Instruction Representation 2

Review

- MIPS defines instructions to be same size as data (one word) so that they can use the same memory (can use **`lw`** and **`sw`**)
- Machine Language Instruction:
  - 32 bits representing a single instruction

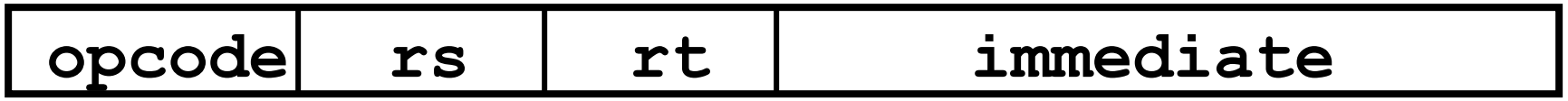
R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		

- Computer actually stores programs as a series of these machine instructions

Outline

- Branch instruction encoding
- Jump instructions
- Disassembly
- Pseudoinstructions and
  "True" Assembly Language (TAL) v.s.
  "MIPS" Assembly Language (MAL)

Branches: PC-Relative Addressing (1/5)



- Use I-Format
- opcode specifies **beq** versus **bne**
- **Rs** and **Rt** specify registers to compare
- What can immediate specify?
  - Immediate is only 16 bits
  - PC is 32-bit pointer to memory
  - Immediate cannot specify entire address to which we want to branch

Branches: PC-Relative Addressing (2/5)

- How do we usually use branches?
  - Answer: **`if-else`, `while`, `for`**
  - Loops are generally small: typically up to 50 instructions
  - Function calls and unconditional jumps are done using jump instructions (`j` and `jal`), not the branches
- Conclusion: Though we may want to branch to anywhere in memory, a single branch will generally change the PC by a very small amount

Branches: PC-Relative Addressing (3/5)

- Solution: PC-Relative Addressing
- Let the 16-bit **`immediate`** field be a signed two's complement integer to be *added* to the PC if we take the branch.
- Now we can branch +/- 2^{15} bytes from the PC, which should be enough to cover any loop.
- Any ideas to further optimize this?

Branches: PC-Relative Addressing (4/5)

- Note: Instructions are words, so they're word aligned
  - The byte address is always a multiple of 4
  - Which means it ends with 00 in binary
  - The number of bytes to add to the PC will always be a multiple of 4
  - Thus, specify the **immediate** in words.
- We can branch +/- 2^{15} words from the PC (or +/- 2^{17} bytes)
- Thus, we can handle loops 4 times as large as a byte offset

Branches: PC-Relative Addressing (5/5)

- Branch Calculation:
  - If we don't take the branch:

    PC = PC + 4

    PC+4 = byte address of next instruction
  - If we do take the branch:

    PC = (PC + 4) + (**`immediate`** * 4)
  - Observations
    - **`Immediate`** field specifies the number of words to jump, which is simply the number of instructions to jump
    - `Immediate` field can be positive or negative.
    - Due to hardware, add **`immediate`** to (PC+4), not to PC;
    - This will be clearer why later in course

Branch Example (1/3)

- MIPS Code:

```
Loop: beq   $9 $0 End
      add   $8 $8 $10
      addi  $9 $8 -1
      j     Loop
End:
```


- Branch is I-Format:

  opcode = 4 (look up in table)

  rs = 9 (first operand)

  rt = 0 (second operand)

  immediate = ???

Branch Example (2/3)

- MIPS Code:

```
Loop: beq   $9 $0 End
      add   $8 $8 $10
      addi  $9 $8 -1
      j     Loop
End:
```


- **`Immediate`** Field:
  - Number of instructions to add to (or subtract from) the PC, starting at the instruction *following* the branch.
  - In **`beq`** case, **`immediate`** = 3

Branch Example (3/3)

- MIPS Code:

```
Loop: beq   $9 $0 End
      add   $8 $8 $10
      addi  $9 $8 -1
      j     Loop
End:
```


THE FINE PRINT:

The textbook Appendix B has the **rs rt** order swapped (a typo?). But it doesn't matter for **BEQ** because we are checking equality. Contrast with SLTI…

beq rs, rt, label

4	rs	rt	Offset
6	5	5	16

slti rt, rs, imm

0xa	rs	rt	imm
6	5	5	16

decimal representation:

