Assignment 6

Question 1.

The assembly code for the factorial program was given in the lab hand out in which I added a line to store the contents of x10 in the first memory address to ensure it held the correct value, which was input into Venus and translated to machine code for use in the processor. This was then held in a register array to be accessed during operation of the processor, as seen in Figure 1. For the code of my choosing, I wanted to ensure use of all types of instructions supported by the design which included r-type, i-type, s-type, b-type and u/j-type. In Venus I adapted the assembly code from part b of the project to include jump and link as well as branch type instructions, covering all the types as stated previously.

```
initial //for factorial
   begin
      instruction[0] = 32'h00600513; //addi x10 x0 6
      instruction[1] = 32'h00c000ef; //jal x1 12
      instruction[2] = 32'h00a02023; //sw x10 0(x0)
      instruction[3] = 32'h0000007f; //halt
      instruction[4] = 32'hff810113; //addi x2 x2 -8
      instruction[5] = 32'h00112223; //sw x1 4(x2)
      instruction[6] = 32'h00a12023; //sw x10 0(x2)
      instruction[7] = 32'h00100293; //addi x5 x0 1
      instruction[8] = 32'h00551863; //bne x10 x5 16
      instruction[9] = 32'h00100513; //addi x10 x0 1
      instruction[10] = 32'h00810113; //addi x2 x2 8
      instruction[11] = 32'h00008067; //jalr x0 x1 0
      instruction[12] = 32'hfff50513; //addi x10 x10 -1
      instruction[13] = 32'hfddff0ef; //jal x1 -36
      instruction[14] = 32'h00012303; //lw x6 0(x2)
      instruction[15] = 32'h00412083; //lw x1 4(x2)
      instruction[16] = 32'h00810113; //addi x2 x2 8
      instruction[17] = 32'h02650533; //mul x10 x10 x6
      instruction[18] = 32'h00008067; //jalr x0 x1 0
      end
```

Figure 1. Factorial assembly and machine code stored in the instruction register array in the main module of the single cycle processor design, taken from Venus apart from the HALT instruction defined in the lab handout.

```
initial //for validation purposes, checks most instructions and stores them to check
   begin
      instruction[0] = 32'h004000ef; //jal x1 4
      instruction[1] = 32'h00200293; //addi x5 x0 2
      instruction[2] = 32'h00400313; //addi x6 x0 4
      instruction[3] = 32'h00031463; //bne x6 x0 8
      instruction[4] = 32'h005303b3; //add x7 x6 x5
      instruction[5] = 32'h00602223; //sw x6 4(x0)
      instruction[6] = 32'h00330393; //addi x7 x6 3
      instruction[7] = 32'h40530e33; //sub x28 x6 x5
      instruction[8] = 32'h00702423; //sw x7 8(x0)
      instruction[9] = 32'h03c384b3; //mul x9 x7 x28
      instruction[10] = 32'h00428293; //addi x5 x5 4
      instruction[11] = 32'h00402903; //lw x18 4(x0)
      instruction[12] = 32'h41248933; //sub x18 x9 x18
      instruction[13] = 32'h00291913; //slli x18 x18 2
      instruction[14] = 32'h01212023; //sw x18 0(x2)
      instruction[15] = 32'h00000033; //add x0 x0 x0
      instruction[16] = 32'h0000007f; //halt
      instruction[17] = 32'h00000000; //place holder
      instruction[18] = 32'h00000000; //place holder
      end
```

Figure 2. Verification assembly and machine code stored in the instruction register array in the main module of the single cycle processor design, taken from Venus apart from the HALT instruction defined in the lab handout.

Question 2.

As discussed in lecture, the single cycle processor requires units for an ALU, Control, Immediate Generator, Registers, Memory, and Program Count (noting that the ROM to hold instructions is omitted as they are stored in a register arrays). The memory is a RAM unit and

was tested and documented in part a of the project, in which a report was a submitted and will be omitted in the documentation of the modules,

