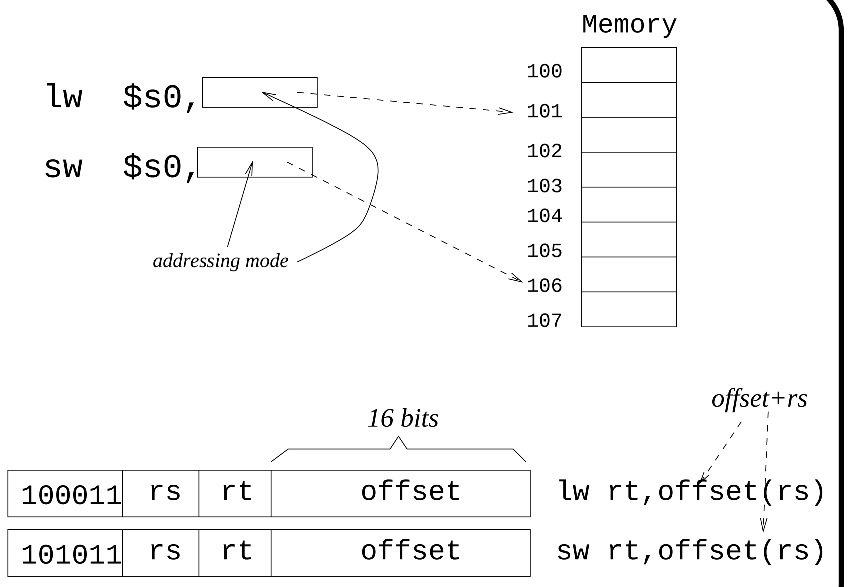
University of
Arizona

CSc 340
Foundations of Computer Systems

Christian Collberg
February 1, 2001

Addressing Modes



100011	rs	rt	offset	lw rt,offset(rs)
101011	rs	rt	offset	sw rt,offset(rs)

MIPS Addressing Modes

- MIPS is a load/store architecture. Only the load & store instructions take a memory address operand.
- MIPS hardware supports a single addressing mode:

  `c(rx)`

  `c` is a 16-bit 2's complement literal integer, `rx` is a register.
- The addressing mode computes ⌜`rx + c`⌝, i.e. the sum of the value in `rx` and `c`.
- This is called register indirect with offset addressing, and can be used to implement pointers.

Pointers

- A pointer is a memory location (variable) that contains the address of another variable.
- Let variable `B` be stored at address 100 and have value 42. Let variable `A` be a pointer to `B` stored at address 108. `A`'s value is 100:

Variable	Address	Memory
	110	
	10C	
`A`	108	100
	104	
`B`	100	42

Pointers . . .

- Suppose we want to use `A` to get the value of `B`:

```
        .data
A:      .word B             # A = &B (B address)
B:      .word 42
        .text
main:   lw    $s0, A        # $s0 = A (&B)
        lw    $s1, 0($s0)   # $s1 = *A (B)
```

- $s0 now contains the address of the variable B, and $s1 contains its value (42).
- We never used the label `'B'` in the code; instead we got the value of `B` through the pointer `A`.

Addressing Modes – Direct memory

- The address format of the MIPS hardware enables several addressing modes.
- For short addresses we can use *direct memory* addressing:

```
lw $s0, 32($zero)
```

- This instruction loads the value at address 32. This is called direct memory addressing because the instruction contains the exact address.
- The immediate value (address) must fit in 16 bits.

Addressing Modes – Register indirect

- Register indirect addressing:

```
lw $s0, 0($s1)
```

- This instruction loads the value from memory at the address contained in register $s1.
- Register indirect is often used to implement pointers.

