Talal Javed Qadri and Jonathan Reshef
Final Project Write Up

Helicopter

For our final project, we used Talal's processor to implement the game helicopter(or flappy bird). The goal of the game was to avoid the oncoming walls. A score counter (in hexadecimal) incremented every time a new wall appeared on the LED array and the player was still alive. The game ends when the player LED collides with a wall LED. The score then remains constant, and the game can only be restarted by the reset button. We created our own 6 by 4 LED matrix. For the player column we used multi color LED's to differentiate between the presence of a player or wall at the LED. The player lights the LED up green, and the wall is depicted by the LED lighting up red. The other three columns are made using only red LED's. The farthest right column intakes the wall LED command and then passes it through the to the next column and so on, until it reaches the player column and moves out of the LED's.

Every cycle we implement an automatic collision check in hardware. We check between the player array and the previous column array that it passes in to detect a collision. If a collision is detected then an L (for lose) is formed in the red LED's as seen in figure 2.

We used three of the buttons on the DE2 to implement our game. The first button moves the player LED down, the second button moves the player LED up, and the third button resets the game. There are checks in our code to make sure that the LED does not go too far up or too far down. These three buttons are taken as inputs into the processor through the DE2.

As the new LED's enter the matrix, it is followed by an empty column to allow the player to move and have a chance to avoid the upcoming LED's. This implementation was added to our game so that the player has an opportunity avoid the next sequence of LED's. This was done to avoid the case where the player gets trapped within two LED's and is in an automatic losing position. This modification was made possible through assembly in which R$0 was made all 0's and added into the rightmost column.

To move the player LED we had an independent register which held the player location and used SLL and SRA to shift the position of the player. The players position is decoded into a 1 hot position and uses the shifts (sll or sra) to either move left or right based on the inputs from the buttons. These positions were always stored into the player register. This was always made to be R$1. We also had a check to make sure that if sll or sra was pressed, and the position was at the f that the position was at the rightmost or leftmost position it remained at its current position. In the Fetch stage we used 3 to 1 mux which either outputted the instruction memory or and sll by 1 bit or the sra by one bit (of $R1 which is also saved into $R1). If a PB was pressed it stopped the the program counter from incrementing in the fetch module. The edge detection was done in the execute module.

Decrementing counter loops were used in the assembly code. The memory module was modified such that on a branch condition the branch control signal was delayed by 3 cycles within the module and this signal was used as the write enable signal (along with other appropriate logic to ensure the write to register control was high and the push buttons for player were not pressed). Hence, whenever, the branch less than condition is satisfied within a particular loop, we branch to a particular position in instruction memory. The instruction at the position is written in assembly to ensure that some value is saved in the first column's led

register (with one hot encoding for the 6 leds using lower 6 bits of the 32 bit ALU result). All the other three column registers are also written to on branching condition so the column register values only update on the branching. The assembly programs below show the loops are made smaller by branching earlier to increase the speed of the falling walls. At the end of the assembly code, if a collision has not been detected we jump back up to the second instruction stored in instruction memory. As you get further down the line of code, the loops got smaller so that the commands went through quicker.

We have attached many different test files that we ran, but the final test file we ran is in Appendix A: Test 5. This test makes use of a pseudo random generator. A linear feedback shift register was constructed in the shiftRegister module which only shifted the lower six bits of a 32 bit number and kept the upper 26 bits always zero. D-FFs were used for the lower six bits. Every cycle, the lower six bits would be shited up by 1 and then written into bits 1 to 6 of the data of the register. The xnor of bit 5 anf bit 0 (where these two bits are taps, was written into the D_FF for the lowest bit of the shift register result. This ensured that the lower six bits of the shift register would be random to a certain degree. The 8th register of the regFile module was kept as a shiftRegister and its values were read after branching to input a random result in first column of LEDs.

