

Question #1

- Explain what the program does:

the program create a loop, where we always initialize before the loop body a variable with 0, in the loop body we add 6, then we subtract 1 from it and then do an and between the viable and 3, and repeat

- does the loop ever end ?


No – it does not, since the condition we use to jump to label in the program is beq zero, zero, REPEAT # Repeat the loop

and it always true.

- Calculate values in register t3 in first loop. Write down the values in the table:

code	Line	Register t3 (calculate)
main	li t3 0x0	0
repeat	addi t3 t3 6	6
	addi t3, t3, -1	5
	andi t3, t3, 3	1
beq zero, zero, REPEAT		1

Question #2

Follow the Step over  and write down the actual values of t3 (Registers window) .each step in the table

LOOP	Line	Register t3 (calculate)	Actual t3
main	li t3 0x0	0	0
repeat	addi t3 t3 6	6	6
	addi t3, t3, -1	5	5
	andi t3, t3, 3	1	1
beq zero, zero, REPEAT		1	1

Compare the calculated and actual values to test your answers.

Question #3

Change the AL_Operations.S assembly program so that it will continuously loop, reading input from the switches and writing the values to the LEDs.

Copy the code:

```
.globl main
```

```
main:
```

```
li t0, 0x80001400 # t0 = READ register
```

```
REPEAT:
```

```
    li t1, 0xFFFF
```

```
    sw t1, 8(t0)
```

```
    lw t5, 0(t0)
```

```
    srli t5, t5, 16
```

```
    mv t3, t5
```

```
    sw t3, 4(t0)
```

```
    beq zero, zero, REPEAT
```