4	0	9	3

binary representation:

000100	00000	01001	0000000000000011

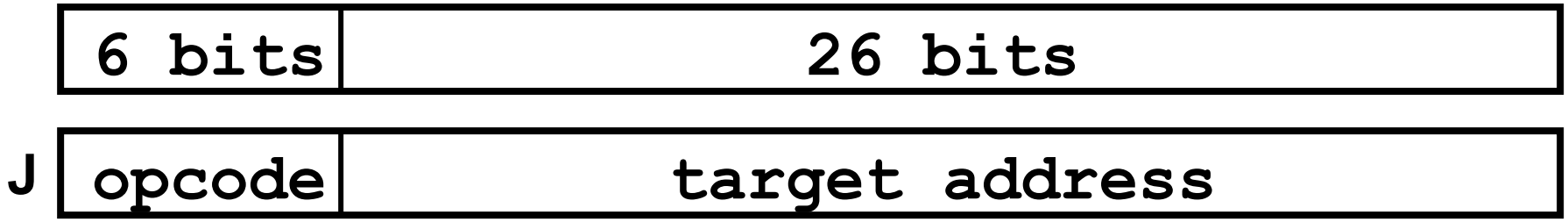
Questions on PC-addressing

- Does the value in branch field change if we move the code?
- What do we do if its > 2^{15} instructions?
- Since its limited to +/- 2^{15} instructions, doesn't this generate lots of extra MIPS instructions?

J-Format Instructions (1/5)

- For branches, we assumed that we won't want to branch too far, so we can specify *change* in PC.
- For general jumps (`j` and `jal`), we may jump to *anywhere* in memory.
- Ideally, we could specify a 32-bit memory address to jump to.
- Unfortunately, we can't fit both a 6-bit opcode and a 32-bit address into a single 32-bit word, so we compromise.

J-Format Instructions (2/5)



- Define "fields" as above
  - As usual, each field has a name
- Key Concepts
  - Keep opcode field identical to R-format and I-format for consistency.
  - Combine all other fields to make room for large target address.

J-Format Instructions (3/5)

- For now, we can specify 26 bits of the 32-bit bit address.
- Optimization:
  - Note that, just like with branches, jumps will only jump to word aligned addresses, so last two bits are always 00 (in binary).
  - So let's just take this for granted and not even specify them.

J-Format Instructions (4/5)

- So, we can specify 28 bits of the 32-bit address.
- Where do we get the other 4 bits?
  - **Always take the 4 highest order bits from the PC**
  - Technically, it means that we cannot jump *anywhere* in memory, but it's adequate 99.9999…% of the time, since programs aren't that long
  - If we ***absolutely*** need to specify a 32-bit address, we can always put it in a register and use the **`jr`** instruction

J-Format Instructions (5/5)

- Summary, with **||** meaning concatenation

New PC = PC[31..28] **||** target address (26 bits) **||** 00

4 bits **||** 26 bits **||** 2 bits = 32-bit address

- Understand where each part came from!

Outline

- Branch instruction encoding
- Jump instructions
- Disassembly
- Pseudoinstructions and "True" Assembly Language (TAL) v. "MIPS" Assembly Language (MAL)

Decoding Machine Language

- How do we convert 1s and 0s to C code?
  - Machine language → assembly → C
- For each 32 bits:
  - Look at opcode: 0 means R-Format, 2 or 3 mean J-Format, otherwise I-Format
  - Instruction type determines which fields exist
  - Write out MIPS assembly code, converting each field to name, register number/name, or decimal/hex number
  - Logically convert this MIPS code into valid C code. Always possible? Unique?

Decoding Example (1/7)

```
00001025
0005402A
11000003
0044102D
20A5FFFF
08100001
```

