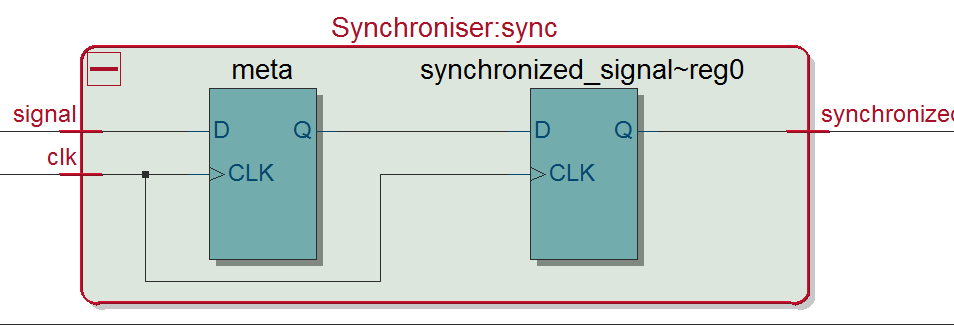
**Digital System Design Project 1 Report**

Part 1 Computer

**Stage 2**

**Synchronizer**

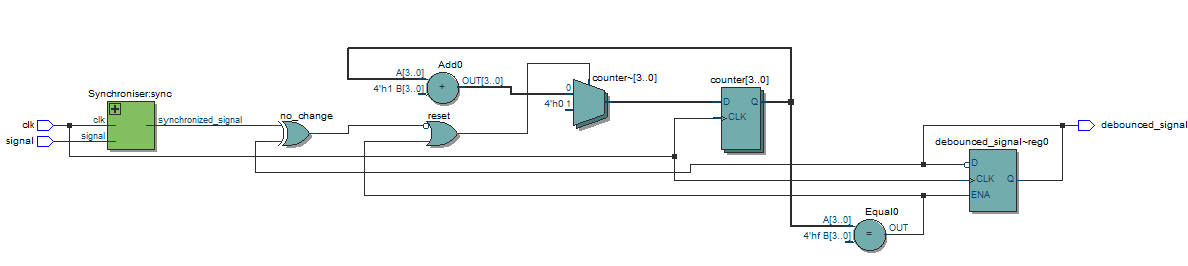


RTL Diagram for the Synchronizer module

**Explanation:**

The implementation of the synchronizer is fairly simple. It consists of two flip flops, which reduce the probability of the signal staying in the metastability state.

**Debounce**



RTL diagram for Debounce module

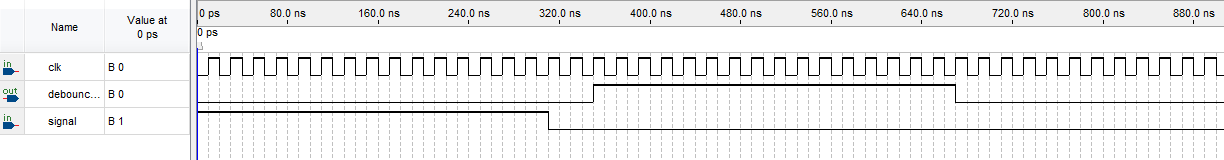
**Explanation:**

There is a counter in the module will keep counting until it counts to 25ms. We have a 50 MHz clock from the DE Board. 50 MHz means there are 5\*10^7 clock cycles in one second. Because 25 ms is 1/40 of 1 second, we need to counts 1.25\*10^6 clock cycles to reach 25ms. In order to have a counter which will be able to count up to 1.25\*10^6, we need the length of the counter to be 21 (log2 (1.25\*10^6)).

There are some critical variables in order to realize the function of the Debounce module.

First there is a “Reset” pin. When Reset is on, the counter in module will be reset to 0. The Reset will be HIGH under two conditions. One condition is when the counter has reached its maximum allowed value, and the other is when the current value is different from the previous one. We use a XOR gate to check the equality between the current signal and the previous one.

If the counter has reached its maximum value (i.e. 25 ms has passed), the signal is changed to its opposite state (like a switch), and else it remains the same.



Functional simulation of Debounce module by university waveform program

In order to verify the function of Debounce module, a waveform program was created. Because it takes a relatively long period of time to simulate over a long period of time like 25ms, to make the simulation more efficient the length of counter was shorten to 4 bits long and the maximum of the counter number was set to 4’b1111 (which is 15 in decimal). These changes would not influence the function of the Debounce module. What we expected from the simulation is the output would be zero if the input value lasted its value less than 15 clock cycles. The simulation matched with the expectation. The delay of the output was caused by the synchronizer module which consists of two flip flops (therefore, it is delayed by two clock cycles).

