LSU EE 4720 Homework 1 Solution Due: 17 February 2012

Problem 1: To save space in a program an array is designed to hold four-bit unsigned integers instead of the usual 32-bit integers (it is known in advance that their values are $\in [0, 15]$). Because this 4-bit data size is less than the smallest MIPS integer size, 8-bits, even a load byte instruction will fetch two array elements. Code to read such an array and a test routine appear on the next page, along with a stub for code to write the array. The routine `compact_array_read` is used to read an element of this array and `compact_array_write` is the start (mostly comments) of a routine to write an element.

(*a*) Add comments to `compact_array_read` appropriate for an experienced programmer. The comments should describe how instructions achieve the goal of reading from the array. The comments **should not** explain what the instruction itself does, something an experienced program already knows. See the test code for examples of good comments. *Note: The code in the original assignment had a bug:* `lb` *should have been* `lbu`*.*

See comments below.

(*b*) Complete the routine `compact_array_write`, so that it writes data into the array. See the comments for details.

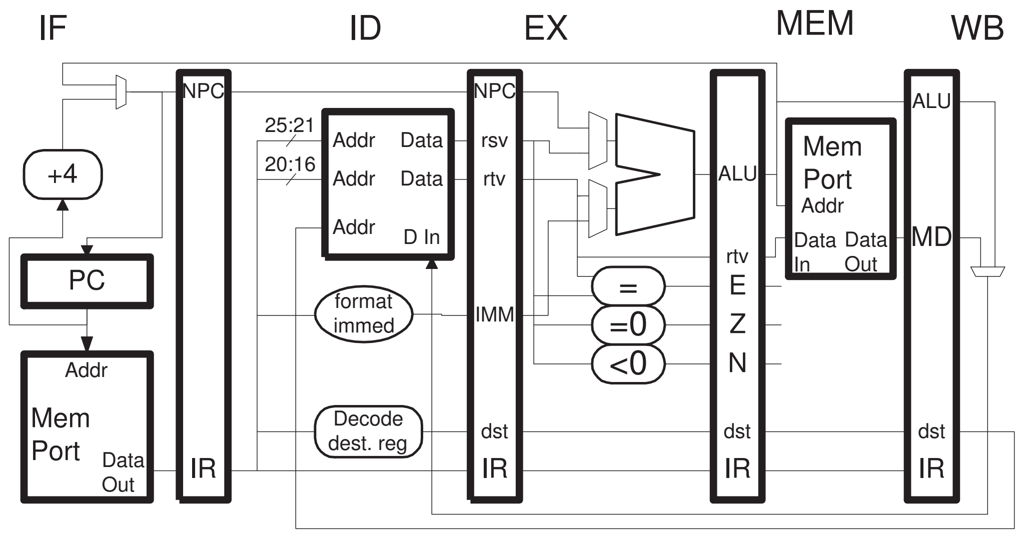
Solution appears below.

```
################################################################################
##
## Test Code
##
        .data
a:      # Array of values to test. Each byte hold two 4-bit elements.
        .byte 0x12, 0x34, 0x56

msg:    # Message format string (similar to printf).
        .asciiz "Value of array element a[%/s0/d] is 0x%/s3/x\n"

        .text
        .globl __start
__start:
        addi $s2, $0, 4    # Last index in array a.
        addi $s0, $0, 0    # Initialize loop index.
LOOP:
        la $a0, a               # First argument, address of array.
        jal compact_array_read
        addi $a1, $s0, 0        # Second argument, index of element to read.
        la $a0, msg             # Format string for test routine's msg.
        addi $s3, $v0, 0        # Move return value (array element) ...
        addi $v0, $0, 11        # ... out of $v0 and replace with 11 ...
        syscall                 # ... which is the printf syscall code.
        bne $s0, $s2 LOOP
        addi $s0, $s0, 1        # Good Comment: Advance index to next
                                #               element of test array.
                                # Bad Comment: Add 1 to contents of $s0.

        li $v0, 10              # Syscall code for exit.
        syscall
```

```
################################################################################
##
## compact_array_read
##
compact_array_read:
        ## Register Usage
        #
        # CALL VALUES
        #   $a0:  Address of first element of array.
        #   $a1:  Index of element to read.
        #
        # RETURN VALUE
        #   $v0:  Array element that has been read.
        #
        # Element size: 4 bits.
        # Element format: unsigned integer.




        ## SOLUTION
        srl $t0, $a1, 1     # Scale array index to byte offset.
        add $t1, $a0, $t0   # Compute address of element.
        andi $t3, $a1, 1    # Determine if loading upper or lower 4 bits.
        bne $t3, $0 SKIP
        lb $t2, 0($t1)      # Load a pair of elements.
        jr $ra              # Return for upper-4-bits case.
        srl $v0, $t2, 4     # Move upper 4 bits into position.
SKIP:   jr $ra              # Return for lower-4-bits case
        andi $v0, $t2, 0xf  # Just want lower bits, so mask off rest.
```

```
################################################################################
#
# compact_array_write
#

compact_array_write:
        ## Register Usage
        #
        # CALL VALUES
        #   $a0:  Address of first element of array.
        #   $a1:  Index of element to write.
        #   $a2:  Value to write.
        #
        # RETURN VALUE
        #   None.
        #
        # Element size: 4 bits.
        # Element format: unsigned integer.



        srl $t0, $a1, 1
        add $t1, $a0, $t0
        andi $t3, $a1, 1
        lb $t2, 0($t1)
        bne $t3, $0 SKIPw
        andi $a2, $a2, 0xf   # Make sure that write value is 4 bits.

        # Even element, put $a2 value in bits 7-4.
        #
        andi $t2, $t2, 0xf   # Write zeros in bits 31-4 (keep only 3-0)
        j FINISH
        sll $a2, $a2, 4      # Put write value in correct position.

SKIPw:
        # Odd element, put $a2 in bits 3-0.
        #
        andi $t2, $t2, 0xf0  # Write zeros everywhere except bits 7-4.
FINISH:
        or $t2, $t2, $a2     # Combine write value with value already present.
        jr $ra
        sb $t2, 0($t1)
```