Register indirect with offset

```
lw $s0, 4($s1)
```

- This loads the value from memory address `$s1 + 4`.
- This is useful for creating structures or records – a collection of related data:

```
        .data
info:   .word 10,11,12
        .text
main:   la    $s0, info      # $s0 = &info
        lw    $s1, 0($s0)    # $s1 = 10
        lw    $s1, 4($s0)    # $s1 = 11
        lw    $s1, 8($s0)    # $s1 = 12
```

Large addresses

- The addressing mode provided by the MIPS hardware is rather limited, primarily because the constant value is only 16 bits.
- The assembler supports full 32-bit addresses, but may have to generate additional instructions to handle them (using register $at ($r1) if necessary).
- This instruction loads from the address associated with the `label`.

```
lw $s0, label
```

Large addresses…

- This may require two instructions if label is larger than 16 bits, e.g.

```
lw    $s0, 0x12345678
      ⇓
lui   $at, 0x1234
lw    $s0, 0x5678($at)
```

- The assembler will make this translation automatically.

Complex addressing modes …

- This instruction loads from the address ⌜`label` ± `32`⌝ (either +32 or -32 works):

```
lw $s0, label ± 32
```

- This instruction loads from the address ⌜`label` ± `(4 + $s1)`⌝:

```
lw $s0, label ± 4($s1)
```

  This form is useful for indexing arrays.

Example I

- Load each integer in array into register `s3`.

```
      .data
array:.word 1,3,5,7
      .text
main: move  $s1, $zero           # i = 0 (index)
loop:                            # do
      mul   $s2, $s1, 4          # offset = i * 4
      lw    $s3, array + 0($s2)  # next element
      add   $s1, $s1, 1          # i = i+1
      blt   $s1, 4, loop         # while i < 4
```

Example I…

- You can eliminate the `mul` and register `s2` if you count by 4s. This changes the loop limit to 4*4:

```
      .data
array:.word 1,3,5,7
      .text
main: move  $s1, $zero            # i = 0 (index)
loop:                             # do
      lw    $s3, array + 0($s1)   # next element
      add   $s1, $s1, 4           # i = i+1
      blt   $s1, 16, loop         # while i < 4
```

Example II

- Consider an array of 3 structures, each with two fields.
- The first field is an integer and the second a character.
- Write a program that does the following:
  1. Sum the first field for all structures, and
  2. print the ASCII value of the character in each.

- First declare the data-structures:

```
      .data             # Data starts here.
      .align   2        # Align on 4-byte boundary
array:                  # array starts here.
      .word    10       # 1st record starts here.
      .byte    'A'      #
      .align   2        #
      .word    17       # 2nd record starts here.
      .byte    'B'      #
      .align   2        #
      .word    89       # 3rd record starts here.
      .byte    'Z'      #
```

- Then declare the code:

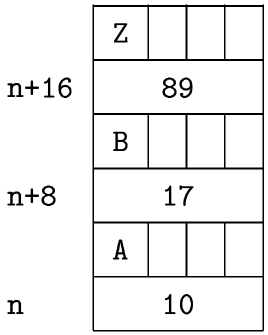
```
      .text
main: move  $s0, $zero            # sum = 0
      move  $s1, $zero            # i = 0
      bge   $s1, 3, done
loop: mul   $s2, $s1, 8           # s2 = offset
      lw    $s3, array + 0($s2)   # s3 = array[i].int
      add   $s0, $s0, $s3         # sum = sum + s3
      lbu   $a0, array + 4($s2)   # a0 = array[i].ch
      li    $v0, 1                # print_int
      syscall
      add   $s1, $s1, 1           # i = i + 1
      blt   $s1, 3, loop
done:
```

Example II. . .

- The ⌜`.align 2`⌝ directives are necessary because each structure requires 5 bytes (4 for the integer and 1 for the character), but the word in the next structure must be word aligned.
- The ⌜`.align n`⌝ directive aligns what follows it on a 2^n boundary.
- Counting by 8 eliminates the multiply.
- You could also shift left by 3 to avoid the multiply.

Example II. . .

- As a result, each structure is 8 bytes long (the last 3 bytes aren't used) so we multiply by 8 to get the offset. Pictorially:



Readings and References

-