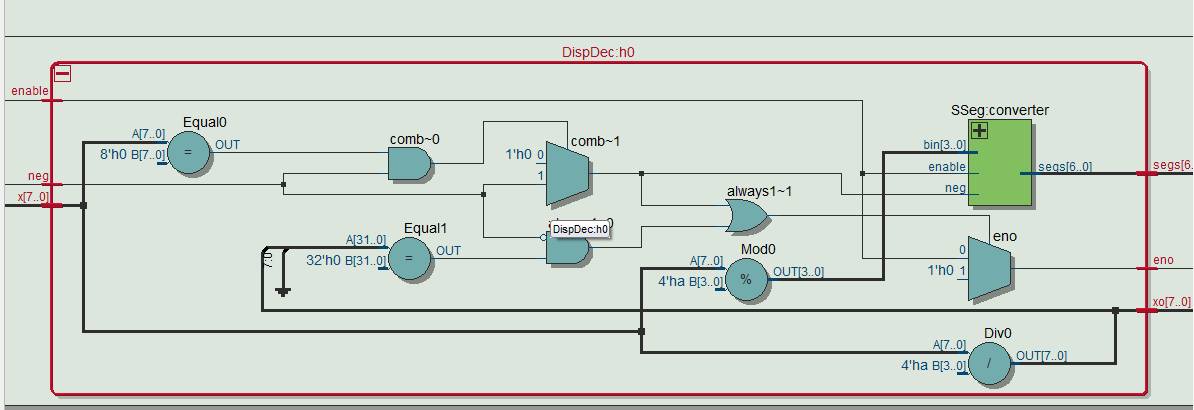
**Mechanism**

Synchronisers are used to process asynchronous signals inputs for instance, push buttons and switches on the board. Passing these signals through a synchronise can reduce the chance of violating the setup time, hold time constrain of a flip flop which will cause metastability (ie avoid signal change too close to the posedge of the clock). It is important that every signal coming from outside of the circuit has been synchronized with the CLOCK\_50 in the circuit.

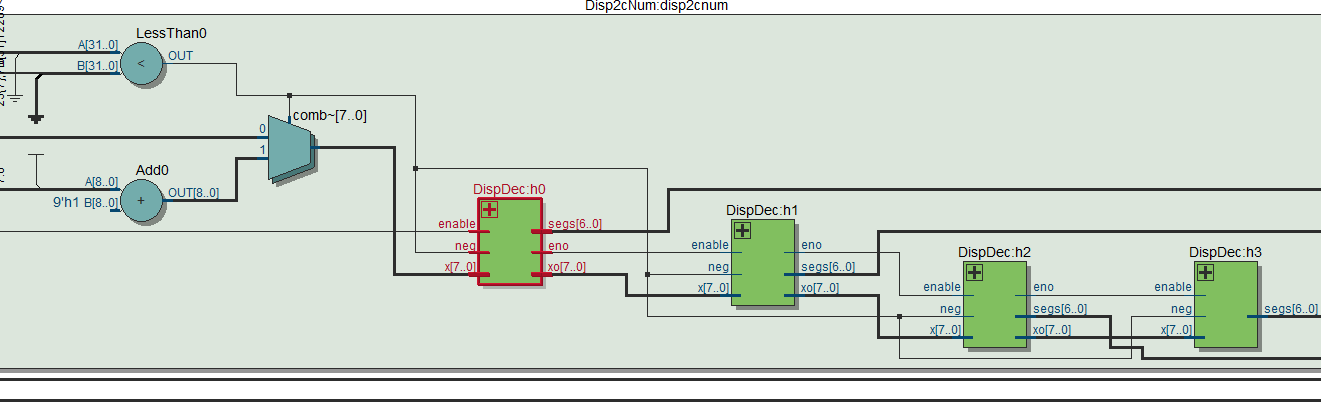
Debouce module was used to make sure the signal is stable (for at 30ms) before they are processed by the combinational logic circuit. The switches on the board might produce an unstable signal when its switched therefore a debouncer is introduced to gain stability.

**Stage 4**

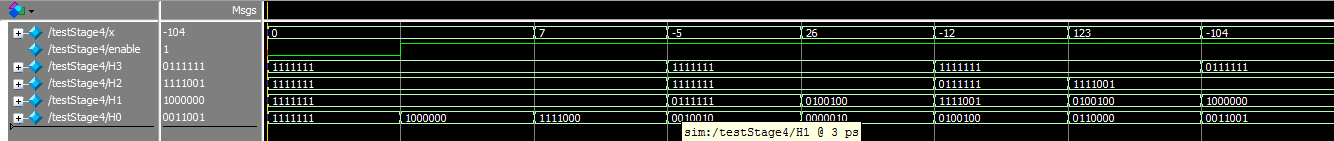
DispDec Module:



RTL Diagram for DispDec Module



RTL Diagram for Disp2cNum Module



Simulation result for Disp2cNum in ModelSim

Explanation:

Disp2cNum is used to display 3 bit decimal number on 4 7-seg displays(the MSB could only be a negative sign). The mechanism of this module is to cascade 4 DispDec instances together to decide what to show on each of the display. Each of the DispDec module checks whther the input is still greater than 1 or not, if it is, get the number by mod 10, if less than 1, pass enable=0 to the next module to shut off that bit of 7-seg display.