The control unit takes in the instruction and outputs and outputs controls indicating jal, jalr, branch, bne, memtoreg, memwrite, alusrc, regwrite, and aluop. Those values then go to control how the rest of the modules act. The code for the control module can be found in the project archive. To ensure proper functioning of the control unit the testbench of figure 3 was used with the corresponding modelsim output in figure 4.

```
1  `timescale 10ns/1ns
2
3  module Control_tb;
4
5  reg [31:0] instruction;
6  wire jal, jalr, branch, bne, memtoreg, memwrite, alusrc, regwrite;
7  wire [2:0] aluop;
8
9  Control Control(instruction, jal, jalr, branch, bne, memtoreg, memwrite, alusrc, regwrite, aluop);
10
11 initial
12    begin
13    #0
14       instruction = 32'h00220133; //add
15       $monitor("%d %d %d %d %d %d %d %d %d %d %d", $realtime, instruction, jal, jalr, branch, bne, memtoreg, memwrite, alusrc, regwrite, aluop);
16    #1
17       instruction = 32'h40610433; //sub
18    #1
19       instruction = 32'h027104b3; //mul
20    #1
21       instruction = 32'h007164b3; //or
22    #1
23       instruction = 32'h002570b3; //and
24    #1
25       instruction = 32'h00100293; //addi
26    #1
27       instruction = 32'h00239113; //slli
28    #1
29       instruction = 32'h00a12103; //lw
30    #1
31       instruction = 32'h007124a3; //sw
32    #1
33       instruction = 32'h00210863; //beq
34    #1
35       instruction = 32'h00321663; //bne
36    #1
37       instruction = 32'h0080006f; //jal
38    #1
39       instruction = 32'h00008067; //jalr
40    #1
41       instruction = 32'h0000007f; //Halt
42    #1
43       $stop;
44    end
45
46 endmodule
```

Figure 3. Control module testbench code which tests for proper outputs when input with r-type, i-type, s-type, b-type, u/j-type, and halt instructions.

```
 0    2228531 0 0 0 0 0 0 1 1 0
 1 1080099891 0 0 0 0 0 0 1 1 1
 2   40961203 0 0 0 0 0 0 1 1 2
 3    7431347 0 0 0 0 0 0 1 1 3
 4    2453683 0 0 0 0 0 0 1 1 4
 5    1049235 0 0 0 0 0 0 0 1 0
 6    2330899 0 0 0 0 0 0 0 1 5
 7   10559747 0 0 0 0 1 0 0 1 0
 8    7414947 0 0 0 0 0 1 0 0 0
 9    2164835 0 0 1 0 0 0 1 0 1
10    3282531 0 0 0 1 0 0 1 0 1
11    8388719 1 0 0 0 0 0 0 1 0
12      32871 0 1 0 0 0 0 0 0 0
13        127 0 0 0 0 0 0 0 0 0
```

Figure 4. Modelsim console output of the control unit test bench, showing proper functioning for the instructions specified in Figure 3.

The immediate generator takes in the instruction and outputs an immediate value, shifting the value by 2 for i-type instruction if changing the stack pointer, as well as for l-type and s-type instruction. Branch and jal are shifted to the right by 1 instead of 2 seeing as they would later be shifted left by 1 (which is done preliminarily). Jalr instructions are not shifted seeing as for all intents and purposes of the instruction in our use the immediate value will be 0. Code for the immediate generator can be found in the project archive. To ensure proper

functioning of the ALU unit the testbench of figure 5 was used with the corresponding modelsim output in figure 6.

```
1   `timescale 10ns/1ns
2
3   module Immediate_Generator_tb;
4
5   reg [31:0] instruction;
6   wire signed [31:0] immediate;
7
8   Immediate_Generator IG(instruction, immediate);
9
10  initial
11     begin
12     #0
13        instruction = 32'hff830193; //addi -8 (not x2)
14        $monitor("%d %d %d", $time, instruction, immediate);
15     #1
16        instruction = 32'h01c21393; //slli 28
17     #1
18        instruction = 32'hff810113; //addi x2 x2 -8
19     #1
20        instruction = 32'hff402203; //lw -12
21     #1
22        instruction = 32'h01402383; //lw 20
23     #1
24        instruction = 32'hfe802223; //sw -28
25     #1
26        instruction = 32'h04902823; //sw 80
27     #1
28        instruction = 32'hfe0000e3; //beq -32
29     #1
30        instruction = 32'h00249a63; //bne 20
31     #1
32        instruction = 32'hfd9ff0ef; //jal -40
33     #1
34        instruction = 32'h00c000ef; //jal 12
35     #1
36        instruction = 32'hff408067; //jalr -12
37     #1
38        instruction = 32'h01208067; //jalr 18
39     #1
40        $stop;
41     end
42
43  endmodule
```

