Notes for Assignment #3

I have created three videos ("V07: Decoding Instructions", "V08: The Sort Program" and "V09: Example Problem") under the Media Gallery Tab that you should watch as part of this assignment. Here are some comments on the reading assignment from the Patterson & Hennessy text:

Page 103: You are probably familiar with the concept of an activation record. There is a register, $fp, that can be used to identify the top of the activation record, though it is typically not needed (unless a procedure is altering the contents of the $sp register throughout the procedure). Notice that the value of $sp is always pointing to the bottom of the activation record.

Page 104: On this page we see the memory layout that is used by MIPS. Much of this is transparent and you need not be concerned with it for now, but it does emphasize that everything is stored in memory: your machine code, static memory, dynamic memory and the stack (notice that references to static memory are not absolute; they are relative to the contents of the $gp register). It is pretty clear that things can go haywire if you store data in the wrong place!

Page 105: The table at the top of this page tells us that it is the responsibility of a procedure to preserve (and restore) the values of $fp and $gp before they are changed. The picture of register responsibilities is now complete!

Tail recursion can easily be replaced by iterative code that will execute faster; on the other hand, the recursive version is often easier to write and understand.

Page 106: There are some tabs missing after the opcodes in the MIPS code at the top of this page.

There is a typo in the quote from the keyboard poem at the beginning of section 2.9. It should read:

!(@|=>

Pages 107-111: There is a two-instruction sequence here that demonstrates the loading and storing of a byte of data between two memory locations. There is no real significance to the fact that the example happens to use $sp and $gp (this would be copying a byte from the stack to the static data area).

In the strcpy code, the authors use lbu, rather that lb, so we should quickly explore the difference. Assume that a word in memory contained the following:

0111 1110 0101 1010 1000 0001 1001 1010

A "lb" of the bottom byte of this word into, say $t0, would leave this value in $t0 (note the sign extension):

1111 1111 1111 1111 1111 1111 1001 1010

A "lbu" would give us (in this case, the register is filled with zeroes):

0000 0000 0000 0000 0000 0000 1001 1010

It should be noted that the strcpy routine would still work if we had used the lb (rather than lbu) instruction, since the sb instruction ignores all bits beyond the rightmost eight. The lbu instruction becomes useful when you need to examine individual characters that are loaded.

The authors assume that you are familiar with the way that the language C stores strings with a terminating zero byte (null character). Remember that a string of length N would require N+1 bytes of storage because of this zero byte.

The strcpy procedure is a traditional example that demonstrates the compactness of the C language. In essence, the parameters x and y (which in MIPS are passed in registers $a0 and $a1) are the base addresses of the two arrays (remember that we are copying from y to x). The code assumes that the array x is at least as long as the array y (no check is made, or is even possible). After transferring the ith element of y to x, the while loop checks to see if that character was the zero byte; if so, the loop terminates. Otherwise, i is incremented and the loop continues.

Here is the MIPS code that the text suggests for this procedure:

```
strcpy: addi    $sp, $sp, -4
        sw      $s0, 0($sp)

        add     $s0, $zero, $zero

L1:     add     $t1, $s0, $a1
        lbu     $t2, 0($t1)

        add     $t3, $s0, $a0
        sb      $t2, 0($t3)

        beq     $t2, $zero, L2

        addi    $s0, $s0, 1
        j       L1

L2:     lw      $s0, 0($sp)

        addi    $sp, $sp, 4
        jr      $ra
```

As suggested in the last assignment, we can streamline things in a leaf procedure by using the temporary registers (rather than the saved registers) to keep track of local variables, like the variable i. This will eliminate the need to save register values on the stack. We also saw that the "li" pseudoinstruction can make the loading of a constant zero more straightforward (rather than the cryptic add of two registers containing zero!). The following revision to the code incorporates these improvements (we use register $t0 rather than $s0):

```
strcpy:  li      $t0, 0

L1:      add     $t1, $t0, $a1
         lbu     $t2, 0($t1)

         add     $t3, $t0, $a0
         sb      $t2, 0($t3)

         beq     $t2, $zero, L2

         addi    $t0, $t0, 1
         j       L1

L2:      jr      $ra
```