DispHex is just Displaying 2 bit of hexadecimal number of 2 7-seg display.

Step 1 Stage 5

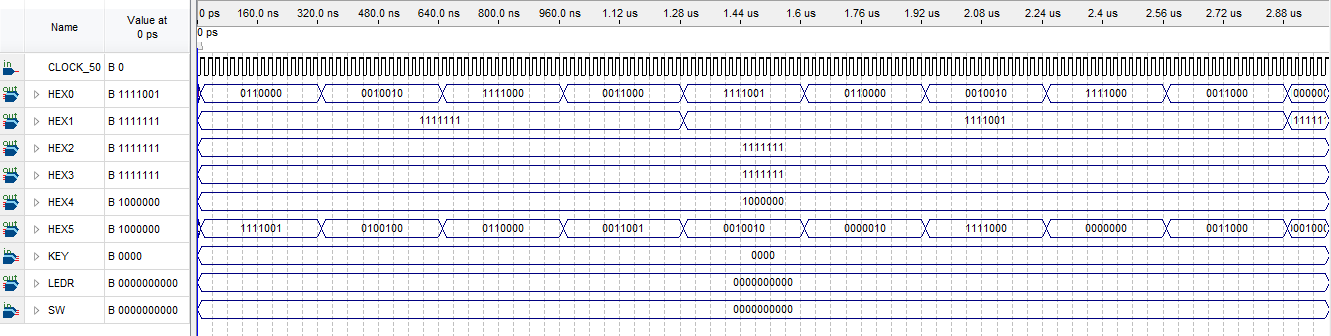
* Why will Quartus complain if the Instruction Cycle is split over two “ always” blocks, with the first having “if go” and the second having if “if Reset”.

If we spilt the Instruction Cycle over two “always” blocks, Quartus gave us the following warning message:

Error (10028): Can't resolve multiple constant drivers for net "IP[7]" at CPUTest.v(200)

It brings multiple constant drivers for IP (i.e. there are multiple flip-flops driving IP at the same time which creates conflicts (so called a race condition, which is undesirable) with each other).

Step 2 Stage 5



Functional simulation of Step 2 Stage 5 by University Waveform Program

* How does Pmem (IP, Instruction) work?

This module receives an address which stores an instruction under that in the module. And this module outputs the instruction so that the CPU can use the instruction to do operations described in it.

* What does the test code do?

The test code sets Dval to HIGH.

And it connects Dout pin of CPU to [17:25] bits of the instruction which is sent from ROM.

17: 25 bits of the instruction is a number value which is also moved to the register which stores the output value.

* Why, after showing 19, does the display remain on 0 until the IP reaches 00 again?

Because there are only 10 non-default cases which have specific instruction under them.

After address 9, it will go to the default address which stores the default instruction NOP, which instructs CPU to stays in the previous state.