Figure 5. Immediate generator testbench code, testing i-type, l-type, s-type, b-type, and u/j-type for both positive and negative values (as well as i-type with respect to the stack pointer)

```
 0 4286775699          -8
 1   29496211          28
 2 4286644499          -2
 3 4282393091          -3
 4   20980611           5
 5 4269810211          -7
 6   76556323          20
 7 4261413091          -8
 8    2398819           5
 9 4255117551         -10
10   12583151           3
11 4282417255         -12
12   18907239          18
```

Figure 6. Modelsim console output for the immediate generator testbench, showing proper functioning with respect to the input instructions for both positive and negative values.

The ALU takes in registers 1 and 2 stored in the instruction, the alusrc (to determine whether to use the register data or immediate value) and aluop (to determine the operation) from the control unit, as well as immediate value generated in the immediate generator unit. It outputs the result of the operation (alures) and a zero value that is used for branch type instructions (when aluop is subtract) that is one when both values are equivalent. Code for the ALU can be

found in the project archive. To ensure proper functioning of the ALU unit the testbench of figure 7 was used with the corresponding modelsim output in figure 8.

```
1  `timescale 10ns/1ns
2
3  module ALU_tb;
4
5  reg alusrc;
6  reg [2:0] aluop;
7  reg signed [31:0] rd1, rd2, immediate;
8  wire zero;
9  wire signed [31:0] alures;
10
11 ALU ALU(alusrc, aluop, rd1, rd2, immediate, zero, alures);
12
13 initial
14    begin
15    #0
16       alusrc = 0; aluop = 0; rd1 = 80; rd2 = -50; immediate = -10; //add immediate
17       $monitor("%d %d %d %d %d %d %d %d", $time, alusrc, aluop, rd1, rd2, immediate, zero, alures);
18    #1
19       alusrc = 1; aluop = 0; rd1 = 80; rd2 = -50; immediate = -10; //add register 2
20    #1
21       alusrc = 0; aluop = 1; rd1 = 5; rd2 = 80; immediate = 5; //subtract immediate
22    #1
23       alusrc = 1; aluop = 1; rd1 = 5; rd2 = 80; immediate = 5; //subtract register 2
24    #1
25       alusrc = 0; aluop = 2; rd1 = 40; rd2 = -2; immediate = 3; //multiply immediate
26    #1
27       alusrc = 1; aluop = 2; rd1 = 40; rd2 = -2; immediate = 3; //multiply register 2
28    #1
29       alusrc = 0; aluop = 3; rd1 = -20; rd2 = 30; immediate = -20; //or immediate
30    #1
31       alusrc = 1; aluop = 3; rd1 = -20; rd2 = 30; immediate = -20; //or register 2
32    #1
33       alusrc = 0; aluop = 4; rd1 = 10; rd2 = -5; immediate = 10; //and immediate
34    #1
35       alusrc = 1; aluop = 4; rd1 = 10; rd2 = -5; immediate = 10; //and register 2
36    #1
37       alusrc = 0; aluop = 5; rd1 = 40; rd2 = -5; immediate = 8; //shift left immediate
38    #1
39       alusrc = 1; aluop = 5; rd1 = 40; rd2 = -5; immediate = 8; //shift left register 2
40    end
41
42 endmodule
```

Figure 7. ALU testbench code which inputs variations of add, subtract, multiply, or, and, and shift for both immediate values and register data.

```
 0 0 0          80          -50          -10 0           70
 1 1 0          80          -50          -10 0           30
 2 0 1           5           80            5 1            0
 3 1 1           5           80            5 0          -75
 4 0 2          40           -2            3 0          120
 5 1 2          40           -2            3 0          -80
 6 0 3         -20           30          -20 0          -20
 7 1 3         -20           30          -20 0           -2
 8 0 4          10           -5           10 0           10
 9 1 4          10           -5           10 0           10
10 0 5          40           -5            8 0        10240
11 1 5          40           -5            8 0            0
```

Figure 8. Modelsim console output for the ALU testbench of Figure 7 showing proper functioning with regard to the input register 1 data inputs and the alusrc register.

