CS 2734, Computer Organization II
Fall Semester, 2000
Final Examination

1. Consider the following MIPS code fragment: **(20)**

```
        .data
A:      .space 40
        .text
# insert MIPS instructions here.
```

   For insertion at the comment, write MIPS instructions that will do the following:

   (a) Create a loop of 10 iterations that will let the register `$t0` take on values 0, 1, ..., 9.

   (b) Store the values of `$t0` into successive words of the array `A`, so that if we printed the 10 locations of `A`, we would print out the numbers 0, 1, ... 9.

   (You should not include code to do this printing. Your MIPS code should do what is asked for above and *nothing more*.)

2. Write portions of MIPS assembler language that will call a function **`Addup`** with parameters **`12`** and **`45`**. Then your code should store the returned value in a variable **`i`** corresponding to register **`$s3`**. You should give the complete code for **`Addup`**, which should add its two arguments and return the result. Your code should not invoke any syscall, and it should not have any of the rest of the main function except the call to **`Addup`** and the storing of the returned value into **`$s3`**. The code for **`Addup`** should be complete. *For full credit, you must follow the MIPS conventions for following for passing parameters and for returning a value from a function.* Thus you should be implementing the following C code: **(20)**

```
/* in main */
i = Addup(12, 45);  /* use $s3 for i */
. . .

int Addup(int a, int b)
{
    return a + b;
}
```

3. For this problem, you are to use one or more xeroxes of Figure 5.33 (final datapath for the *multi*-cycle implementation of MIPS). You will be tracing through the path of the **`lw`** instruction on this multi-cycle model. You should use several colors of highlighters to trace the paths the instruction and associated data takes through the diagram. (Or you can trace these paths using more than one diagram.) Do *not* show data traveling to "dead-end" components, which will eventually have no effect. In particular, do not show the computation of the branch address, which will not be used in this case. **(20)**

   For this diagram, do *not* give the values of control signals.

   Below the diagram, or in some other way, carefully identify *which cycle* (or step) of handling the instruction belongs to each part of the highlighted datapath (just for data, not control). Thus you should identify *Cycle 1*, *Cycle 2*, *Cycle 3*, and perhaps *Cycle 4* and *Cycle 5* (if the instruction uses Cycles 4 and 5).

   Use the following specific instruction:

```
lw $t5,92($t1)
```

or in machine language form:

```
0x8d2d005c                              (in hexadecimal)
100011 01001 01101 00000 00001 011100 (fields in binary)
    35     9    13                92 (fields in decimal)
```

Start at the left side, showing the PC coming in, and assume this instruction is read from the Instruction memory. *Be sure to identify the different cycles.* Don't forget the PC. Show what values are traveling along the different lines, assuming the following initial values:

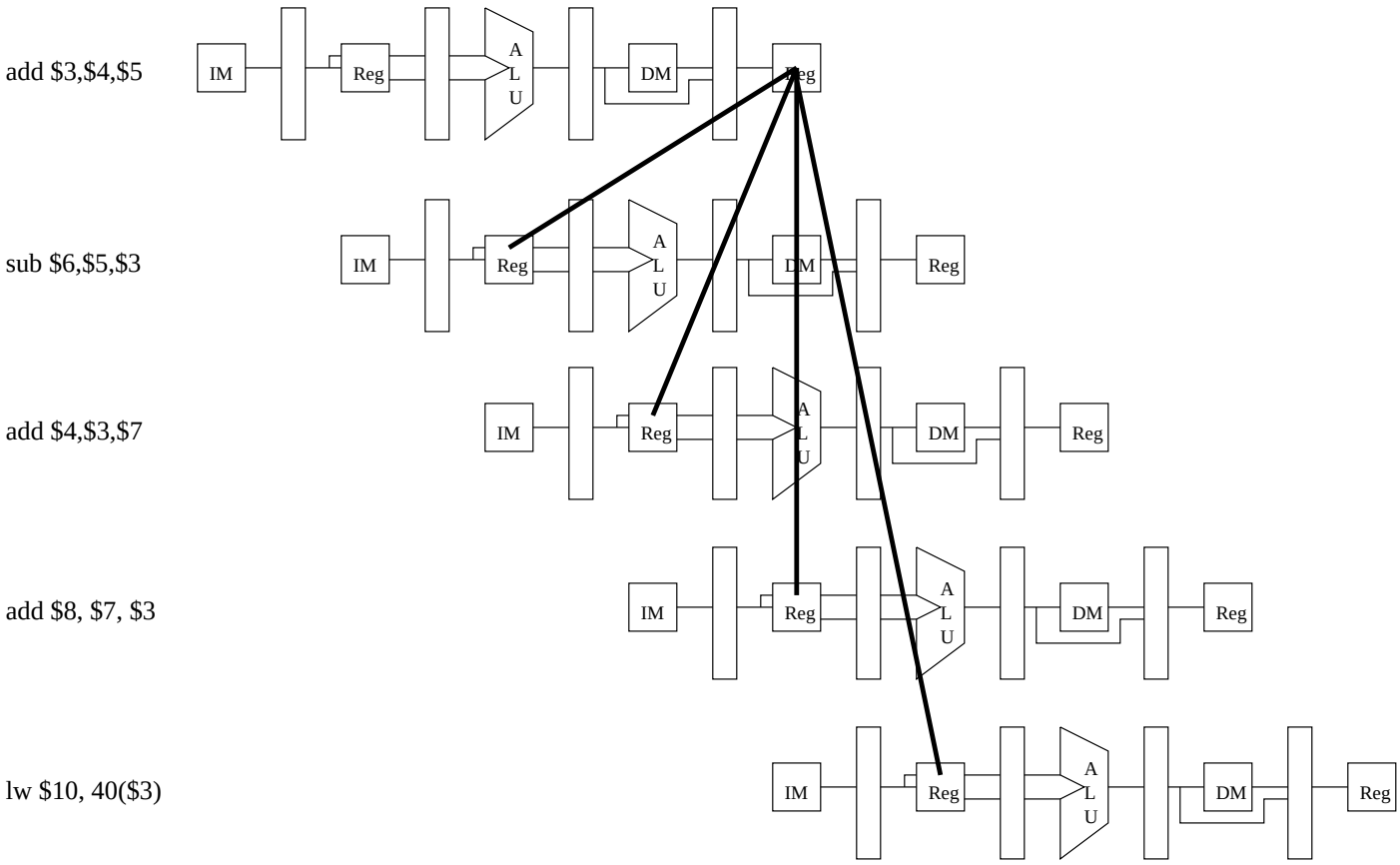
(a) **`$t1`** and **`$t5`** are registers numbers **`9`** and **`13`** (decimal), respectively.