Let's test our code! Here is the same code with a little driver program added:

```
strcpy:  li      $t0, 0

L1:      add     $t1, $t0, $a1
         lbu     $t2, 0($t1)

         add     $t3, $t0, $a0
         sb      $t2, 0($t3)

         beq     $t2, $zero, L2

         addi    $t0, $t0, 1
         j       L1

L2:      jr      $ra

main:    la      $a0, result
         la      $a1, quote

         jal     strcpy

         la      $a0, result
         li      $v0, 4
         syscall
         la      $a0, cr
         syscall

         li      $v0, 10
         syscall

         .data
quote:   .asciiz  "MIPS programmers rule the world!"
cr:      .asciiz  "\nSee you later, chief.\n"
result:  .space   50
```

Remember that the .asciiz directive creates a string with the zero byte added at the end; this is perfect for our demonstration program! The MIPS .space directive allocates 50 bytes of storage (result would be the address of the first of those 50 bytes). There is no significance to the value 50, except that I was confident that my quote had length less

than that! Here are a couple of fine points: (1) at first glance it may seem that the second "la $a0, result" instruction is not necessary, but remember that the parameter registers ($a0-$a3) are not preserved during procedure calls; and (2) since we are assuming that a syscall does preserve all registers (except a result register, if a value is returned), we don't bother to reset $v0 to 4 before the cr string is displayed.

Here is the output produced by an execution of this code:

```
MIPS programmers rule the world!
See you later, chief.
```

When we cover section 2.14 of the text, we will revisit this code one more time to see how pointer arithmetic can further simplify the solution (or at least shorten it!).

Unicode provides a wealth of new display symbols, but at a cost in storage. Notice that there are three special instructions, lh, lhu and sh, that allow manipulation of 16-bit values (halfwords).

Assume that a word in memory contained the following:

0111 1110 0101 1010 1000 0001 1001 1010

An "lh" on the bottom half of the word into $t0 would give us (note the sign extension):

1111 1111 1111 1111 1000 0001 1001 1010

Finally, a "lhu" would give us (in this case, the register is filled with zeroes):

0000 0000 0000 0000 1000 0001 1001 1010

UTF-8 actually uses a variable length encoding that maintains the 8-bit coding for ASCII. Some of you may have seen such techniques in an algorithms course (Huffman Codes are the classical example) where the more frequently occurring symbols use shorter representations than the other symbols. This gives you the best of both worlds: an expanded character set without an enormous increase in total storage requirements.

Page 112: Notice that the constant field in I-format instructions is only 16 bits long. Therefore, it takes two instructions to load a 32-bit constant into a register. The first instruction, lui, places a 16-bit constant into the top half of a register, filling the bottom half with zeroes (this filling of the bottom half with zeroes is quite important). The bottom half of the constant is inserted with the ori (or immediate) instruction; note how the correct use of the ori instruction depends on the bottom half of the register being zeroed out. Also, notice that an add immediate (addi) is not used in this second step; there would be a danger of an unwanted sign extension of the top bit of the constant.

I've got some good news! You probably remember that we have been using the li (load immediate) pseudoinstruction to set the register $v0 before doing a syscall. The MIPS assembler (and the QtSpim simulator) are pretty smart! If you use li with a constant that will fit in 16 bits, then the instruction actually assembled is an ori with $zero and that 16-bit constant; otherwise, the assembler creates the two-instruction sequence (lui and ori) that accomplishes the task. That is pretty cool, don't you think?

Pages 113-115: This is important, but interesting stuff! Since conditional branching typically causes a branch to a "nearby" location (i.e., not too far from the location of the

branch itself), the address that appears in such branches is relative to the value in the $pc register (and the $pc register will actually be pointing to the instruction immediately after the branch instruction in question). Even though memory is byte-addressable, all instructions begin at word boundaries (multiples of four). Because of this, the constant that appears in the conditional branch will indicate a number of words (rather than bytes) beyond or prior to (positive or negative, respectively) the value in the $pc register. This allows MIPS to make the best use of the 16-bit constant; the range of the branches is four times the amount that would be possible if byte-addressing were used. Do notice that the constant must allow for positive and negative values to accommodate branching to an instruction above the branch instruction.