The program count module updates the program count on the clock edge based on the instruction, particularly based on the 7-bit opcode of the instruction. For a halt instruction the pc should remain the same, it should increase by the immediate value if the instruction is a beq (and the zero condition is satisfied), bne (and the zero condition is not satisfied), or a jal. The immediate value would typically be shifted to the left one value, but we accounted for the shift in the generation of the immediate value so it can be omitted. For the purposed of the jalr instruction as we need it the pc with simply be the value stored in the return address register plus one to move onto the next instruction. If none of the conditions are met, then the program count

should increase by one to move onto the next instruction. Code for the Program Count module can be found in the project archive. To ensure proper functioning of the Program Count unit the testbench of figure 9 was used with the corresponding modelsim output in figure 10.

```
1  `timescale 10ns/1ns
2
3  module Program_Count_tb;
4
5  reg clk, branch, zero, bne, jal, jalr;
6  reg signed [31:0] immediate, alures;
7  reg [6:0] instruction;
8  wire [31:0] pc;
9
10 Program_Count PC(clk, branch, zero, bne, jal, jalr, instruction, alures, immediate, pc);
11
12 initial
13    begin
14    #0
15       clk = 1; branch = 0; zero = 0; bne = 0; jal = 0; jalr = 0; instruction = 0; alures = 0; immediate = 4; //begin at posedge (counts up 1 immediately)
16       $monitor("%d %d %d %d %d %d", $time, branch, zero, instruction, immediate, pc);
17    #1
18       branch = 1; zero = 0; bne = 0; jal = 0; jalr = 0; instruction = 0; alures = 12; immediate = 2; //count up
19    #1
20       branch = 0; zero = 1; bne = 0; jal = 1; jalr = 0; instruction = 8; alures = 10; immediate = 8; //jal 8
21    #1
22       branch = 1; zero = 1; bne = 0; jal = 0; jalr = 0; instruction = 0; alures = 0; immediate = 5; //branch 5
23    #1
24       branch = 0; zero = 0; bne = 0; jal = 0; jalr = 1; instruction = 5; alures = 8; immediate = 3; //jalr, set to 8 + 1
25    #1
26       branch = 0; zero = 0; bne = 1; jal = 0; jalr = 0; instruction = 0; alures = 0; immediate = 4; //bne 4
27    #1
28       branch = 0; zero = 1; bne = 1; jal = 0; jalr = 0; instruction = 0; alures = 1; immediate = 0; //bne not satisfied, inc 1
29    #1
30       branch = 0; zero = 0; bne = 0; jal = 1; jalr = 0; instruction = 4; alures = 0; immediate = 2; //jal 2
31    #1
32       branch = 0; zero = 0; bne = 0; jal = 0; jalr = 1; instruction = 2; alures = 4; immediate = 2; //jalr, set to 4 + 1
33    #1
34       branch = 0; zero = 0; bne = 0; jal = 0; jalr = 0; instruction = 9; alures = 0; immediate = 2; //inc 1
35    #1
36       branch = 0; zero = 0; bne = 0; jal = 0; jalr = 0; instruction = 127; alures = 0; immediate = 2; //hault, stay the same
37    #1
38       branch = 0; zero = 0; bne = 0; jal = 0; jalr = 0; instruction = 0; alures = 0; immediate = 0; //inc 1
39    #1
40       $stop;
41    end
42
43 always @*
44    #0.5 clk <= ~clk;
45
46 endmodule
```

Figure 9. Program count testbench code, showing inputs wher the jal condition is met, jalr condition is met, branch and zero is met, branch is met but zero is not, bne and zero conditions are met, and bne is met but zero is not. Also includes a halt instruction and cases in which no conditions are met and the pc should increase by one.

```
 0 0 0   0          4          1
 1 1 0   0          2          2
 2 0 1   8          8         10
 3 1 1   0          5         15
 4 0 0   5          3          9
 5 0 0   0          4         13
 6 0 1   0          0         14
 7 0 0   4          2         16
 8 0 0   2          2          5
 9 0 0   9          2          6
10 0 0 127          2          6
11 0 0   0          0          7
```

Figure 10. Modelsim console output for the Program Count testbench of Figure 9, showing proper functioning for the input conditions and immediate values.