To detect collisions we had a built in hardware detection in the pipeProcessor module. To do this we had a 1 bit wire take the output of all the bits from the playerPosition and column4. The logic for the collision detection is shown below:

assign collisionDetected1 = ((playLeds[5] & row4Leds[5]) | (playLeds[4] & row4Leds[4]) | (playLeds[3] & row4Leds[3]) | (playLeds[2] & row4Leds[2]) | (playLeds[1] & row4Leds[1]) | (playLeds[0] & row4Leds[0]));

When the game detects the collision, we set an output bit to always be high(until the reset button is pressed). This bit was the select bit for muxes in the main processor page and selected the outputs for row3, row2, row1. These outputs were properly set up to display an L on the red LEDs to show that the game had been lost..

While simulating our results in Quartas, the outputs for functional simulation showed perfect results for the given assembly code. While trying to simulate Quartas in timing, the simulation often suggested that nothing was being written even if the clock cycle was made to be very large to compensate for logic delays. However, the outputs of the game on the DE2 board showed up as expected. For example we had set a test output LED on the DE2 to show up high if there was a collision, and this worked even when our timing simulations didn't show it working.

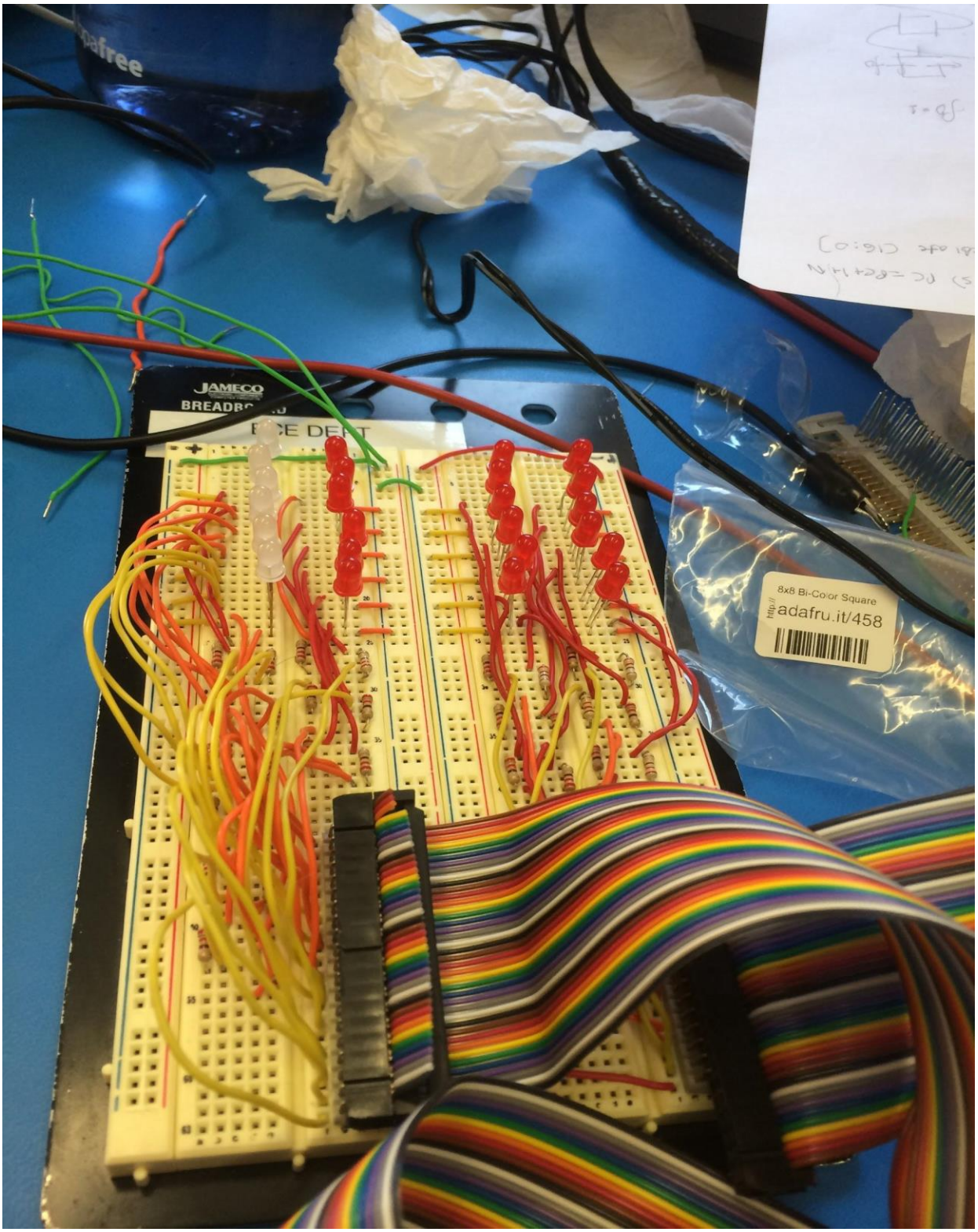

Figure 1: Our Circuit

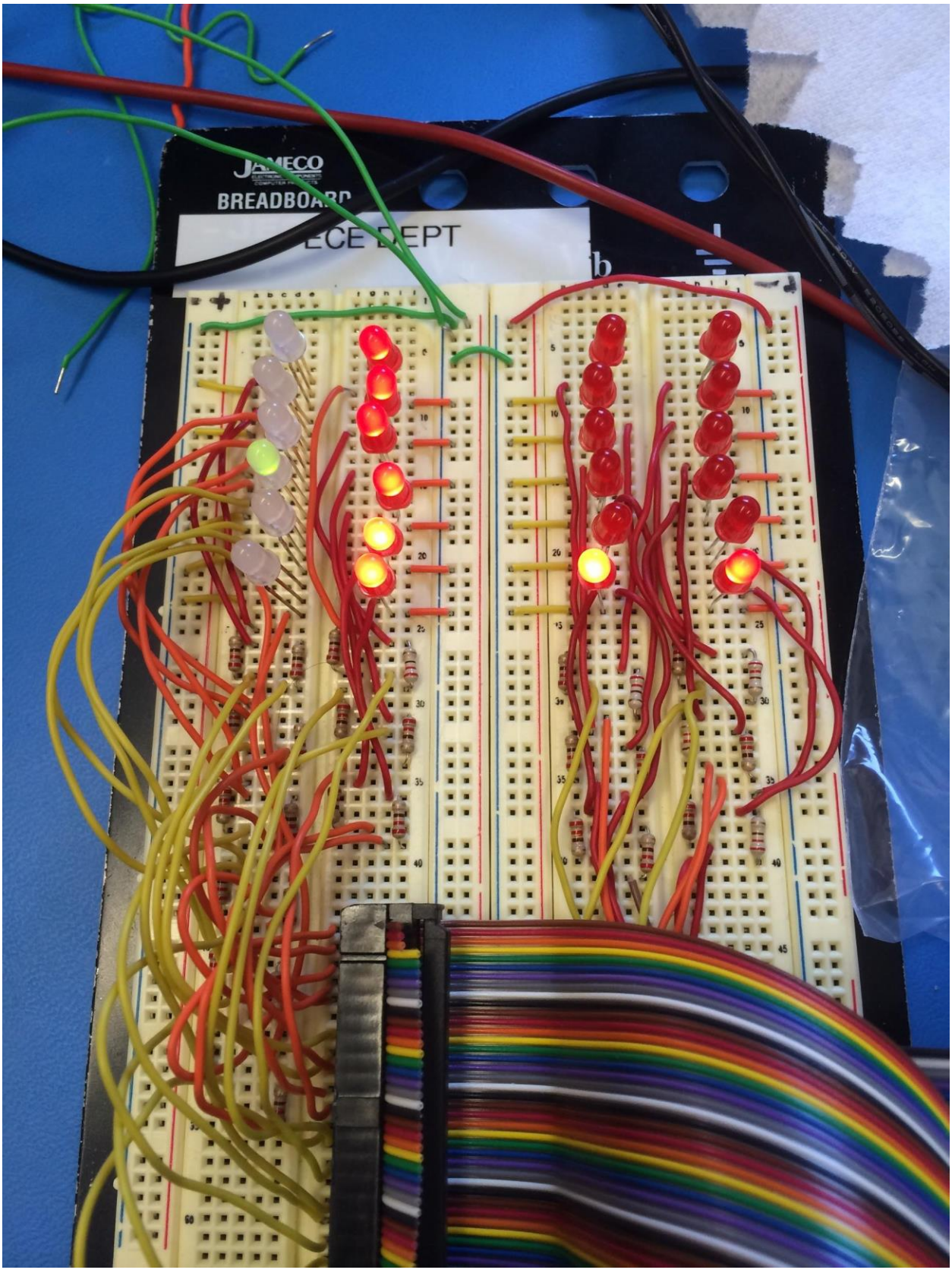

Figure 2: Circuit When You Lose

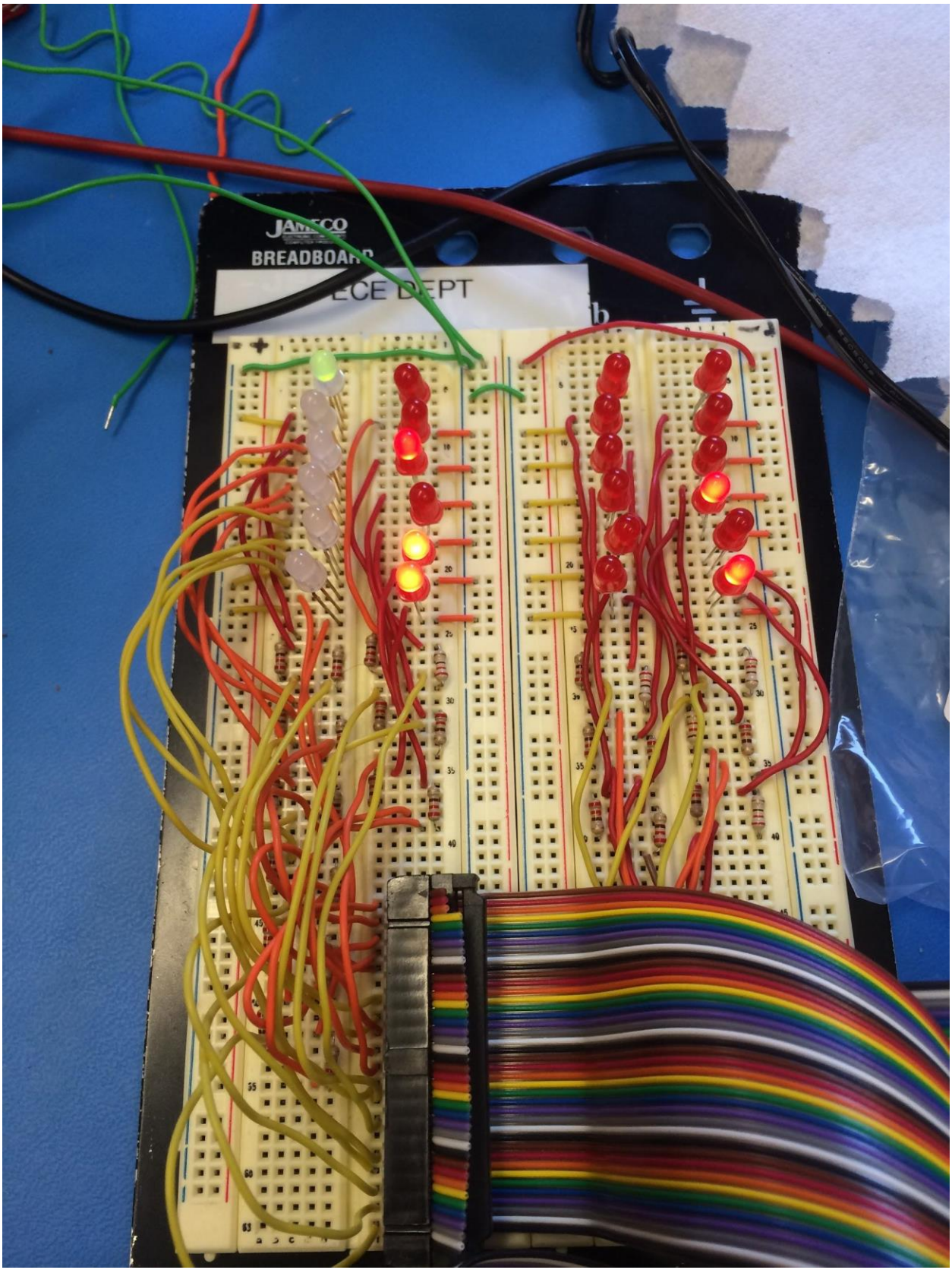

Figure 3: Blanks Between Inputs

Appendix A:
Test 1:

```
.text
main:
addi $r1, $r0, 1
addi $r2, $r0, 100
addi $r3, $r0, 90
addi $r30, $r0, 1
blt $r2, $r3, 2
sub $r2, $r2, $r30
j 4
addi $r8, $r0, 5
addi $r15, $r0, 80
blt $r2, $r15, 2
sub $r2, $r2, $r30
j 9
addi $r9, $r0, 3
addi $r4, $r0, 70
blt $r2, $r4, 2