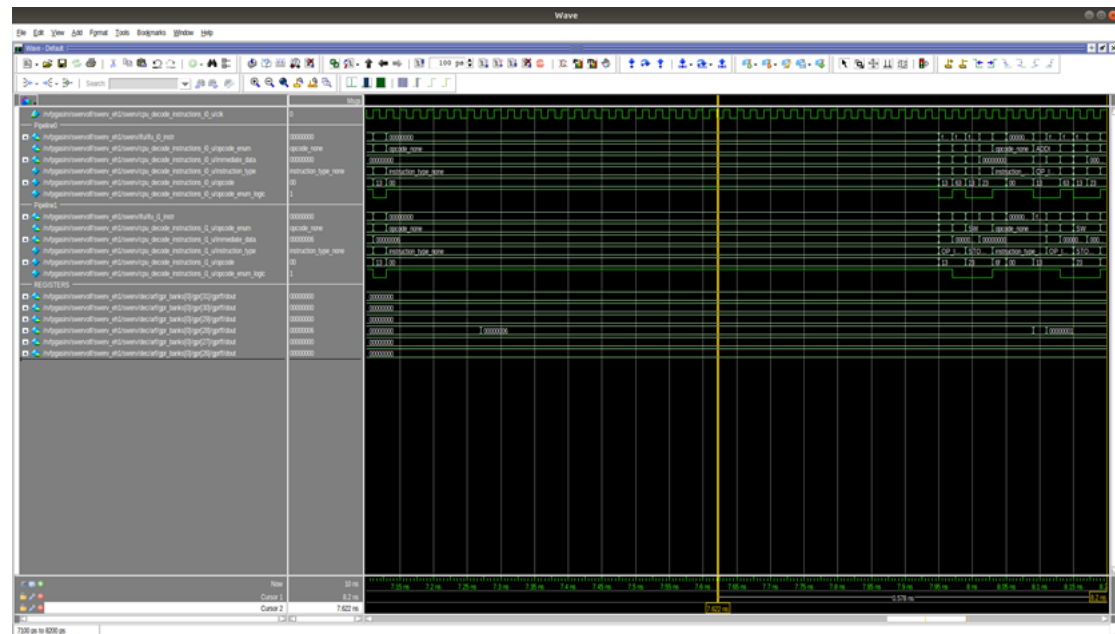
```
    nop
```

```
.end
```

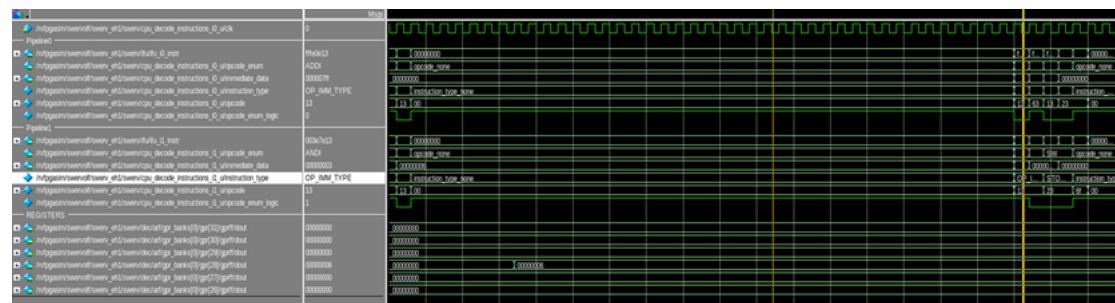
Question #4

Follow the program execution in the simulation, provide screenshots and explain all the steps.

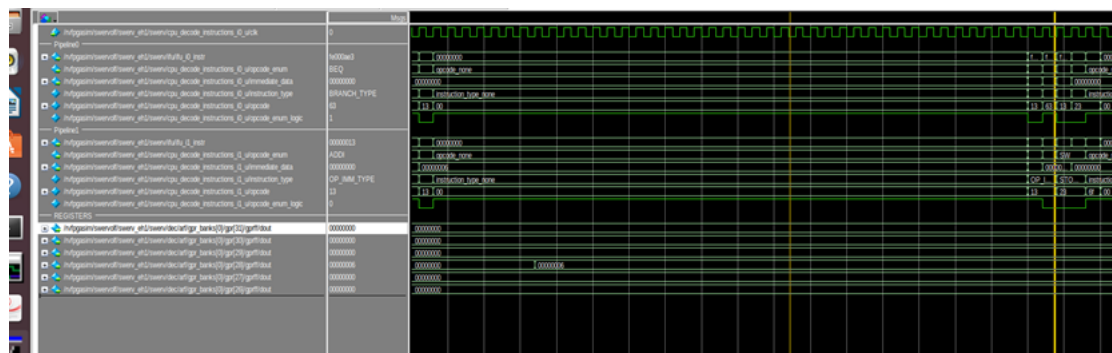
At 7.1 we fetch the first 2 instructions (li t3, 0x0 and addi t3, t3, 6), both are decoded as a addi instruction, each one in a different pipe



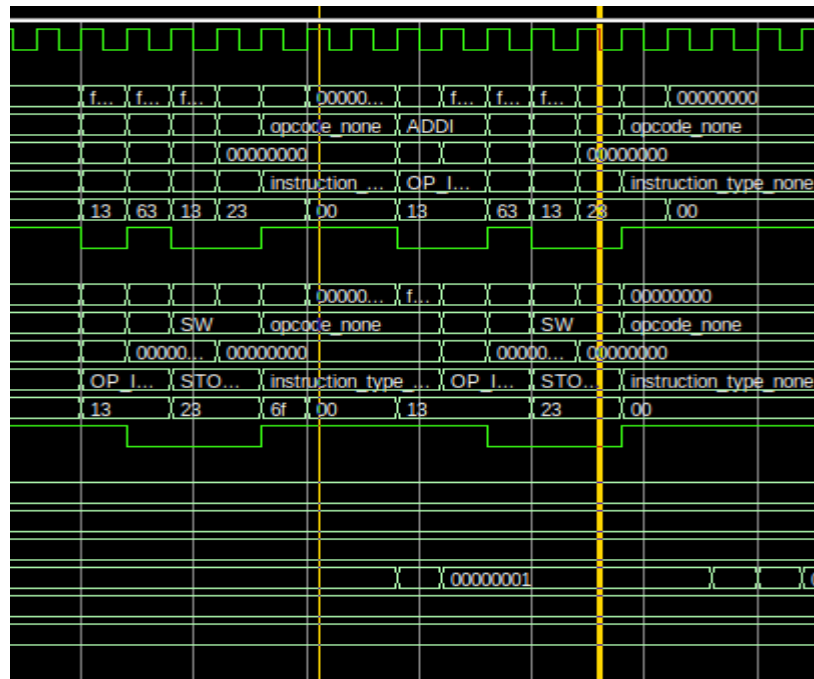
then we can see us moving to andi and addi operations



and lastly we move to beq and addi



we see that there is garbage instructions at the end for a brief time before we do the actual branch



Question #5

Look at ScrollLEDs.c program and explain what the program does.

The program is an infinite loop which has 3 loops in it.