The registers module updates the register array on the clock edge, taking in the regwrite, memtoreg, and jal from the Control unit, destination register and registers 1 & 2 from the instruction, the program count, ALU result, Memory data, and outputs the data read from registers 1 and 2. All 32 registers are initialized to 0 except for the stack pointer in register 2. Register one is updated to hold the current program count when a jal instruction is executed. If regwrite is enabled the memtoreg control determines if the destination register is set to the memory result (if the bit is 1) or the ALU result (if the bit is 0). Code for the Registers module can be found in the project archive. To ensure proper functioning of the Registers unit the testbench of figure 11 was used with the corresponding modelsim output in figure 12.

```
`timescale 10ns/1ns

module Registers_tb;

reg clk, regwrite, memtoreg, jal;
reg [4:0] reg1, reg2, destreg;
reg signed [31:0] pc, alures, memres;
wire signed [31:0] rd1, rd2;

Registers R1(clk, regwrite, memtoreg, jal reg1, reg2, destreg, pc, alures, memres, rd1, rd2);

    begin
    #0
        clk = 1; regwrite = 1; memtoreg = 0; jal = 0; reg1 = 0; reg2 = 31; destreg = 1; pc = 0; alures = 80; memres = -95; //set reg 1 to 80, check 0 and 31 to ensure 0 (starts at posedge, reg 1 changes immediately)
        $monitor("%d %d %d %d %d %d %d %d %d %d %d %d %d", $time, regwrite, memtoreg, jal, reg1, reg2, destreg, pc, alures, memres, rd1, rd2);
    #1
        regwrite = 1; memtoreg = 1; jal = 0; reg1 = 0; reg2 = 31; destreg = 30; pc = 0; alures = -230; memres = 10; //set reg 30 to 10, check 0 and 31 to ensure 0
    #1
        regwrite = 1; memtoreg = 0; jal = 0; reg1 = 1; reg2 = 30; destreg = 1; pc = 0; alures = -20; memres = 15; //set reg 1 to -20, check 1 and 30
    #1
        regwrite = 1; memtoreg = 1; jal = 0; reg1 = 1; reg2 = 30; destreg = 29; pc = 0; alures = 17; memres = 542; //set reg 29 to 542, check 1 and 30
    #1
        regwrite = 1; memtoreg = 0; jal = 0; reg1 = 2; reg2 = 29; destreg = 2; pc = 0; alures = 276; memres = 53; //set reg 2 to 276, check 2 and 29 to ensure 2 (sp) starts at 31
    #1
        regwrite = 1; memtoreg = 1; jal = 0; reg1 = 2; reg2 = 29; destreg = 28; pc = 0; alures = -13; memres = -26; //set reg 28 to -26, check 2 and 29
    #1
        regwrite = 1; memtoreg = 0; jal = 0; reg1 = 3; reg2 = 28; destreg = 3; pc = 0; alures = 0; memres = 10; //set reg 3 to 0, check 3 and 28
    #1
        regwrite = 1; memtoreg = 1; jal = 0; reg1 = 3; reg2 = 28; destreg = 27; pc = 0; alures = 810; memres = -575; //set reg 27 to -575, check 3 and 28
    #1
        regwrite = 1; memtoreg = 0; jal = 0; reg1 = 4; reg2 = 27; destreg = 4; pc = 0; alures = -12; memres = 94; //set reg 4 to -12, check 4 and 27
    #1
        regwrite = 1; memtoreg = 1; jal = 0; reg1 = 4; reg2 = 27; destreg = 26; pc = 0; alures = 75; memres = 0; //set reg 26 to 0, check 4 and 27
    #1
        regwrite = 0; memtoreg = 0; jal = 0; reg1 = 5; reg2 = 26; destreg = 5; pc = 0; alures = 4; memres = 34; //regwrite not enabled and memtoreg 0, check 5 and 26
    #1
        regwrite = 0; memtoreg = 1; jal = 0; reg1 = 5; reg2 = 26; destreg = 25; pc = 0; alures = -59; memres = 10; //regwrite not enabled and memtoreg 1, check 5 and 26
    #1
        regwrite = 1; memtoreg = 0; jal = 0; reg1 = 6; reg2 = 25; destreg = 6; pc = 0; alures = -48; memres = 4; //set reg 6 to -48, check 6 and 25
    #1
        regwrite = 1; memtoreg = 1; jal = 0; reg1 = 6; reg2 = 25; destreg = 24; pc = 0; alures = 0; memres = -114; //set reg 24 to -114, check 6 and 25
    #1
        regwrite = 1; memtoreg = 1; jal = 1; reg1 = 7; reg2 = 24; destreg = 1; pc = 12; alures = 0; memres = 0; //set reg1 to pc = 12
    #1
        regwrite = 0; memtoreg = 0; jal = 0; reg1 = 7; reg2 = 24; destreg = 1; pc = 0; alures = 0; memres = 0; //check that reg1 was set to pc
    #1
        $stop;
    end

always @*
    #0.5 clk <= ~clk;

endmodule
```