sub $r2, $r2, $r30
j 14
addi $r10, $r0, 7
addi $r5, $r0, 60
blt $r2, $r5, 2
sub $r2, $r2, $r30
j 19
addi $r18, $r0, 21
addi $r4, $r0, 50
blt $r2, $r4, 2
sub $r2, $r2, $r30
j 24
addi $r10, $r0, 11
j 1

.data
wow: .word 0x00000001
itis: .string ASDASDASDASDASDASD
var: .char Z
label: .char A
```

Appendix A:
Test 2:

```
.text
main:
addi $r1, $r0, 1
addi $r2, $r0, 100
addi $r3, $r0, 95
addi $r30, $r0, 1
blt $r2, $r3, 2
sub $r2, $r2, $r30
j 4
addi $r8, $r0, 5
addi $r15, $r0, 90
blt $r2, $r15, 2
sub $r2, $r2, $r30
j 9
addi $r9, $r0, 3
addi $r4, $r0, 85
blt $r2, $r4, 2
sub $r2, $r2, $r30
j 14
addi $r10, $r0, 7
addi $r5, $r0, 80
blt $r2, $r5, 2
sub $r2, $r2, $r30
j 19
addi $r18, $r0, 21
addi $r4, $r0, 75
blt $r2, $r4, 2
sub $r2, $r2, $r30
j 24
addi $r10, $r0, 11
j 1

.data
wow: .word 0x00000001
itis: .string ASDASDASDASDASDASD
var: .char Z
label: .char A
```

Appendix A:
Test 3:

```
.text
main:
