CS 2734, Org II, Fall 2000
Final Exam, Selected and Partial Answers

1.

```
    # CS 2734, Computer Organization II, Fall 2000
    # MIPS program giving answer to Final, question 1
            .globl main
    main:       addu    $s7, $zero, $ra
    ########## Start of answer to Question 2 ################
            .data
    A:  .space 40
            .text
            la      $s0, A          # address of A
            addi    $t0, $0, 0      # loop counter, start at 0
            addi    $t1, $zero, 10  # to terminate loop
            add     $t2, $0, $s0    # address of item in A
    Loop:   sw      $t0, 0($t2)     # store current $t0 into A
            addi    $t0, $t0, 1     # increment loop counter
            addi    $t2, $t2, 4     # increment pointer into A
            bne     $t0, $t1, Loop  # branch back to form loop
    ########## End of answer to Question 2 ##################
        ## Print the array
            la      $a0, A
            li      $a1, 10
            jal     Write_array
            jal     Newl
    ########## Finish main###################
            addu    $ra, $zero, $s7
            jr      $ra
    ########## write an array ###############
    Write_array:
            addi    $sp, $sp, -4    # room for $ra on stack
            sw      $ra, 0($sp)     # save $ra because not leaf
        ## initialization for loop
            move    $s0, $a0        # $s0 = $a0 = start of A
            move    $s1, $a1        # $s1 = $a1 = N
            move    $t1, $zero      # start $t1 = 0, the index
    LoopA:  beq     $s1, $t1, EndA  # if (N == index) goto EndA
        ## write value for A[i]
            addu    $t2, $t1, $t1
            addu    $t2, $t2, $t2
            addu    $t2, $s0, $t2   # $t2 = index*4 + start of A
            li      $v0, 1
            lw      $a0, 0($t2)     # integer to print
            syscall
        ## write a blank
            jal     Blan
            addi    $t1, $t1, 1
            j       LoopA
    EndA:   lw      $ra, 0($sp)
            addi    $sp, $sp, 4
            jr      $ra
    ########## write newline ################
    Newl:   li      $v0, 4
            la      $a0, Newline
            syscall
            jr      $ra
    ########## write blank ##################
    Blan:   li      $v0, 4
            la      $a0, Blank
            syscall
            jr      $ra

            .data
    Blank:      .asciiz  " "
    Newline:   .asciiz  "\n"
```

```
    ############### output ###############################
    # four06% spim -file quiz4.s
    # 0 1 2 3 4 5 6 7 8 9
    ######################################################
```

Alternatively, the following code uses the mul pseudo-instr:

```
    ########## Start of answer to Question 2 ################
            .data
    A:  .space 40
            .text
            la      $s0, A          # address of A
            addi    $t0, $0, 0      # loop counter, start at 0
            addi    $t1, $zero, 10  # to terminate loop
    Loop:   mul     $t2, $t0, 4     # pseudo-instr, mult $t0 by 4
            add     $t3, $t2, $s0   # add to start addr of A
            sw      $t0, 0($t3)     # store current $t0 into A
            addi    $t0, $t0, 1     # increment loop counter
            bne     $t0, $t1, Loop  # branch back to form loop
    ########## End of answer to Question 2 ##################
```

2.

```
##### CS 2734, Final Exam, Problem 2 #######
        .globl main
main:   addu    $s7, $zero, $ra

########## MAIN FOR PROB 2 #################
        addi    $a0, $0, 12     # First param = 12
        addi    $a1, $0, 45     # Second param = 45
        jal     Addup           # call Addup
        add     $a0, $0, $v0    # $v0 = ret val
        li      $v0, 1          # print it
        syscall
        jal     Newl            # print newline

# Finish main
        addu    $ra, $zero, $s7 # normal end of main
        jr      $ra             # return to system
########## END OF MAIN ####################

########## PROB 8, function Addup ##########
Addup:
        add     $v0, $a0, $a1
        jr      $ra
########## END OF FUNCTION Addup ###########

########## write newline ################
Newl:
        li      $v0, 4
        la      $a0, Newline
        syscall
        jr      $ra
########## DATA ##########################
        .data
Newline: .asciiz "\n"
########################################
# Output:
#   57
########################################
```

---

3. Just the standard lw diagram for the multi-cycle implementation.

---

4. See pages 477 and 481. Considering the lines from left to right, the first two lines require data forwarding controlled by the forwarding unit (one forwarded from the next cycle, and the other forwarded from two cycles down.) The third line

is handled by having the Register File write in its first half cycle and read in its second half cycle, so the data value is already written by the time it needs to be read. The final line goes forward in time and is not a hazard at all.