The jump (j) and the jump-and-link (jal) instructions use the J-type instruction format that has a 26-bit constant (which is never negative). In this case the branch address is absolute (not relative to $pc); but in a fashion similar to the conditional branch instructions, the constant is the number of words from the beginning of memory (so a jump to the constant 1000 would actually be a jump to byte 4000). The elaboration on page 114 of the text discusses how jumps to an address beyond these bounds are accomplished; you'll find that it is quite clever!

Work carefully through the example on page 115; make sure that you understand why the constant in the instruction at word 80012 is 2 (not 3, or 8, or 12). Also notice that the constant in the instruction at word 80020 is 20000 (not 80000).

Page 116: We see here that you don't need to fret about trying to branch too far away using PC-relative addressing. The assembler will convert your code into an equivalent set of instructions (typically 2) that uses a jump instruction that has a 26-bit constant.

Page 117: More key stuff! You need to be intimately familiar with the addressing modes shown on this page. Immediate and register addressing have their values in the instruction itself, so there is no reference to memory.

Pages 118-120: You definitely need to know how to decode machine code. You certainly don't need to memorize a table like that shown on page 119, but you do need to know how to use it! Notice that opcodes 000000 and 010000 are further defined in separate tables (utilizing either the funct field or the rs field as an extension to the opcode). [This is done for a few other opcodes that we will encounter later in the term.] The table on page 120 provides a review of the basic instruction formats.

You do have to be aware of the order and use of the rs, rt, rd and shamt fields when putting an instruction together. For example, notice that the rd register is the third register in the instruction format; but rd is typically used to specify the destination register and that register appears first when writing that instruction (for example, in the instruction "add $t1, $t2, $t3", $t1 is the destination register). You will find that the green instruction card at the front of the textbook (also available under Course Materials) provides these details, as does Appendix A at the end of the text (particularly pages A-49 through A-80). Notice that Appendix A contains a more complete opcode map on page A-50; it's a little scary looking at first, but it provides the details needed to decode instructions like add or bgez.

Page 121: This material on synchronization may be new to many of you. The need for atomic operations comes up often in the study of operating systems. I hope all of you will agree that the simple lock described at the bottom of this page is really cool!

Page 122: The code sequence (and the explanation immediately following) on this page is problematic. Let us first look at the corresponding code sequence that appears in the 4th edition of the text:

```
try:    add     $t0, $zero, $s4     # copy exchange value  [line 1]
        ll      $t1, 0($s1)         # load linked          [line 2]
        sc      $t0, 0($s1)         # store conditional    [line 3]
        beq     $t0, $zero, try     # branch store fails   [line 4]
        add     $s4, $zero, $t1     # put load value in $s4 [line 5]
```

The "ll $t1, 0($s1)" instruction in this code sequence not only loads the value of the memory location pointed to by $s1, but also sets a user-transparent bit (LLbit). This LLbit is automatically reset if an event takes place (like a store by another process) that might possibly have changed the contents of that storage location. When the subsequent "sc $t0, 0($s1)" instruction executes, one of two things happens:

1) If the LLbit is still set, then the value in $t0 is stored into the location pointed to by $s1 and the value 1 (for success) is placed in $t0.
2) If the LLbit is no longer set, the store does not take place and the value 0 (for failure) is placed in $t0.

Notice that the original value of $t0 is destroyed in either case.

IMPORTANT NOTE: QtSpim does not simulate multiple processes, so the "sc" instruction will always succeed.

To see how this code sequence works, let's number the instructions 1 through 5 and assume that $s4 = 5, $s1 = 1000 and that location 1000 contains 10:

1) Instruction 1 executes; $t0 = 5.
2) Instruction 2 executes; $t1 = 10 and the LLbit is set.
3) Assume at this point that the value at location 1000 has been changed to 15 by some other process. This will automatically cause the LLbit to be reset.
4) Instruction 3 executes; since the LLbit is no longer set, the store fails, the contents of location 1000 remains 15 and $t0 is set to 0 (for failure).
5) Instruction 4 executes; since $t0 is zero, the branch to instruction 1 takes place.
6) Instruction 1 executes; $t0 is again set to 5.
7) Instruction 2 executes; $t1 = 15 (remember that some other process has changed the value at location 1000 to 15) and the LLbit is set.
8) Assume that no additional changes to location 1000 take place at this point.