(b) The contents of register **`9`** is **`200`** (decimal).

(c) The PC has value **`16`**.

4. The following diagram shows the pipelined execution of five instructions. Instructions 2 through 5 depend on the result (**`$3`**) of Instruction 1. In each case, the wide line shows the dependency. **(15)**

   Explain how each of the four dependencies is handled. In cases where forwarding is used to handle the dependency, use a highlighter to show the forwarding step. If there is no need for special forwarding by the hardware, explain why no forwarding is needed. (So for each instruction after the first, either draw a forwarding line, or explain how the dependency is handled without forwarding.)



5. Consider the xerox of Figure 6.42 from your text, showing pipelined execution where forwarding is required. There are two diagrams, an upper one and a lower one, showing successive cycles. **(20)**

   The pipelined MIPS machine is in the middle of executing the following sequence of instructions:

```
sub     $2, $1, $3
and     $4, $2, $5
or      $4, $4, $2
add     $9, $4, $2
```

   (a) In the upper diagram, there are two instances of forwarding. What register values are being forwarded, from where to where, and in support of which instruction?

   (b) In the upper diagram, how does the hardware decide which register values to forward? How does it get the proper data fed into the ALU?

   (c) In the lower diagram, what forwarding is occurring, and how is it managed by the hardware?

6. Consider the xerox of Figure 6.47 from your text, showing pipelined execution, where a *stall* is needed. The instructions in execution are: **(20)**

```
lw      $2, 20($1)
and     $4, $2, $5
or      $4, $4, $2
add     $9, $4, $2
```

   (a) After which instruction is a stall needed? Why is a stall needed in this spot? How many cycles must the stall last?

   (b) What unit detects the need for a stall, and how does it know a stall is needed?

   (c) In order to actually carry out a stall, two separate (distinct) actions must be taken. Describe them in some detail.

   (d) Describe briefly what value is forwarded from the lw instruction and how it is forwarded to where it is needed.

7. Consider the xerox of Figure 6.52 from your text, showing pipelined execution, where a *stall* is needed. The instructions in execution are: **(20)**

```
sub     $10, $4, $8
beq      $1, $3,  7 # branches to lw below
and     $12, $2, $5
or      $13, $2, $6
add     $14, $4, $2
. . .
lw       $4,  50($7)
```

   The figure assumes the branch is taken, that is control transfers from the **`beq`** to the **`lw`** instruction.

   (a) What hardware changes does the book show to limit a stall on **`beq`** to just 1 cycle (if taken) and to avoid a stall (if not taken)?

(b) Using Figure 6.52, describe how the stall is inserted.

8. (a) With a word or two, describe each of the three kinds of coding that were mentioned in class: *source* coding, *channel* coding, and *secrecy* coding. **(15)**

   (b) Suppose we have a *Hamming code* with 24 bits, in positions numbered 1, 2, 3, ..., 24.

      i. Which of these bits are used as check bits and which are used as data bits?

      ii. Which bit positions are checked by the first check bit?

      iii. What does this Hamming code do in terms of error detection/correction? (Assume the extra overall parity check is included with the extra 0th check bit.)

9. Consider the exception (or trap) handlers described in the book and in class. **(15)**

   (a) Under what circumstances is the code for the exception handler executed?

   (b) The final few lines of the handler might be as follows. Say what each of these lines is doing.

```
mcf0     $k0, $14
rfe
addiu    $k0, $k0, 4
jr       $k0
```

10. Consider the xerox of Figure 7.8 from the text, showing the diagram for the cache memory of a DECStation 3100. **(15)**

   (a) What is the size of the field used for an *index* (that is, for a cache table entry)? From this, say how many words of data this cache holds.

   (b) Why does this hardware not pay attention to the low order 2 bits?

   (c) Why is the *Tag* field in the cache 16 bits wide?

   (d) Describe the process that occurs with a cache *hit*. How does the hardware know there is a hit? (Be careful, there are *two* conditions.) How does the hardware manage to get to the correct data without some kind of search?

   (e) Describe what the text says happens on an instruction cache *miss*. (Four steps.)