EE 468/568: Microcomputers - 2

Project # 1

Due Date: Monday May 17, 2010 by 11:59 PM EST

Instructions:
1) Use WinMIPS64 to solve all the problems for this project.
2) Submit your code <lastname.p1.s> and <lastname.p2.s> for the first two problems. For problem 3, submit lastname.p3.pdf i.e. I do not want an executable for the last problem. Write a complete report on all your observations and the associated code within the PDF document. Submit all three files to kodi@ohio.edu. In addition, please drop a hardcopy of the simulation code and word document into my mailbox.

Problem 1 (35 Points)

Write a WinMIPS64 code to determine whether a given number is an Armstrong number.

The *n*-digit numbers equal to sum of *n*th powers of their digits (a finite sequence), called Armstrong numbers. They first few are given by 1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407, 1634, 8208, 9474, 54748,….

For example, consider 371, $371 = 3^3 + 7^3 + 1^3 = 27 + 343 + 1 = 371$. A sample C code for your assistance is given below:

```
#include <stdio.h>

int main(){
int num,i;
float digit,sum,capture;
sum = 0.0;
capture = 0.0;
printf( "Enter the total number of digits\n" );
scanf( "%d", &num );

for( i = 0; i < num; i++ ) {
        printf( "Enter the %d digit:\n", i+1 );
        scanf( "%f", &digit );
        sum = sum + (digit*digit*digit);
        capture = capture * 10 + digit;
}
printf( "Entered Number: %f\n", capture );
printf( "Sum of Product: %f\n", sum );
if( capture == sum )
        printf( "Armstrong number" );
else
        printf( "Not an Armstrong number" );
}
```

All examples can be used for manipulation. Note that the numbers displayed are in floating point. Display the numbers using double format in WinMIPS64. You need to test the code only for 1-4 digit numbers from 0 to 9999. It is sufficient to follow the mentioned C logic that checks for 4-digit Armstrong numbers, even though more digits can be specified. Provide a complete report of your code i.e. stall count, code size, CPI.

Problem 2: (30 Points)

Fibonacci numbers are: 0,1,1,2,3,5,8,13,21,34,55,etc i.e. two consecutive numbers are added to generate the next number. If the user enters 6, then the display should be 0, 1, 1, 2, 3, 5 i.e. 6 fibonacci numbers should be displayed. Generate the numbers and display it. The sample C code is:

```
#include <stdio.h>

main()

{

      int x,y,z,num,count;
      printf( "Enter the number of fibonacci numbers > 2:\n");
      scanf("%d",&num);
      x = 0;
      y = 1;
      count = 2;
      printf( "The numbers are:\n");
      printf("%d\n%d\n",x,y);
      while(count<num)
      {

            z = x + y;
            printf( "%d\n",z);
            x = y;
            y = z;
            count++;

      }

}
```

Again all examples can be used for manipulation. Note that the numbers displayed are integers. Provide a complete report of your code i.e. stall count, code size, CPI.

Problem 3 (20 Points)

Consider the following code and answer the following questions. Note that F2 register holds a scalar constant that cannot be changed for the computation (see MUL.D instruction)

```
      .data
      .text
main:

      DADDI      R3,R0,8
      DADDI      R1,R0,1024
      DADDI      R2,R0,1024
Loop: L.D        F0,0(R1)
      MUL.D      F0,F0,F2
      L.D        F4,0(R2)
      ADD.D      F0,F0,F4
      S.D        F0,0(R2)
      DSUB       R1,R1,R3
      DSUB       R2,R2,R3
      BNEZ       R1,Loop

      HALT
```

(a) Enable forwarding (check under the *Configure* tab). Run the code. How many stalls do you see? Can you identify where these stalls occur (the pair of instructions) that cause this stall. Hint: Run in Single Cycle mode using F7. What is the CPI?

(b) Execute the code by enabling *Enable Branch Target Buffer* (check under the *Configure* tab). How many stalls do you see? How many stalls do you see and what exactly does the *Enable Branch Target Buffer* do? What is the CPI and what is the speedup when compared to (a)?

(c) Execute the code by enabling *Enable Delay Slot* (check under the *Configure* tab). You will need to put one instruction to be executed, else HALT instruction will stop the code from executing. What is the CPI and the speedup compared to (a). Which scheme is better, branch target buffer or delay slot?

(d) Re-arrange the loop without unrolling. You can move individual instructions, however the output of this dummy loop should be exactly the same i.e. adjust the offset for memory instructions (load/store). Can you reduce the stalls for this code? What is the new CPI and the speedup when compared to (a)?

(e) Now, transform the loop by unrolling the loop, reschedule the instructions, enable delay slot or branch target buffer to completely minimize the stalls. What is the CPI and what is the speedup when compared to (a)?