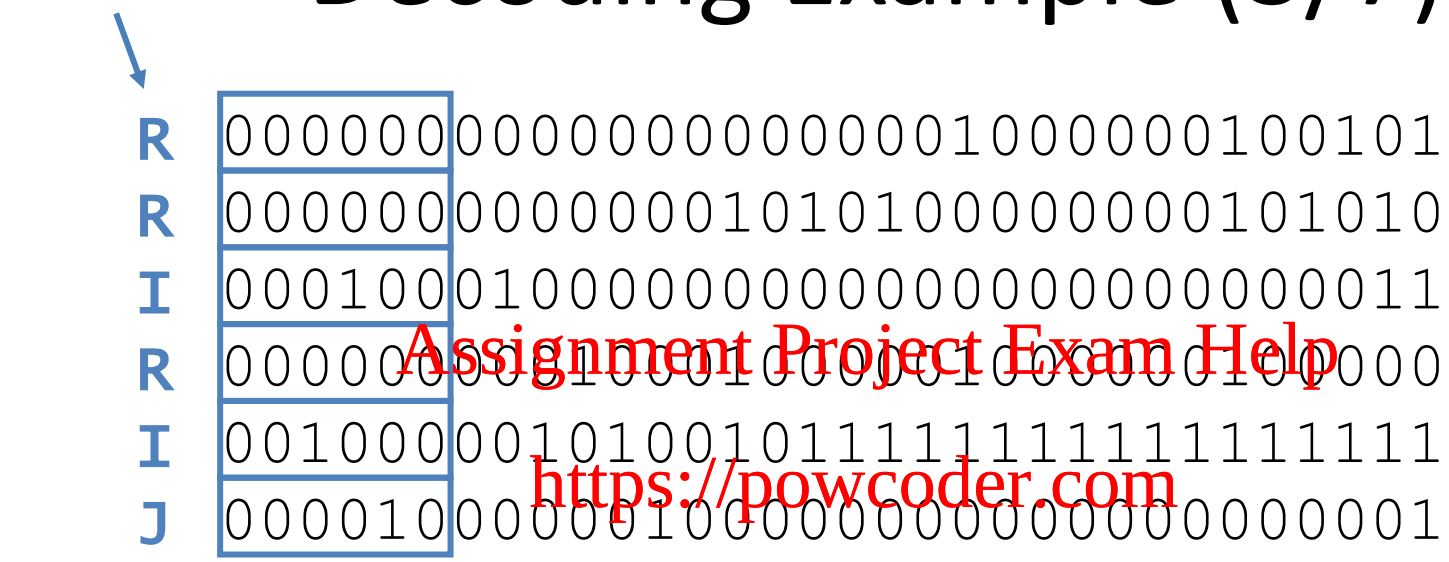

- Six machine language instructions in hex
- Let the first instruction be at address 419430410 (0x00400000).
- Next step: convert to binary

Decoding Example (2/7)

```
00000000000000000010000000100101
00000000000001010100000000101010
00010001000000000000000000000011
00000000010001000100000010000000
00100000101001011111111111111111
00001000000100000000000000000001
```

