CS 2733/2731, Computer Organization II
Spring Semester, 2004
Final Examination

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

```
        .data
# stored in A are the first 10 prime numberss
A:      .word  2, 3, 5, 7, 11, 13, 17, 19, 23, 29
        .text
# insert MIPS instructions here.
```

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

(a) Read an integer **`i`** and load it into register **`$s0`**. (Recall that to read an integer, you use a **`syscall`** with a **`5`** in register **`$v0`**. The integer read pops into **`$v0`**, overwriting the **`5`** that was there.)

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

(c) Store a **`0`** into the **`i`**th location of **`A`**. (Thus if **`i == 5`**, the **`0`** should overwrite the **`13`** in the array **`A`**.)

(d) Use a loop to add the values of **`A`** up to the new **`0`** element, leaving the result in register **`$s2`**. (You must use a loop for this.)

(e) 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*. In particular you should not print a final newline or blank.

---

2. Write portions of a *single* MIPS assembler language program that will call a function **`PrintSum`** with parameters **`55`** and **`89`**. The function **`PrintSum`** should add its parameters and print the result. It should then call a function **`PrintNewLine`** whose code is given below. You should give the complete code for the function **`PrintSum`** and for the call to **`PrintSum`**. Because **`PrintSum`** calls another function, it must save the return address on the stack at the start and must restore the return address at the end (and it should restore the stack as well). Thus you should be implementing the following C code (almost the same as Java, except for the **`printf`** function): **(15)**

```
void PrintSum(int a, int b) {
   int c = a + b;
   printf("%i", c);
   PrintNewLine();
}
void PrintNewline(void) {
   printf("\n");
}
. . .
/* in main */
int x = 55, y = 89;
PrintSum(x, y);
. . .
```

Assume that the function **`PrintNewline`** is implemented with:

```
PrintNewline:
      la     $a0,   Newl
      addi   $v0,  $0, 4
      syscall
      jr     $ra
     .data
Newl:  .asciiz "\n"
```

---

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 **`lw`** 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, which will eventually have no effect.) Then write in the values for the *relevant* control signals. (Do *not* give control signals that serve to keep "dead-end" paths from having an effect.) **(25)**

   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. Don't forget the handling of the PC by this instruction. 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 contents of memory at location **`292`** is **`144`** (decimal).

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

   (Don't forget to handle the PC as well as the rest of the instruction.)

(30)

4. For this problem you will be including an additional instruction into the *multi*cycle datapath described in the text, an instruction not included in the examples in the text. The new instruction is **`addi`** (add immediate).

   You are to use one or more xeroxes of Figure 5.33 (the final datapath for the multicycle implementation of MIPS). You should decide how many cycles to use (three, four, or five). No additional datapaths are necessary, but you need to determine the control signals for each cycle.

   Use a highlighter to show the data lines traversed during each cycle, labeling the data line with the cycle number. *You must show the relevant control line values needed for each cycle.*

   Specifically, suppose the instruction is **`addi $t3, $t2, 45`**, which in machine language form is:

```
214b002d                            (in hexadecimal)
001000 01010 01011 0000000000101101 (fields in binary)
     8    10    11               45 (fields in decimal)
```

   In marking the values on data lines, assume that **`$t2 = $10`** holds the value **`51`**.

---

(20)

5. Consider the xerox from your text (bottom part of Figure 6.42, page 487), showing one stage of pipelined execution where forwarding is required.

   Please use a pen and/or marker to show answers directly on the xerox. There is only *one* instance of forwarding here.

   (a) Identify exactly what register is being forwarded, what stage it is forwarded *from* and what stage it is forwarded *to*. (The pipeline stages are: IF, ID, EX, MEM, and WB.)

   (b) Carefully highlight the line that is carrying the actual 32-bit quantity that is being forwarded. Label this line clearly with the letter **`A`**.

   (c) Identify the other forwarding line that would be used for forwarding if it were not for the forwarding that is occurring as described and highlighted in part (b) above. Carefully highlight this line (that could have been used to forward a 32-bit quantity, but is not in fact being used here). Label this line clearly with the letter **`B`**.

   (d) There are various lines into the Forwarding Unit that carry 5-bit register numbers. Carefully highlight these 5-bit lines. Label each line clearly with the letter **`C`**.

   (e) There are one or more 1-bit control lines coming into the Forwarding Unit. Carefully highlight these 1-bit lines. Label each line clearly with the letter **`D`**. What is the purpose of these lines?

   (f) Finally, there is a control line that is making the correct forwarding occur. Carefully highlight this line. Label it clearly with the letter **`E`**.

(20)

6. Consider the xerox of Figure 6.52 from your text, showing pipelined execution. The instructions in execution are:

```
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 6.52.

(a) On the upper figure, carefully highlight the line that is carrying the new branch address, the address at which instruction will start on the next cycle in the lower figure. Label this line clearly with the letter **A**.

(b) On the upper figure, carefully highlight the line that is providing the values used for the bubble that will be in place during the next cycle in the lower figure. Label this line clearly with the letter **B**.

(c) What kind of a bubble is in pipeline stage ID in the lower figure?

---

(15)

7. Consider the xerox of Figure 6.56 from your text, showing how an exception is handled during pipelined execution. (This figure is for reference, and you don't need to draw on it.)

(a) What is the exception? (Only one was discussed at this point in the book.)

(b) In which pipeline stage is the exception occurring?

(c) Three bubbles are shown. What makes these bubbles work as bubbles, that is, why do they have no effect on execution? (Two different answers for the two types of bubbles.)

(d) Where did the instruction at the left in the bottom figure come from? What is now happening with the computer?

---

(15)

8. Consider the xerox of Figure 7.8 from the text, showing the diagram for cache memory.

(a) What is the size of the field used for an *index* (that is, how many bits give the location of the entry in the cache)? 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) Consider 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.)