addi $r1, $r0, 1
addi $r2, $r0, 100
addi $r3, $r0, 95
addi $r30, $r0, 1
blt $r2, $r3, 2
sub $r2, $r2, $r30
j 4
addi $r8, $r0, 5
addi $r15, $r0, 90
blt $r2, $r15, 2
sub $r2, $r2, $r30
j 9
addi $r9, $r0, 0
addi $r4, $r0, 85
blt $r2, $r4, 2
sub $r2, $r2, $r30
j 14
addi $r10, $r0, 3
addi $r5, $r0, 80
blt $r2, $r5, 2
sub $r2, $r2, $r30
j 19
addi $r18, $r0, 0
addi $r4, $r0, 75
blt $r2, $r4, 2
sub $r2, $r2, $r30
j 24
addi $r10, $r0, 7
addi $r4, $r0, 70
blt $r2, $r4, 2
sub $r2, $r2, $r30
j 29
addi $r8, $r0, 0
addi $r15, $r0, 65
blt $r2, $r15, 2
sub $r2, $r2, $r30
j 34
addi $r9, $r0, 21
addi $r4, $r0, 60
blt $r2, $r4, 2
sub $r2, $r2, $r30
```

```
j 39
addi $r10, $r0, 0
addi $r5, $r0, 55
blt $r2, $r5, 2
sub $r2, $r2, $r30
j 44
addi $r18, $r0, 11
addi $r4, $r0, 50
blt $r2, $r4, 2
sub $r2, $r2, $r30
j 49
addi $r10, $r0, 0
j 1

.data
wow: .word 0x00000001
itis: .string ASDASDASDASDASDASD
var: .char Z
label: .char A
```