9) Instruction 3 executes; since the LLbit is still set, the store takes place, the value at location 1000 is set to 5 and $t0 is set to 1 (for success).

10) Instruction 4 executes; since $t0 is not zero, the branch does not take place.

11) Instruction 5 executes; $s4 = 15.

The value of $s4 and the contents of location 1000 have been successfully swapped.

It is possible to have instructions between the "ll" and the "sc" instructions (but it, obviously, increases the possibility of interference by other processes). For example, you could treat the contents of a storage location as a shared counter that you are trying to increment (i.e., you would do the load, increment that value and attempt to store it back).

Let us take a quick look at the code sequence that appears in the current (5^{th}) edition of the text:

```
again:  addi    $t0, $zero, 1       # copy locked value
        ll      $t1, 0($s1)         # load linked
        sc      $t0, 0($s1)         # store conditional
        beq     $t0, $zero, again   # branch if store fails
        add     $s4, $zero, $t1     # put load value in $s4
```

It appears that the authors are attempting to implement the concept of a lock, where $s1 contains the address of the lock. In this case, we are attempting to set the lock by placing 1 into it (this is the value that is in $t0). Notice that once we get to the last line in this code sequence (the add instruction), we know that the atomic exchange has been successful (i.e., that the value 1 in $t0 has been placed into the lock and that the previous value of the lock is now in $t1), but <u>not</u> that we have been granted access to the lock. If the value of $t1 is zero, we <u>have</u> been granted access; on the other hand, if the value of $t1 is 1, we have merely interchanged a 1 with another 1 and we have <u>not</u> been granted access to the lock.

It should also be clear that it makes little sense to place the previous value of the lock in $s4. In fact, assuming that we want to loop until access to the lock is granted, I would suggest replacing the add instruction with

```
        bne     $t1, $zero, again   # loop until lock access granted
```

Before moving on to another topic, I would like to suggest a more reasonable piece of code that would implement the idea of a lock. Consider the following code (where it is again assumed that the <u>address</u> of the lock is in $s1):

```
again:  li      $t1, 1              # used when attempting to set lock
        ll      $t0, 0($s1)         # load linked value of lock
        bnez    $t0, again          # if lock already set, loop
        sc      $t1, 0($s1)         # store conditional (attempt to set lock)
        beqz    $t1, again          # loop back if store fails
```

When the code "falls through" the second branch, the lock has been successfully set. Once the lock is no longer needed, a simple

```
        sw      $zero, 0($s1)
```

releases the lock.

Pages 124-126: We have already seen pseudoinstructions (like li). Some pseudoinstructions require the use of a temporary register to accomplish their task. Register $at is reserved for that use (so, you as a programmer should not use $at directly). On page 125, the text discusses the blt pseudoinstruction (which, by the way, is not mentioned in the example on page 95). The assembler converts the blt pseudoinstruction into a slt and a bne instruction; it is necessary to temporarily use the $at register when this done. For example, an instruction like

```
blt     $t1, $t2, loop
```

is converted into two instructions:

```
    slt   $at, $t1, $t2
    bne   $at, $zero, loop
```

You should familiarize yourself with the six distinct pieces of an object file for Unix systems (object files for other operating systems are quite similar in content) and the linking process.

Page 128: Notice the use of the global pointer, $gp, for relative addressing (similar to the PC-relative addressing that we saw earlier). The 8000_{hex} and 8020_{hex} constants that appear in the example on this page actually represent negative values.

Pages 129-130: This is important information on loaders, dynamically linked libraries and the lazy procedure linkage versions of DLLs.

Page 132: Is the idea of a Just In Time compiler (JIT) cool, or what?