Problem 2: The MIPS code below executes on the illustrated implementation. The loop iterates for many cycles. The register file bypasses data from the write ports to the read port in the same cycle.



```
LOOP: # 1st Iter   0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 srl r4, r3, 2     IF ID EX ME WB
 sw r4, 0(r3)         IF ID ----> EX ME WB
 bne r4, r2 LOOP         IF ----> ID EX ME WB
 addi r3, r3, 4                   IF ID EX ME WB
 nop                                 IF ID EXx
 nop                                    IF IDx
# Second Iteration 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 srl r4, r3, 2                             IF ID EX ME WB
 sw r4, 0(r3)                                 IF ID ----> EX ME WB
 bne r4, r2 LOOP                                 IF ----> ID EX ME WB
 addi r3, r3, 4                                           IF ID EX ME WB
# Third Iteration  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 srl r4, r3, 2                                                     IF
```

(*a*) Show a pipeline execution diagram for the code above on the illustrated implementation for enough iterations to determine CPI.

Solution appears above. The implementation lacks bypass paths, forcing the store to stall two cycles, waiting for the result of the `srl`. Also notice that in this implementation the branch resolves in ME and so the branch target is not fetched until the branch is in WB. *Grading Note: In most submissions the branch target was fetched one cycle too early.*

The second iteration starts at cycle 8, the third at cycle 16. The state of the pipeline is identical in cycles 8 and 16 (`addi` in ME and `bne` in WB), and so the third iteration will execute identically to the second. The time for the second iteration is $16 - 8 = 8$ cycles, so the third and subsequent iterations will be 8 cycles. The CPI is then $\frac{8}{4} = 2\,\text{CPI}$.