1001 0002	
addi $i, $i, 1		
		2

Example: find length of string
[C:
http://unixlab.sfsu.edu/~whsu/csc256/PROGS/3.3.cpp
MIPS: http://unixlab.sfsu.edu/~whsu/csc256/PROGS/3.3]

```
int main() {
  char str[] = "abcde";
  char *ptr;
  int count = 0;

  ptr = str;
  while (*ptr != 0) {
    count++;
    ptr++;
  }
  cout << count << endl;
}
                    // System.out.println(count);
```

MIPS version (excerpts):

```
# ptr    $s0
# count  $s1

       .text
main:  li  $s1, 0    #  int count = 0;
       la  $s0, str  #  ptr = str;
loop:  lbu $t0, ($s0)
       beq $t0, $0, end
                     #  while (*ptr != 0) {
loop1: addi $s1,$s1,1 #    count++;
       addi $s0,$s0,1 #    ptr++;
       b     loop
       lbu   $t0, ($s0)
       bne   $t0, $0, loop1
end:
```

How to make this more efficient?

Integer arrays

int x[6];
int *ptr;

label	address	contents	Array element
`x:`	`0x10010000`		`x[0]`
	`0x10010004`		`x[1]`
	`0x10010008`		`x[2]`
	`0x1001000c`		`x[3]`
	`0x10010010`		`x[4]`
	`0x10010014`		`x[5]`

base address of x[] = &x[0] =x
= 0x10010000

address of x[i] = &x[0] + i*4
(for 32-bit int arrays only!)

Working with integer arrays in C/C++ and MIPS:

C/C++:

```
int x[6];

for (i=0;i<6;i++)
  x[i] = i;
```

Rewrite in MIPS:

* Need to calculate address of x[i]
  use formula: &x[i] = &x[0] + i*4
* Use la instruction to get address of x

Choose some registers:
i is $i
&x[0] is in $base
$temp is a temporary

MIPS version:

```
x:     .word   0:6
# at label x, allocate 6 words
# initialize to 0

       li      $i,0
       la      $base,x

loop:  mul     $t0, $i, 4
       add     $t0, $base, $t0  # $t0 = &x[i]
       sw      $i, ($t0)        # x[i] = i
       addi    $i, $i, 1
       blt     $i, 6, loop
```

Trace:

label	address	contents	Array element
`x:`	`0x10010000`	~~0~~ 0	`x[0]`
	`0x10010004`	~~0~~ 1	`x[1]`
	`0x10010008`	0	`x[2]`
	`0x1001000c`	0	`x[3]`
	`0x10010010`	0	`x[4]`
	`0x10010014`	0	`x[5]`

$base = 0x1001 0000

	$i	$t0
mul $t0, $i, 4	0	
		0
add $t0, $base, $t0		
		0x1001 0000
sw $i, ($t0)		
addi $i, $I, 1		
	1	
mul $t0, $i, 4		
		4
add $t0, $base, $t0		
		0x1001 0004
sw $i, ($t0)		

Rewrite C/C++ sequential array access code using pointers:

```
int x[6];
int *ptr;

ptr = x;

for (i=0;i<6;i++) {
    *ptr = i;
    ptr++;
}
```

Trace:

label	address	contents	Array element
`x:`	`0x10010000`	~~0~~ 0	`x[0]`
	`0x10010004`	~~0~~ 1	`x[1]`
	`0x10010008`	0	`x[2]`
	`0x1001000c`	0	`x[3]`
	`0x10010010`	0	`x[4]`
	`0x10010014`	0	`x[5]`

	$i	$ptr
	0	0x1001 0000
sw $i, ($ptr)		
addi $ptr, $ptr, 4		
		0x1001 0004
addi $i, $i, 1		
	1	
sw $i, ($ptr)		
addi $ptr, $ptr, 4		
		0x1001 0008
addi $i, $i, 1		
	2	

Rewrite in MIPS:

Choose some registers
$ptr is ptr
$i is i

```
x:      .word   ?:6

        li      $i,0
        la      $ptr, x   # ptr = x;

loop:   sw      $i, ($ptr)    # *ptr = i;
        addi    $ptr, $ptr, 4   # ptr++;
        addi    $i, $i, 1
        blt     $i, 6, loop
```

[C:
http://unixlab.sfsu.edu/~whsu/csc256/PROGS/3.4.cpp
MIPS: http://unixlab.sfsu.edu/~whsu/csc256/PROGS/3.4]

Summary

New MIPS instructions:

lw	load word
sw	store word
lb	load byte
lbu	load byte unsigned
sb	store byte
la	load address

Spim assembler directives:

.data	data allocations follow
.text	program code follows
.word	allocate a word
.byte	allocate a byte

Topics:

MIPS code for random array access
MIPS code for sequential array access
Pointers and arrays