CS 2733/2731, Computer Organization II
Spring Semester, 2003
Final Examination

1. Consider the following MIPS code fragment: **(15)**

```
      .data
# stored in A are squares of first 7 primes
A:    .word  4, 9, 25, 49, 121, 169, 289
      .text
# insert MIPS instructions here.
```

For insertion at the comment, write a *single* MIPS program that will do the following:

(a) Put the starting address of `A` into register `$s1`.

(b) Inside a loop, access each element of `A` and add these values, leaving the result in register `$s2`. [You must use a loop for this.]

(c) Print the resulting sum, using `syscall`. [Recall that `syscall` requires `$v0` equal to 1 to print the value in `$a0`.]

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 **`F`** with parameter **`12`**. Then your code should store the returned value in a variable **`i`** corresponding to register **`$s3`**. You should give the complete code for **`F`**, which should add its argument to itself and return the result. Your code should not invoke any syscall, and it should not have any of the rest of the main function except the call to **`F`** and the storing of the returned value into **`$s3`**. The code for **`F`** should be complete. *For full credit, you must follow the MIPS conventions for passing parameters and for returning a value from a function.* (You do not need to explicitly save any registers on the stack.) Thus you should be implementing the following C or Java code: **(15)**

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

int F(int a) {
    return a + a;
}
```

---

3. For this problem, you are to use a xerox of Figure 5.29 (final datapath for the *single*-cycle implementation of MIPS). You will be tracing through the path of the **`beq`** instruction on this single-cycle model. You should use a highlighter to trace the path the instruction and associated data takes through the diagram. (Do *not* show data traveling to "dead-end" components, **(25)**

which will eventually have no effect.) Indicate the values of *relevent* control signals, without highlighting the control line. (Do *not* give control signals that serve to keep "dead-end" paths from having an effect.)

Use the following specific instruction:

```
beq   $t2, $t5, Loop
```

or in machine language form:

```
0x114d0008                            (in hexadecimal)
000100 01010 01101 00000 00000 001000 (fields in binary)
     4    10    13                  8 (fields in decimal)
```

Start at the left side, showing the PC value coming in, and assume this instruction is read from the Instruction Memory. Show what values are traveling along the different lines, assuming the following initial values:

(a) **`$t2`** and **`$t5`** are register numbers **`10`** and **`13`** (decimal), respectively.

(b) Assume that the contents of each of these registers is **`5234`**, so you should assume that the branch is taken. (The branch will be taken because the two register values are equal.)

(c) Assume the PC has value **`20`** (decimal) initially. On the proper line, give the *final* PC value, assuming the branch is taken. Don't forget to highlight the parts related to the PC as well as the rest of the instruction.

---

4. 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. **(25)**

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`**.

---

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
```

Please use a pen and/or marker to show answers directly on Figure 6.42.

(a) The above instructions require *four* instances of data forwarding. Mark the instructions above with circles and arrows to show exactly which registers are being forwarded to which instructions.

(b) In the upper diagram, use a **`B`** to mark *all* the lines having to do with forwarding from the **`and`** instruction.

(c) In the upper diagram, use a **`C`** to mark *all* the lines having to do with forwarding from the **`sub`** instruction.

(d) In the lower diagram, use a **`D`** to mark *all* the lines having to do with the forwarding step that is occurring, either from the **`and`** or the **`or`** instruction.

(e) In the lower diagram, explain which register is begin forwarded, from which instruction. (The lines used for control and data should have been marked with a **`D`**.)

---

6. Consider the xerox of Figure 6.52 from your text, showing pipelined execution, where a *stall* is needed. The instructions in execution are: **(15)**

```
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. Please use a pen and/or marker to show answers directly on Figure 6.47.

(a) On the lower figure, mark with a **`A`** the portions of the hardware that were introduced to make the **`beq`** finish in two cycles

(b) On the upper figure, mark with a **`B`** the lines used to insert a stall.

(c) What form does the stall take?

---

7. This problem deals with the *Hamming Code*. **(15)**

(a) What error detection and correction is the full binary Hamming code capable of?

(b) Suppose you have the following six data bits: **`0 1 1 0 0 1`**. Insert the proper values for the check bits in the correct positions. (Do not use position **`0`**.)

(c) Suppose the bit in position **`6`** is transmitted in error. Show how the Hamming code will be able to correct this error.

---

8. This problems deals with *exceptions* in MIPS. **(15)**

(a) For the hardware described in the text, what two conditions were handled?

(b) As the exception handler (or trap handler) is written, if an instruction causes an exception, what instruction is put into execution after handling the instruction that caused the exception?

(c) In the pipelined model, list the important things that must be done in case of an exception.

---

9. Consider the xerox of Figure 7.7 from the text, showing the diagram for cache memory. **(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 20 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.)