Page 134: Notice how the authors consistently use the sll (rather than mul) instruction when they need to multiply by a power of two. A shift instruction executes faster than a multiply.

The classical solution to the swap procedure involves placing one of the values in a temporary variable, traditionally called temp. When writing swap in assembly, we end up with a temporary copy of both values; since the values must be loaded into registers ($t0 and $t2, in this case), those registers essentially serve as temporary copies of the original values.

The code given in the text for swap looks good. Since it is a leaf procedure, there is no need to save any registers on the stack. The code is given below (with comments that are a little less verbose):

```
swap: sll   $t1, $a1, 2       # compute address of v[k]
      add   $t1, $a0, $t1

      lw    $t0, 0($t1)       # load v[k]
      lw    $t2, 4($t1)       # load v[k+1]

      sw    $t2, 0($t1)       # swap values
      sw    $t0, 4($t1)

      jr    $ra               # return to calling routine
```

Pages 135-139: After a quick look at the C code in figure 2.26, it should be clear that the outer loop might as well start at i = 1 (rather than 0). Also, the end of the inner for loop should have "j -= i" rather than just "j = i" (this is shown correctly on the middle of page 136).

You are encouraged to examine the authors' development of this procedure very carefully. Since sort is not the main program, it is obligated to preserve any of the save registers ($s0-$s7) that it uses. Since it is not a leaf procedure, sort must also be careful to preserve the value that is in $ra (so it can eventually get back to the procedure that called it) and its own parameters (in $a0 and $a1) before setting the parameters for swap.

There is a fine point here. Since swap's first parameter is the same as the first parameter for sort (the address of the array v), it would seem that there would be no need to preserve the value of $a0 (and you could get by with it here). However, you should not assume that the argument registers ($a0-$a3) are not changed by a called procedure. It is easy to imagine a situation where a procedure might choose to change the values of its parameters during execution. For example, if one of the parameters is a count, you might decrement the count until you reach zero. Remember that sort and swap are using the same set of registers!

Procedure sort also needs to preserve any of the temporary registers ($t0-$t9) that have values that are needed once the swap procedure returns. In this case, there are not any such registers. Notice that even though sort does use $t0 through $t4, sort does not depend on those values being unchanged by the procedure swap (i.e., the next reference to those registers involves a value being stored in those registers, not fetched).

I'd like to make one last point about the sort procedure preserving its own parameters before calling the swap procedure. The only reason that we had to do that was that our code planned to use the values of those parameters after the swap procedure was called. If that had not been the case, then we would have no need to preserve those values. In this sense, the argument registers are treated like the temporary registers; preserve them if you need them later. There are two ways to preserve them: put them on the stack, or put them in one of the saved registers (this is what was done with sort's parameters). But if you do use the saved registers, then the previous values of those registers must be protected through use of the stack. As circular as this discussion seems, you need to understand it!

Here is the final code for sort (with abbreviated comments). The move instruction is actually a pseudoinstruction that allows you to move one register to another. An instruction like "move $t1, $t2" actually gets converted into an instruction that adds $t2 to $zero and puts the result in $t1. I suspect that the use of "move" seems more natural!

```
sort:  addi $sp, $sp, -20      # save $ra and $s0-$s3
       sw   $ra, 16($sp)       # on stack
       sw   $s3, 12($sp)
       sw   $s2, 8($sp)
       sw   $s1, 4($sp)
       sw   $s0, 0($sp)

       move $s2, $a0           # keep sort's parameters
       move $s3, $a1           # in $s2 and $s3
```

```
        li   $s0, 1             # outer loop (i)
for1:   slt  $t0, $s0, $s3
        beq  $t0, $zero, exit1

        addi $s1, $s0, -1       # inner loop (j)
for2:   slti $t0, $s1, 0
        bne  $t0, $zero, exit2
        sll  $t1, $s1, 2
        add  $t2, $s2, $t1
        lw   $t3, 0($t2)        # load v[j] and v[j+1]
        lw   $t4, 4($t2)
        slt  $t0, $t4, $t3
        beq  $t0, $zero, exit2 # see if swap is needed

        move $a0, $s2           # if so, perform swap
        move $a1, $s1
        jal  swap

        addi $s1, $s1, -1       # decrement j (inner loop)
        j    for2

exit2:  addi $s0, $s0, 1        # increment i (outer loop)
        j    for1

exit1:  lw   $s0, 0($sp)        # restore values from stack
        lw   $s1, 4($sp)
        lw   $s2, 8($sp)
        lw   $s3, 12($sp)
        lw   $ra, 16($sp)
        addi $sp, $sp, 20

        jr   $ra                # return to calling routine
```