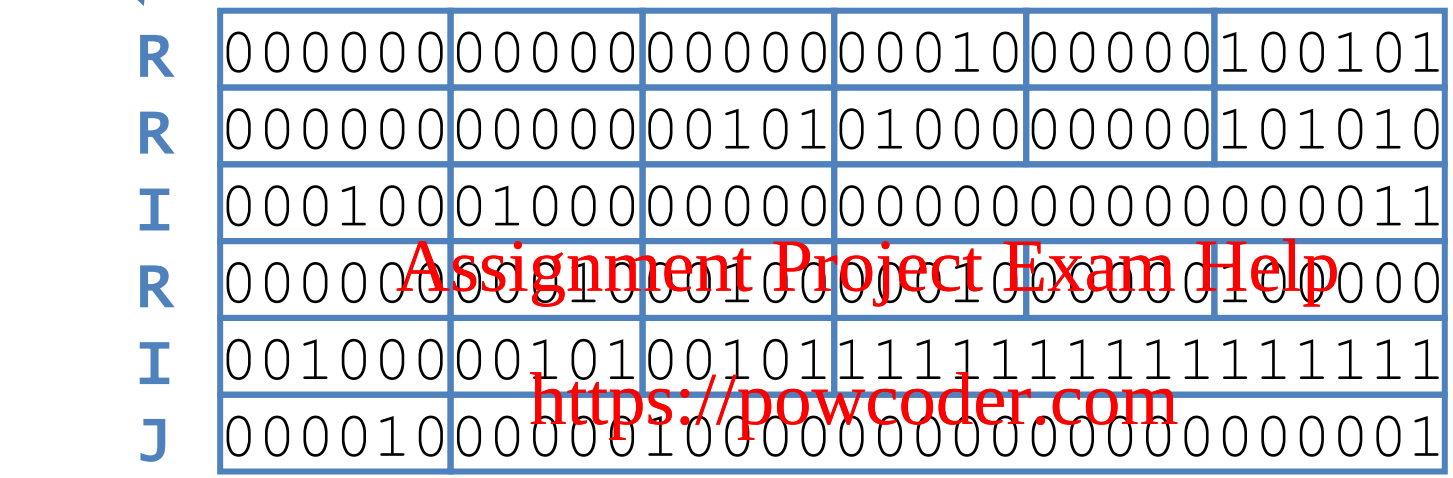

- The machine language instructions in binary
- Next step: identify opcode and format

Decoding Example (3/7)



- Opcode (first 6 bits) to determine the format
- 0 means R-Format, 2 or 3 means J-Format, otherwise I-Format
- Next step: separation of fields

Decoding Example (3/7)



- Fields separated based on format/opcode

Decoding Example (4/7)

Format							
R	0	0	0	2	0	37	or
R	0	0	5	8	0	42	slt
I	4	8	0	3			beq
R	0	8	4	2	0	32	add
I	8	5	5	-1			addi
J	2	1048577					j

- Convert binary to decimal
- Next step: translate ("disassemble") to MIPS assembly instructions

Decoding Example (5/7)

```
0x00400000   or     $2, $0, $0
0x00400004   slt    $8, $0, $5
0x00400008   beq    $8, $0, 3
0x0040000c   add    $2, $2, $4
0x00400010   addi   $5, $5, -1
0x00400014   j      0x100001
```


- MIPS Assembly, with memory locations
- For a Better solution, translate to more meaningful instructions
  - Need to fix the branch and jump and add labels

Decoding Example (6/7)

```
                or    $v0, $0, $0
LOOP:           slt   $t0, $0, $a1
                beq   $t0, $0, EXIT
                add   $v0, $v0, $a0
                addi  $a1, $a1, -1
                j     LOOP
EXIT:
```


- Next step: translate to C code
  - Many options

Decoding Example (7/7)

- C code:
  - Mapping: $v0: product
    $a0: multiplicand
    $a1: multiplier


```
product = 0;
while (multiplier > 0) {
   product += multiplicand;
   multiplier -= 1;
}
```

Pseudoinstructions

Outline

- Branch instruction encoding
- Jump instructions
- Disassembly
- Pseudoinstructions and
  “True” Assembly Language (TAL) v. “MIPS” Assembly Language (MAL)

Recall Load Upper Immediate (LUI)

- So how does **`lui`** help us?
  - Example:

```
addi   $t0,$t0, 0xABABCDCD
```

  becomes:

```
lui    $at, 0xABAB
ori    $at, $at, 0xCDCD
add    $t0,$t0,$at
```

  - Now each I-format instruction has only a 16-bit immediate.
- *Assembler can do this automatically!*
  - If number too big, then just automatically replace `addi` with `lui`, `ori`, `add`

True Assembly Language

- Pseudoinstruction: A MIPS instruction that doesn't turn directly into a machine language instruction.
- What happens with pseudoinstructions?
  - They're broken up by the assembler into several "real" MIPS instructions.
  - But what is a "real" MIPS instruction? Answer in a few slides
- First some examples

Example Pseudoinstructions

- Register Move

```
move   reg2, reg1
```

Expands to:

```
add    reg2, $zero, reg1
```

- Load Immediate

```
li     reg, value
```

If value fits in 16 bits:

```
addi   reg, $zero, value
```

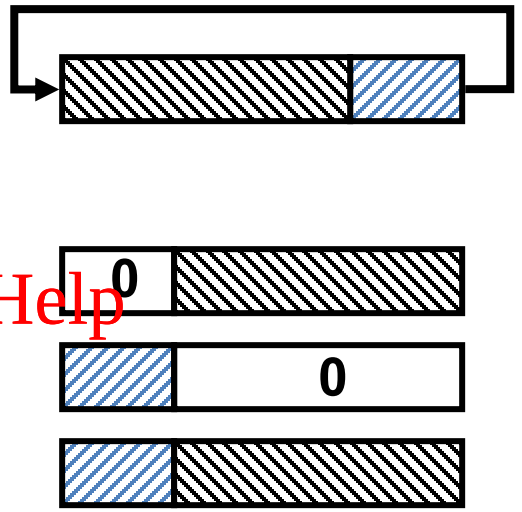
else:

```
lui    reg, upper_16_bits_of_value
ori    reg, reg, lower_16_bits
```