At first, we set-on the most left led, in the first loop we move the set-on led to the left after a constant delay (set on led 1, after delay turn of and set led 2 ... till we reach the last led)

in second loop, we do the same but from right to left.

then we increment the number of set-on LEDs and repeat (second iteration of the “bigger” loop will start with led 0 & 1 on, then we turn on both and set on led 1 and 2 and so on).

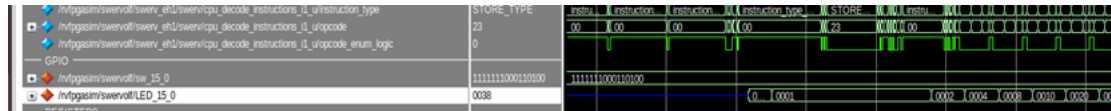
Question #6

What is displayed on LEDs? Explain

We see what the program we explained in 5 does, - some sort of dynamic LEDs.

Question # 7

- Attach a screenshot of the simulation and explain the behavior of of signal: /rvfpgasim/swervolf/LED_15_0.



We can see the referred to signal behaves as an mask to which LEDs are set on, we can see that it behaves as we expected the elds to behave in q5 and 6, starting with a mask of 0x0001, then 0x002 and so one due to the loop behavior.

Question # 8

Modify the code so that it waits for a string of 16 characters and prints it on the terminal in one line.

Copy the code:

```
#include <stdio.h>

#include <stdlib.h>

unsigned char input_data[17];

unsigned int uart_stat;

int main() {

    /* Initialize Uart */

    uartInit();

    printfNexys("type on terminal: echo Hi from Uart > /dev/ttyUSB1 \n");

    int counter = 0;

    input_data[16] = '\0';

    while (1) {

        while (counter < 16)

        {

            uart_stat = M_UART_RD_REG_LSR();

            if (uart_stat & UART_RX_AVAIL) //Wait to receive char from UART

            {

                input_data[counter] = UART_RX_DATA;

                counter++;

            }

        }

        printfNexys("%s", input_data);

        counter = 0;

    }

    return 0;

}
```

Exercise 1. How to access hardware devices

```
// memory-mapped I/O addresses

#define GPIO_SWs  0x80001400

#define GPIO_LEDs  0x80001404

#define GPIO_INOUT 0x80001408

#define DELAY 0x300000

#define READ_GPIO(dir) (*(volatile unsigned *)dir)

#define WRITE_GPIO(dir, value) { (*(volatile unsigned *)dir) = (value); }

unsigned val, i;

int main ( void )
{
    unsigned En_Value=0xFFFF;

    WRITE_GPIO(GPIO_INOUT, En_Value);
    WRITE_GPIO(GPIO_SWs, En_Value);

    val = 0;
    while (1) {
        i = 0;
        val = READ_GPIO(GPIO_SWs); //read switches
        val = val >> 16;
        WRITE_GPIO(GPIO_LEDs, val); // display val on LEDs
        while (i < DELAY) {
            i++;
        }
        i = 0;
        val = 0;
        WRITE_GPIO(GPIO_LEDs, val); // display val on LEDs
        while (i < DELAY) {
            i++;
        }
    }
    return(0);
}
```

Exercise 2. 4 bit addition

```
// memory-mapped I/O addresses

#define GPIO_SWs  0x80001400

#define GPIO_LEDs  0x80001404

#define GPIO_INOUT 0x80001408


#define DELAY 0x3000000

#define READ_GPIO(dir) (*(volatile unsigned *)dir)

#define WRITE_GPIO(dir, value) { (*(volatile unsigned *)dir) = (value); }

unsigned val_lsb, val_msb, val, res, i;

unsigned mask_lsb, mask_msb;


int main ( void )
{
    unsigned En_Value=0xFFFF;

    WRITE_GPIO(GPIO_INOUT, En_Value);

    WRITE_GPIO(GPIO_SWs, En_Value);

    val = 0;

    mask_lsb = 0xf;

    i = 0;

    while (1) {

        val = READ_GPIO(GPIO_SWs); //read switches

        val = val >> 16;

        val_lsb = (val) & mask_lsb;

        val_msb = val >> 12;

        res = val_msb + val_lsb;

        WRITE_GPIO(GPIO_LEDs, res); // display val on LEDs

        while (i < DELAY) {

            i++;

        }

        i = 0;

    }

    return(0);
}
```