It's always nice to see your code work! The following driver program tests the sort/swap procedures using a specific array with 10 values:

```
main:  la     $a0, text1          # display original values in array
       li     $v0, 4
       syscall

       la     $a0, array          # call sort procedure
       li     $a1, 10
       jal    sort

       la     $a0, text2          # display sorted values
       li     $v0, 4
       syscall

       la     $t0, array          # base address of array
       li     $t1, 10             # number of values

loop:  beq    $t1, $zero, done    # print numbers one at a time

       lw     $a0, 0($t0)         # actual value
       li     $v0, 1
       syscall

       la     $a0, blank          # print a space after each number
       li     $v0, 4
       syscall
```

```
        addi   $t0, $t0, 4          # load next value
        addi   $t1, $t1, -1         # decrement counter
        j      loop

done:   la     $a0, cr              # print a newline
        li     $v0, 4
        syscall

        li     $v0, 10              # exit
        syscall

        .data
array: .word   100, -2, 19, 211, -80, 0, 16, 923, 19, 301
text1: .asciiz "Testing sort with 100 -2 19 211 -80 0 16 923 19 301\n"
text2: .asciiz "\nValues after sort: "
blank: .asciiz " "
cr:    .asciiz "\n"
```

To run this program, you would place the swap procedure, the sort procedure and the main program (if main is not last, you would need a ".text" prior to the code that follows the data) on one file. I have placed the resulting file, **sort.s**, under Course Materials.

Here is the output from an execution of my program:

```
Testing sort with 100 -2 19 211 -80 0 16 923 19 301
Values after sort: -80 -2 0 16 19 19 100 211 301 923
```

Pages 140-141: Figure 2.28 shows that the improvement in performance gained from code optimization is not trivial. Figure 2.29 shows that a more efficient algorithm can often beat the socks off of the most optimized code utilizing an inferior algorithm!

Pages 141-144: This code assumes that size is greater than zero. It should be obvious that the (final) pointer version of the clear procedure is more efficient, since some of the calculations are moved outside of the loop. The elaboration at the end of this section suggests a "fix" to the code that would allow it to work even if size is not positive. I am listing both versions (array indices and pointers) of the revised code below:

Array indices version

```
clear: move   $t0, $zero
       j      here
loop:  sll    $t1, $t0, 2
       add    $t2, $a0, $t1
       sw     $zero, 0($t2)
       addi   $t0, $t0, 1
here:  slt    $t3, $t0, $a1
       bne    $t3, $zero, loop
```

Pointers version

```
clear: move   $t0, $a0
       sll    $t1, $a1, 2
       add    $t2, $a0, $t1
       j      here
loop:  sw     $zero, 0($t0)
       addi   $t0, $t0, 4
here:  slt    $t3, $t0, $t2
       bne    $t3, $zero, loop
```

Code for strcpy was developed in Section 2.9. At that point I promised to revisit strcpy when Section 2.14 was covered. Here is a streamlined version of strcpy, using pointers. It is also an example of a procedure that actually changes both of its parameters (hopefully, this brings home the fact that you should not assume that a procedure will not change its parameters).

```
strcpy: lb   $t1, 0($a1)      # load a character
        sb   $t1, 0($a0)      # store that character

        beq  $t1, $zero, L1   # quit if that character
```

