CS 2734, Org II, Spring 2001
Final Exam, Selected and Partial Answers

1.

```
# Answer to Final Exam, Problem 1
        .globl  main
main:   add     $s7, $0, $ra    # save return address

        .data
  A:    .word  2, 4, 6, 8, 10, 12, 14, 16, 18, 20
        .text
################# Answer to Problem 1 ##########################
        la      $s1, A          # start address of A
        addi    $s2, $0, 0      # running sum
        addi    $s3, $0, 0      # array index of A
        addi    $s4, $0, 10     # constant 10

 Loop:  lw      $t1, 0($s1)     # $t1 = A[$s3]
        add     $s2, $s2, $t1   # $s2 = sum of A[] so far
        addi    $s1, $s1, 4     # $s1 += 4
        addi    $s3, $s3, 1     # $s3 += 1
        bne     $s3, $s4, Loop  # brach back to Loop until $s4 == 10

        addi    $v0, $0, 1      # print the sum
        add     $a0, $0, $s2
        syscall
################# End of answer to Problem 1 ###################
        addi    $v0, $0, 4      # print a newline
        la      $a0, Newln
        syscall

        add     $ra, $0, $s7    # restore return address
        jr      $ra
        .data
Newln:  .asciiz "\n"
################# Output #######################################
four06% spim -file exam1_2.s
110
```

Another anwser:

```
################# Second Answer to Problem 1 ###################
        la      $s1, A          # start address of A
        addi    $s2, $0, 0      # running sum
        addi    $s3, $0, 0      # array index of A
        addi    $s4, $0, 10     # constant 10

 Loop:  mul     $t0, $s3, 4     # $t0 = array index * 4
        add     $t2, $t0, $s1   # $t2 = start of A + offset
        lw      $t1, 0($t2)     # $t1 = contents at start of A + offset
        add     $s2, $s2, $t1   # $s2 = sum of A[] so far
        addi    $s3, $s3, 1     # $s3 += 1
        bne     $s3, $s4, Loop  # brach back to Loop until $s4 == 10
```

```
        addi    $v0, $0, 1      # print the sum
        add     $a0, $0, $s2
        syscall
################# End of answer to Problem 1 ###################
---------------------------------------------------------------------------
2.
##### CS 2734, Final Exam, Problem 2 #######
        .globl main
main:   addu    $s7, $zero, $ra

########## MAIN FOR PROB 2 #################
        addi    $a0, $0, 12     # Param = 12
        jal     F               # call F
        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 2, function F ############
F:
        add     $v0, $a0, $a0
        jr      $ra
########## END OF FUNCTION F ##########

########## write newline ################
Newl:
        li      $v0, 4
        la      $a0, Newline
        syscall
        jr      $ra
########## DATE #########################
        .data
Newline: .asciiz "\n"
#########################################
# Output:
#   24
#########################################
---------------------------------------------------------------------------
3. This instruction is like the first part of lw or sw
and the last part of add.  No additional data lines
or control lines are needed.

Fetching the instruction, updating the PC, and fetching
registers are the same as for all instructions.

The ALU does the same as for lw or sw:
```

ALUresult = (Output of Read data 1) + sign-extend(IR[15-0]);
The control settings are the same as for those instructions:
ALUSrc = 1, ALUOp = 00. (So that input to ALU becomes 010 (add).)

For the rest, we have to route the ALUresult around back into the register file, using IR[20-16] for the register.
(Note that add has Reg[IR[15-11]] = ALUresult)
Thus we need RegWrite = 1, MemtoReg = 0, and RegDest = 0.
(Note that add needs RegDest = 1, because the destination register is in bits 15-11 for add, while it is bits 20-16 for addi.)

---

4. Just the standard beq diagram for the multi-cycle implementation.

---

5. (b) In executing the second instruction (and), the value computed for $2 by the sub must be forwarded into the ALU. The need for this is detected by the forwarding unit, and the actual forwarding is carried by asserting a control line to a mulitplexor in front of the ALU.

---

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 written 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. (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.

---

9. When the clock signal is asserted (rising clock edge), the value of D (asserted) goes through the first D latch, but waits at the second D latch until the clock deasserts (falling clock edge). At this point the value of D gets all the way through the flip-flop

---