Appendix A:
Test 4:

```
.text
main:
addi $r1, $r0, 1
addi $r2, $r0, 100
addi $r3, $r0, 97
addi $r30, $r0, 1
blt $r2, $r3, 2
sub $r2, $r2, $r30
j 4
addi $r8, $r0, 5
addi $r15, $r0, 94
blt $r2, $r15, 2
sub $r2, $r2, $r30
j 9
addi $r9, $r0, 0
addi $r4, $r0, 91
blt $r2, $r4, 2
sub $r2, $r2, $r30
j 14
```

```
addi $r10, $r0, 3
addi $r5, $r0, 88
blt $r2, $r5, 2
sub $r2, $r2, $r30
j 19
addi $r18, $r0, 0
addi $r4, $r0, 85
blt $r2, $r4, 2
sub $r2, $r2, $r30
j 24
addi $r10, $r0, 7
addi $r4, $r0, 82
blt $r2, $r4, 2
sub $r2, $r2, $r30
j 29
addi $r8, $r0, 0
addi $r15, $r0, 79
blt $r2, $r15, 2
sub $r2, $r2, $r30
j 34
addi $r9, $r0, 21
addi $r4, $r0, 76
blt $r2, $r4, 2
sub $r2, $r2, $r30
j 39
addi $r10, $r0, 0
addi $r5, $r0, 73
blt $r2, $r5, 2
sub $r2, $r2, $r30
j 44
addi $r18, $r0, 11
addi $r4, $r0, 70
blt $r2, $r4, 2
sub $r2, $r2, $r30
j 49
addi $r10, $r0, 0
j 1

.data
wow: .word 0x00000001
itis: .string ASDASDASDASDASDASD
var: .char Z
label: .char A
```

Appendix A:
Test 5:

```
.text
main:
addi $r1, $r0, 1
addi $r2, $r0, 100
addi $r3, $r0, 97
addi $r30, $r0, 1
blt $r2, $r3, 2
sub $r2, $r2, $r30
j 4
addi $r8, $r8, 5
addi $r15, $r0, 92
blt $r2, $r15, 2
sub $r2, $r2, $r30
j 9
addi $r9, $r0, 0
addi $r4, $r0, 88
blt $r2, $r4, 2
sub $r2, $r2, $r30
j 14
addi $r8, $r8, 0
addi $r5, $r0, 85
blt $r2, $r5, 2
sub $r2, $r2, $r30
j 19
addi $r18, $r0, 0
addi $r4, $r0, 82
blt $r2, $r4, 2
sub $r2, $r2, $r30
j 24
addi $r8, $r8, 0
addi $r4, $r0, 80
blt $r2, $r4, 2
sub $r2, $r2, $r30
j 29
addi $r9, $r0, 0
addi $r15, $r0, 78
blt $r2, $r15, 2
sub $r2, $r2, $r30
j 34
addi $r8, $r8, 21
addi $r4, $r0, 76
```

```
blt $r2, $r4, 2
sub $r2, $r2, $r30
j 39
addi $r10, $r0, 0
addi $r5, $r0, 74
blt $r2, $r5, 2
sub $r2, $r2, $r30
j 44
addi $r8, $r8, 11
addi $r4, $r0, 72
blt $r2, $r4, 2
sub $r2, $r2, $r30
j 49
addi $r10, $r0, 0
j 1

.data
wow: .word 0x00000001
itis: .string ASDASDASDASDASDASD
var: .char Z
label: .char A
```