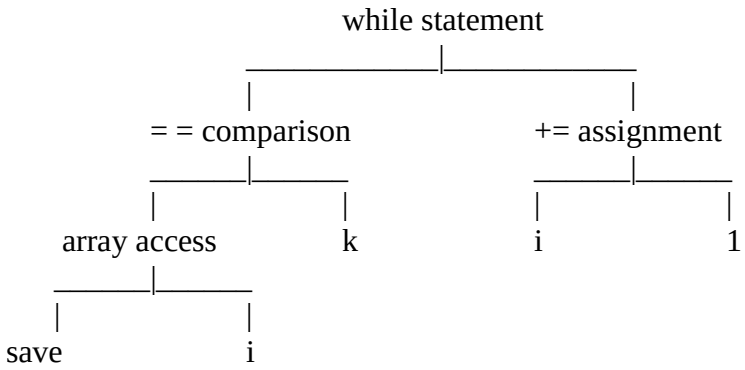
```
                                # was a zero byte
            addi $a0, $a0, 1    # increment both pointers
            addi $a1, $a1, 1

            j    strcpy         # continue loop

L1:         jr   $ra            # return to calling routine
```

Pages 2.15-1 through 2.15-14: This is a marvelous introduction to compiler optimization. It should reinforce the appreciation that you probably already have for the people who create such system software. CDA 3100 is certainly not a compiler class, but there is much to be gained from a light exposure to the topic. These few pages do a good job of introducing you to an exciting subject.

On page 2-15.4, the rightmost derivation for identifier should be a lower case "i". The text suggests that these syntax trees can be simplified by eliminating long chains of straight-line descendents. If we were to do that for the example in FIGURE 2.15.2, we get the following:



Page 2.15-6: There is an error on line 8 of the unoptimized intermediate code. The line

#

should actually be the two lines

x[i] =
li R106,x

Likewise, there is an error on line 8 of the optimized code. The line

x[i] = li R106,x add R105,R104,4

should be simply

add R105,R104,4

Page 2.15-9: In FIGURE 2.15.4, notice that lines 8-10 are a separate block from the block consisting of lines 1-7.

Page 2.15-12: You may have encountered the graph coloring problem in an algorithms class.

Page 2.15-15: This is pretty cool stuff, particularly if you have never looked into the details of Java bytecodes.

Page 2.15-16: Notice that the Java Virtual Machine (JVM) uses the stack (rather than registers) to keep track of values. Needless to say, that is one busy stack!

Page 2.15-17: You are obviously not expected to memorize this table! In most cases, however, the behavior is obvious. Note the meaning of TOS, NOS and NNOS in this chart.

Pages 2.15-18 through 2.15-20: When looking at these code examples, remember that Java arrays have two words (at the beginning) reserved for a pointer to a methods table and for the size of array.

Page 2.15-22: In FIGURE 2.15.10, the first line of class Comparable should be

```
public int compareTo (int x)
```

Page 2.15-23: You probably have gotten tired of all the bounds checks! The second line of code in FIGURE 2.15.11 should have a constant of 4, not -4.

Page 2.15-25: Notice that when using a jump table, you have to explicitly pass the return address (since a jal instruction is not being executed).

Pages 145-159: We won't be doing any programming with the ARM or the x86 instruction sets, but it is appropriate that you have some familiarity with them (since there are like a zillion computers and embedded systems out there using them). You will find that the ARM instruction set is very similar to MIPS. Section 2.17 gives a nice list of milestones in the development of the Intel x86. We are clearly not talking about a reduced instruction set here! There are a variety of addressing modes. In addition, there is considerable flexibility in the area of data transfer. Notice that there are some register use restrictions (remember that in MIPS, any of the registers may be used in an instruction that calls for a register). Instructions in MIPS are all 4 bytes in length. The x86 has variable length instructions, ranging anywhere from 1 to 17 bytes in length. By the end of this section, you should have a serious appreciation for the simplicity of the MIPS instruction set.

Page 2.21-3: You should familiarize yourself with the various architectural styles that are shown in FIGURE 2.21.1.

Page 2.21-4: Notice that stack architectures allow many instructions to be quite short in length (since the operands are on the stack and are not part of the instruction). In fact, many of the instructions consist of only an opcode. We saw some of this in the discussion on Java bytecodes.

Page 2.21-9: This is an amazingly concise, but informative, history of compilers.