Figure 11. Registers module testbench code, showing an increment of register 1 from 0 to 7 and decrement of register 2 from 31 to 24. The first two instructions ensure registers begin at 0, at which point the registers begin to change. Inputs also include a jal instruction, as well as cases in which regwrite is disabled.

```
 0  1  0  0   0 31  1          0          80         -95           0           0
 1  1  1  0   0 31 30          0        -230          10           0           0
 2  1  0  0   1 30  1          0         -20          15         -20          10
 3  1  1  0   1 30 29          0          17         542         -20          10
 4  1  0  0   2 29  2          0         276          53         276         542
 5  1  1  0   2 29 28          0         -13         -26         276         542
 6  1  0  0   3 28  3          0           0          10           0         -26
 7  1  1  0   3 28 27          0         810        -575           0         -26
 8  1  0  0   4 27  4          0         -12          94         -12        -575
 9  1  1  0   4 27 26          0          75           0         -12        -575
10  0  0  0   5 26  5          0           4          34           0           0
11  0  1  0   5 26 25          0         -59          10           0           0
12  1  0  0   6 25  6          0         -48           4         -48           0
13  1  1  0   6 25 24          0           0        -114         -48           0
14  1  1  1   7 24  1         12           0           0           0        -114
15  0  0  0   1 24  1          0           0           0          12        -114
```

Figure 12. Modelsim output of the Registers testbench of Figure 11 showing proper register 1 and 2 outputs as specified by the inputs. Registers end at 0, jal sets register one to the current pc, and the regwrite and memtoreg controls work as expected.

Question 3.

The factorial and verification codes were run through Venus to get an idea of what the memory contents should look like if the processor functioned properly. For the factorial code, the end of the memory contents should hold the return address of where the factorial program was called, followed by the number we wish to take the factorial of, and then the return address of the second jal in the else statement, the factorial decreased by one, continuing until the stored value hits one. At which point the code goes through and pulls the factorial and its decrements to multiply iteratively, storing the final value in the first memory address. Proper functioning is shown by the memory contents extracted after running the program as seen in Figure 13. The validation code I created shows that within address 1 and 2 of the memory should store the values put into registers 6 (value 4) and 7 (value 4+3), respectively. The contents of register 18 should be put into the last memory address (value (7*2) – 4 << 2 = 40). Proper Functioning (at a low frequency of 50 MHz) is shown by the memory contents extracted after running the program as seen in Figure 14.

Instance 0: MEM

000000	00	00	02	D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000005	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000a	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000f	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	01
000014	00	00	00	0D	00	00	00	02	00	00	00	0D	00	00	00	03	00	00	00	0D
000019	00	00	00	04	00	00	00	0D	00	00	00	05	00	00	00	0D	00	00	00	06
00001e	00	00	00	01	00	00	00	00												

Figure 13. Memory contents after running the factorial code from Question 1 for the factorial program, showing proper functioning as expected by Venus.

Instance 0: MEM

000000	00	00	00	00	00	00	00	04	00	00	00	07	00	00	00	00	00	00	00	00
000005	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000a	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000f	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000014	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000019	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00001e	00	00	00	00	00	00	00	28												

Figure 14. Memory contents after running the code for the verification program, showing proper functioning as expected by Venus. Noting that the code was run at a lower frequency than the factorial code of 50 MHz to ensure proper function.

Question 4.

Using the board and incrementing the PLL clock, my processor has a maximum clock frequency of 70 MHz! This was mainly optimized by instead of shifting the generated immediate values I simply omitted the last two (or one) bits right away, allowing the result to get to the ALU faster which can then take the rest of the clock cycle to compute the result. The clock settings that yielded the maximum frequency can be found in Figure 15.