---

5. Refer to the material before the diagram on page 487. In the top diagram, the instruction "or $4, $4, $2" needs the results of two previous instructions, namely the result of the "add $4, ..." that will be in $4, which is forwarded back into the ALU from Stage 4, and the result of "sub $2, ..." which is forwarded back into the ALU from Stage 5. The register values are always available, but the forwarding unit processes the register _numbers_, and sets control signals so that a multiplexor picks off the forwarded register value.

In the bottom diagram, the instruction "add $9, $4, $2" needs the value of the register $4, which has been calculated by the next instruction "or, $4, ...", and is forwarded from Stage 4. Note that there is another computed value for $4, calculated one cycle earlier and available from Stage 5, but the forwarding unit will not use this earlier value.

---

6. A stall is needed after the lw instruction. See Section 6.5. The Hazard Detection Unit notices that a lw instruction is in Stage 3 (by checking that the MemRead flag is 1). It also checks that the result of the lw is in a register needed by the next instruction. In this case it inserts a 1-cycle stall, by inserting zeros in for the control signals, and by deasserting the IF/IDWrite control line, so that nothing is writing into the IF/ID on that cycle. It also deasserts the PCWrite signal, so that nothing is written into the PC. Thus the next instruction is _not_ written into IF/ID until IF/IDWrite is asserted again, when the flow of instructions can start up. Also the PC value is not updated until the next cycle.

Thus a "bubble" is created in the sequence of instructions going through. Two cycles later (see Fig. 6.48) when the lw instruction is in its final stage and when the following and instruction is in its execute stage, the value of $2 must be forwarded into the ALU for use by the and instruction, using the Forwarding unit as for other data hazards. At the same time, lw finishes loading the new value into $2.

---

7. There is a stall on beq in case of a successful branch. One moves beq into cycle 2 to allow execution at the branch target instruction with only one cycle stall. (Otherwise, one would need 2 or more cycles of stall.) The new hardware in cycle 2 is an adder to calculate the branch address, and to add a comparer to compare the two branch registers for equality. The result of the comparer will set a control line IF.Flush to turn the newly fetched instruction into a nop by zeroing the IF/ID pipeline register.

---

8. We talked about three kinds of coding:
    source (using compression)
    channel (using error detection/correction)
    secrecy (using cryptography)

Bit positions 0 (overall parity), 1, 2, 4, 8, and 16 are used as check bits, if there are 24 bits altogether (plus the 0th bit). The bit in position 1 is used to check the parity of the odd-numbered bits. The full Hamming code gives single error correction and double error detection.

---

9. The exception handler gets invoked automatically in case of an exception (internal unusual event, such as overflow or undefined instruction) or an interrupt (external unusual event, such as an I/O device wanting service). mfc0 has coprocessor 0 move the EPC (in coprocessor 0 register $14) into ordinary register $k0 (used by the kernel). rfe resets the status register so that the program will no longer execute in supervisory mode. The addiu adds 4 to the EPC value, so that we return to the instruction after the offending one. jr $k0 does the actual return.

---

10. (a) 14 bits for the index means 2^14 entries = 16K words = 64K bytes of data in the cache.
(b) The address should be a word address, so the low order 2 bits will always be 0s.
(c) These are the remaining bits of the address, 32 - 14 - 2 = 16. After we know that bits 1-0 are 0s, that bits 15-2 match because they are the same index entry, we need to check the remaining 16 bits 31-16 with the 16 bits in the tag field to see that the addresses are exactly the same.
(d) The hardware uses bits 15-2 as an index into the cache to directly access a word, with no searching. If bits 31-16 match the Tag field and if the Valid bit is on, there is a hit.
(c) See the four items at the bottom of page 551.