True Assembly Language

- Problem:
  - When breaking up a pseudoinstruction, the assembler may need to use an extra register.
  - If it uses any regular register, it'll overwrite whatever the program has put into it.
- Solution:
  - Reserve a register (`$1`, called `$at` for "assembler temporary") that the assembler will use when breaking up pseudo-instructions.
  - Since the assembler may use this at any time, it's not safe to code with it.

Example Pseudoinstructions



- Rotate Right Instruction

```
ror     reg, value
```

Expands to:

```
srl     $at, reg, value
sll     reg, reg, 32-value
or      reg, reg, $at
```


- No operation instruction

```
nop
```

Expands to instruction = 0_{ten}

```
sll     $0, $0, 0
```

Example Pseudoinstructions

- Wrong operation for operand

```
addu    reg, reg, value # should be addiu
```

If value fits in 16 bits:

```
addiu   reg, reg, value
```

else:

```
lui     $at, upper 16 bits of value
ori     $at, $zero, lower 16 bits of value
addu    reg, reg, $at
```

True Assembly Language

- MAL (MIPS Assembly Language): the set of instructions that a programmer may use to code in MIPS; this includes pseudoinstructions
- TAL (True Assembly Language): the set of instructions that can actually get translated into a single machine language instruction (32-bit binary string)
- A program must be converted from MAL into TAL before it can be translated into 1s and 0s.

Questions on Pseudoinstructions

- How does MIPS assembler recognize pseudoinstrucitons?
  - It looks for officially defined pseudo-instructions, such as `ror` and `move`
  - It looks for special cases where the operand is incorrect for the operation and tries to handle it gracefully

Question

- Which lines below are pseudo-instructions (MIPS Assembly Language); that is, not TAL?

1. addi $t0, $t1, 40000
2. beq $s0, 10, Exit
3. sub $t0, $t1, 1

A. 1 only
B. 2 only
C. 3 only
D. 1 and 2
E. 2 and 3
F. All of the above

Question

- Which lines below are pseudo-instructions (MIPS Assembly Language); that is, not TAL?

1. addi $t0, $t1, 40000
2. beq $s0, 10, Exit
3. sub $t0, $t1, 1

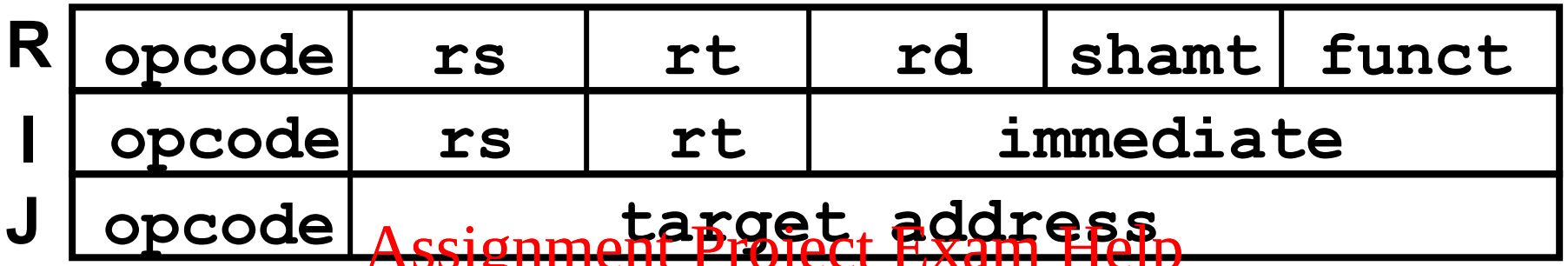
40,000 > +32,767 thus need lui, ori

beq: both must be registers

sub: both must be registers;
even if it was subi,
there is no subi in TAL;
generates addi $t0,$t1, -1

A. 1 only
B. 2 only
C. 3 only
D. 1 and 2
E. 2 and 3
F. All of the above

Summary



- Machine Language Instruction:
  - 32 bits representing a single instruction

- Branches use PC-relative addressing,
  Jumps use absolute addressing

- Disassembly is easy: starts by decoding `opcode` field
- Assembler expands real instruction set (TAL) with pseudoinstructions (MAL)

Summary

- To understand the MIPS architecture and be sure to get best performance, it is best to study the True Assembly Language instructions.