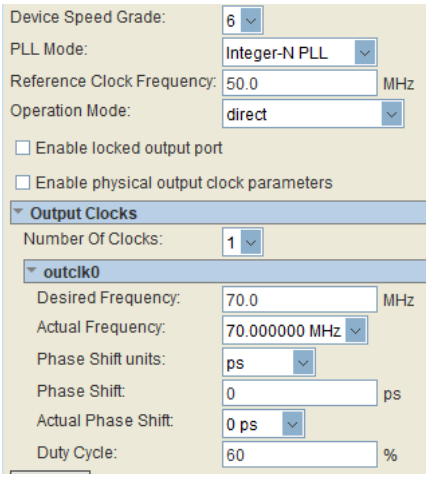


Figure 15. Maximum clock frequency settings for the factorial 6 program of questions 1 and 3, showing a maximum frequency of 70 MHz with a duty cycle of 60 percent.

Question 5.

The findings reported in question four were for factorial 6, running the program again for factorial 12 shows a different maximum clock frequency. Taking 12 factorial I was able to run the processor at a max frequency of only 65 MHz, lower than that of 6 factorial. This is because

the ALU takes longer to compute the multiplication for larger numbers due to the labor-intensive nature of bit multiplication. Multiplication introduces a lot of growth in the number (for example 2 8-bit numbers result in a 16 bit one) which ultimately requires more of the resources and thus time to execute! The maximum clock settings for the 12 factorial code are shown in Figure 16. Validation of the code shows similar results to that of Figure 13, with a stored value in address 0 of 1C8CFC00, which is expected.

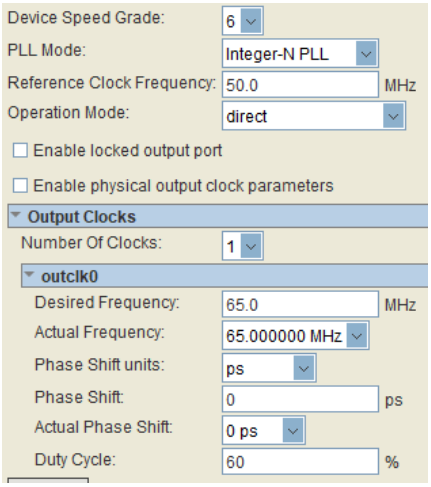
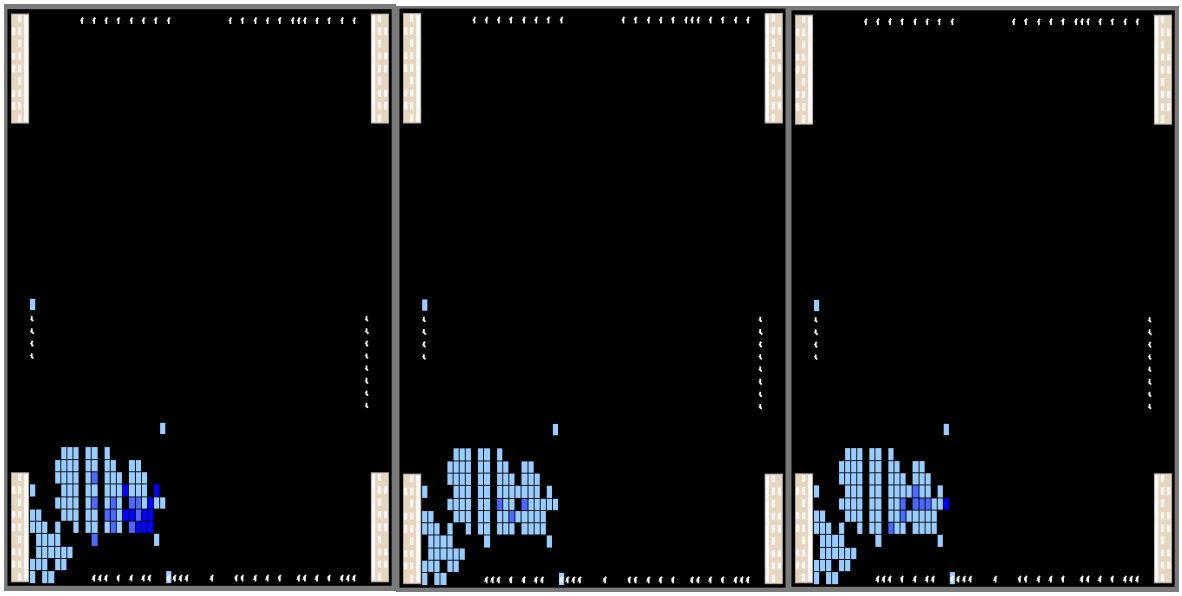


Figure 16. Maximum clock frequency for the factorial 12 program, showing a maximum frequency of 65 MHz with a duty cycle of 05 percent, slightly less than that of the factorial 6 program.

Question 6.

The floor plan for the ALU, Control, Immediate Generator, Memory, Program Count, and Register modules are shown in figure 17. The logic utilization summary is also shown in Figure 28, showing the use of 629 ALMs, 593 DLRs, 2 DSPs, and 1 PLL.



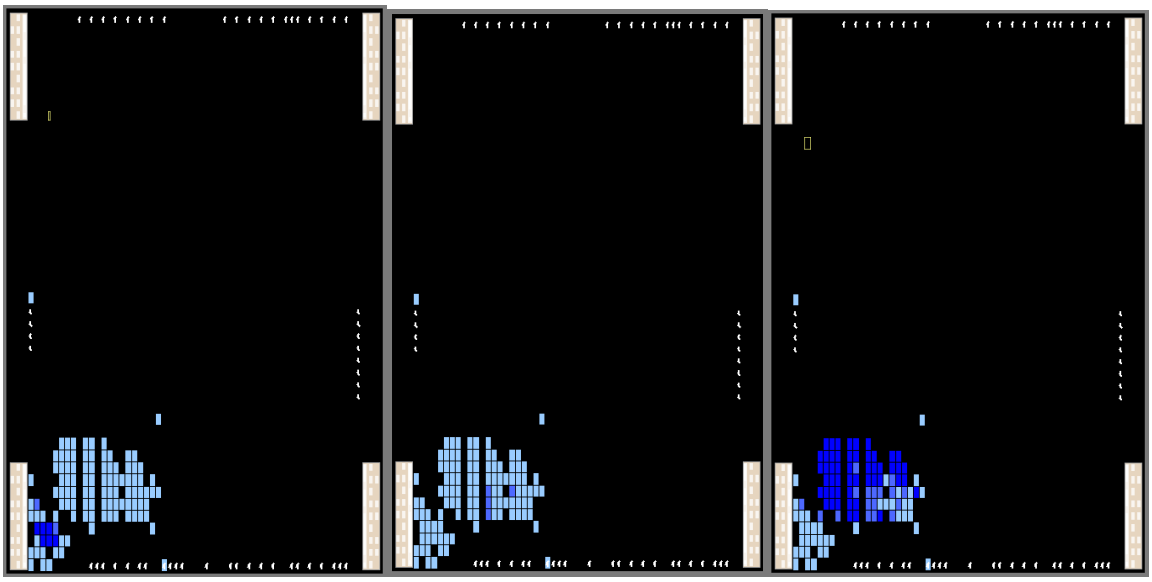


Figure 17. Floor plan for the modules that make up the processor (the elements making them up being dark blue). On the top showing ALU, Control, and Immediate generator (from left to right) and on the bottom showing the Memory, Program Count, and Registers (from left to right)

	Resource	Usage
1	Estimate of Logic utilization (ALMs needed)	629
2		
3	Combinational ALUT usage for logic	906
1	-- 7 input functions	38
2	-- 6 input functions	219
3	-- 5 input functions	311
4	-- 4 input functions	189
5	-- <=3 input functions	149
4		
5	Dedicated logic registers	593
6		
7	I/O pins	1
8	Total MLAB memory bits	0
9	Total block memory bits	1024
10		
11	Total DSP Blocks	2
12		
13	Total PLLs	1
1	-- PLLs	1
14		
15	Maximum fan-out node	PLL_Clock:PLL|PLL_Clock_0002:pll_clock_inst|altera_pll:altera_pll_i|outclk_wire[0]
16	Maximum fan-out	481
17	Total fan-out	6862
18	Average fan-out	4.42

Figure 18. Logic resource utilization, showing a usage of 629 ALMs, 593 DLRs, 2 DSPs, and 1 PLL.