Organization of an Assembly Program

- Assembly Program
  - Just plain text file with data declarations, program code
  - Suffix .asm for MARS simulator (suffix .s in some other simulators)
  - Contains data declaration section followed by program code section
- Data Declarations
  - Placed in section of program identified with assembler directive **.data**
  - Declares variable names used in program; storage allocated in main memory (RAM)
- Code
  - Placed in section of text identified with assembler directive **.text**
  - Contains program code (instructions)
  - Starting point for code execution given label **main:**
  - Ending point of main code should use exit system call (more later…)

Template for a MIPS program

```
# Bare-bones outline of MIPS assembly language program

.data
# variable declarations here
# ...

.text

main: # indicates start of code (first instruction to execute)
# remainder of program code here
# ...
# ...
```

Data Declarations

- Format for declarations:

```
name: .storage_type value(s)
```


  - Optional label identifier always followed by colon

  - Storage type directive always starts with a period
  - Create storage for variable of specified type, given name, specified value

  - Value(s) gives initial value(s), except for storage type `.space`, when the value gives number of bytes to be allocated

Data Declaration Examples

```
var1:    .word  3           # integer variable with initial value 3

array1: .byte 'a','b'       # 2-element character array,
                            # with elements initialized to a and b

array2: .space 40           # allocate 40 consecutive bytes,
                            # with storage uninitialized
                            # could be used as a 40-element character
                            # array, or a 10-element integer array;
                            # a comment should indicate which!

str1:    .asciiz "hi!"      # null terminated string 68 69 21 00
w1:      .word 0x00216968   # same as str1 for little endian
w2:      .word 0x68692100   # same as str1 for big endian

myStructure:                # structure with a float and string pointer
.float   1.5
.word    str1
```

Little Endian vs Big Endian

How are words stored in Memory?

- **Little endian** - Stores the **L**east significant byte at the **lowest address**. Example: Intel Pentium Processors.
- **Big endian** - Stores the Most (**B**ig) significant byte at the **lowest address**. Example: Sun/SPARC, IBM/RISC 6000.

System Calls and I/O (MARS Simulator)

Service	Code	Arguments	Result
print integer	1	$a0 = value	(none)
print float	2	$f12 = float value	(none)
print double	3	$f12 = double value	(none)
print string	4	$a0 = address of null terminated string	(none)
read integer	5	(none)	$v0 = value read
read float	6	(none)	$f0 = value read
read double	7	(none)	$f0 = value read
read string	8	$a0 = address where string to be stored $a1 = number of characters to read + 1	(none)
memory allocation	9	$a0 = number of bytes of storage desired	$v0 = address of block
exit (end program)	10	(none)	(none)

Hello World Example

```
            .data
string1:    .asciiz   "Hello World.\n"

            .text
main:       li $v0, 4           # load system call code 4 into $v0
            la $a0, string1 # address of string to print into $a0
            syscall             # call operating system to print
            li $v0, 10          # load exit call code 10 into $v0
            syscall             # call operating system to exit
```


LA and LI pseudoinstructions do the same job.

Using LA instead of LI lets us show that we are loading an address.

- We should probably skip the rest of this stuff…

Examples using Pointers

- The following slides are more practice on the differences between a pointer and a value, and showing how to use pointers
- Example uses of pointers:
  - Arrays of primitive types (e.g., integer)
  - Pointers to data structures

Assembly Code to Implement Pointers

- Dereferencing ⇒ data transfer in assembly

  `... = ... *p ...;` ⇒ load

  Get value from location pointed to by p,

  load word (**`lw`**) if int pointer,

  load byte unsigned (**`lbu`**) if char pointer

  `*p = ...;` ⇒ store

  Put value into location pointed to by p

Assembly Code to Implement Pointers

Let `c` be an `int`, have value 100, be at memory address 0x10000000.

Let `p` be in `$a0`, `x` in `$s0`

```
p = &c;   /* p gets 0x10000000 */
x = *p;   /* x gets 100 */
*p = 200; /* c gets 200 */
```



```
                  # p = &c;  /* p gets 0x10000000 */
lui $a0, 0x1000   # p = 0x10000000
                  # x = *p;  /* x gets 100 */
lw  $s0, 0($a0)   # dereferencing p
                  # *p = 200; /* c gets 200 */
addi $t0, $0, 200
sw   $t0, 0($a0)  # dereferencing p
```

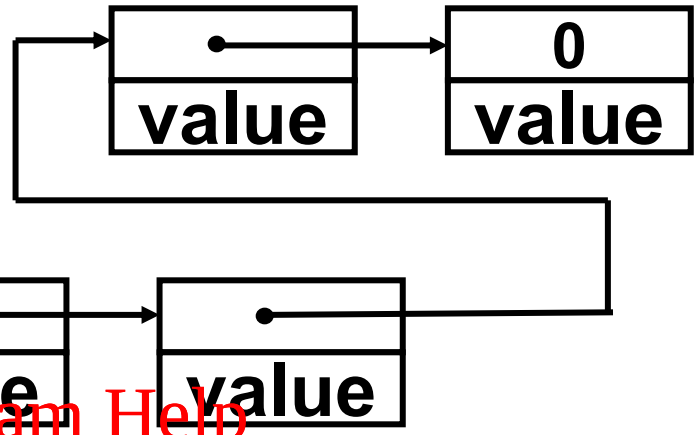
Practice with Arrays

- Implement a bubble sort function

```
void bubblesort( int* A, int length )  {
  boolean swapped;
  int n = length - 1;
  do {
    swapped = false;
    for ( int i = 0; i < n; i++ ) {
      if ( A[i] > A[i+1] ) {
        swap( A, i, i+1 );
        swapped = true;
      }
    }
    n = n - 1;
  } while ( swapped );
}
```


```
Void swap( int* A, int i, int j )  {
  int tmp = A[i];
  A[i] = A[j];
  A[j] = tmp;
}
```

Pointers to Structures



C Example linked list:


```
struct node{
    struct node *next;
    int value;
};
```

If `p` is a pointer to a node, declared with `struct node *p`, then:

`(*p).value` or `p->value` for "value" field,

`(*p).next` or `p->next` for pointer to next node

Linked-list in C

```
main (void) {
  struct node *head, *temp, *ptr;
  int sum;

  /* create the nodes*/
  head = (struct node *) malloc(sizeof(struct node));
  head->value = 23;
  head->next = 0;

  temp = (struct node *) malloc(sizeof(struct node));
  temp->next = head;
  temp->value = 42;

  head = temp;

  /* add up the values */
  ptr = head;   sum = 0;
  while (ptr != 0) {
    sum += ptr->value;
    ptr = ptr->next;
  }
}
```

Linked-list in MIPS Assember (1/2)

```
# head:s0, temp:s1, ptr:s2, sum:s3
# create the nodes
        li   $a0,8# sizeof(node)
        jal  malloc     # the call
        move $s0,$v0    # head gets result
        li   $t0,23
        sw   $t0,4($s0) # head->value = 23
        sw   $zero,0($s0)# head->next = NULL

        li   $a0,8
        jal  malloc
        move $s1,$v0        # temp = malloc
        sw   $s0,0($s1) # temp->next = head
        li   $t0,42
        sw   $t0,4($s1) # temp->value = 42

        move $s0,$s1        # head = temp
```

In MARS we would use SYSCALL 9 instead

Linked-list in MIPS Assember (2/2)

```
# head:s0, temp:s1, ptr:s2, sum:s3

        #add up the values
        move $s2,$s0          # ptr = head
        move $s3,$zero        # sum = 0
loop:   beq  $s2,$zero,exit # exit if done
        lw   $t0,4($s2)       # get value
        addu $s3,$s3,$t0      # compute new sum
        lw   $s3,0($s2)       # ptr = ptr->next
        j    loop             # repeat
exit:
```

Review and More Information

- Textbook
  - 2.5 Representing Instructions in the computer
  - 2.10 Addressing for 32-bit immediates
  - 2.12 Translating and Starting a Program
    - Just the section on the **Assembler** with respect to pseudoinstructions (pg 124, 125, 5th edition)