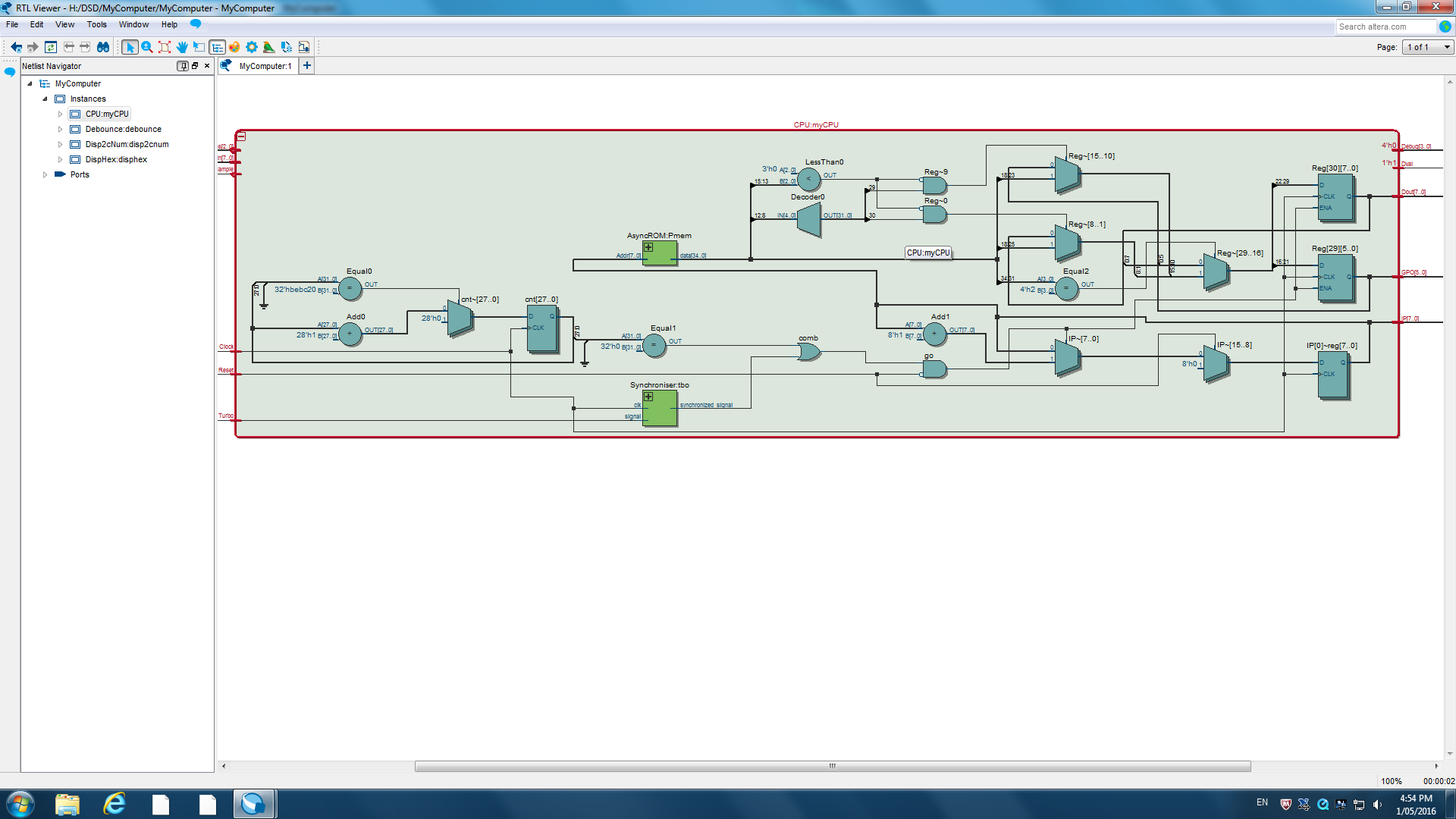
* Are letters or numbers being stores into the program memory?

Numbers are stored into the program memory.

**Mechanism**

Machine code: Machine codes are just binary numbers in a certain order to be read by CPU. In this case IP: {cmd\_grp, cmd, arg1\_type, arg1, arg2\_type, arg2, addr}. They are defined as constants in the CPU.vh head file for easy reading purpose. For example `MOV is 0010. When the computer reads this number at the beginning 4 bit of command, it will know it’s a command group and checks for further information to decide what to do with this line of command. Commands are generated with the AsyncROM module in ROM.v. IP as instruction pointer are generated in the CPU module representing a state and it’s automatically incremented by 1 every “go” cycle (state generator). AsyncROM returns a command for the CPU to process task at each different state (IP).

Stage 6



RTL Diagram for Stage 6

* How does the Turbo feature work?

When we turn on the Turbo pin, the CPU will ignore the counter which has a cycle of 250ms. CPU will run with a full speed which is 50 MHz by executing every command on the rising edge of the 50MHz clock.

* What does it mean to have synchronized turbo signal?

Having a synchronized turbo signal reduces the chance of having metastability.

* What can go wrong if we had used Turbo instead of turbo\_safe?

Having more chance of going to metastability.

Stage 7

Simulation and Testing:

The test for the move command is implemented with the instruction program given in Stage 10, with the line 4: data = {`MOV, `SHL, `REG, `DOUT, `REG, `DOUT, `N8};

It’s expected the computer to behave in the manner of: when IP the instruction pointer reach 4, shift the Reg[`DOUT] in CPU the left by one bit. More explanation of the behaviour of this command is given in Stage 10.

* What do get\_number and get\_location do?

“get\_number” function has two inputs arg\_type and arg. Arg\_type specifies the type of arg.

When the type is “`REG”, the type of arg (argument) is a register.

The function reads the number stored in the register specified in arg (argument). If the type is “`IND”, the function reads the number in the register which is specified by the content in the register indicated by the argument.

The default case deals with the case when argument type is “`NUM” and anything else.

The number is just the argument itself.

Get\_location function works in a similar way. Instead of getting the number (value), it outputs the location that to store the result of the operation.

These two functions make interpreting the instructions easier.

* What do the instruction `MOV `SHL and `MOV `SHR do exactly?

`MOV `SHL shifts the input number left by 1 bit and stores the result in the register specified in the command. `MOV `SHR does the similar thing. Instead of shifting the input number left, it shifts it to the right by one bit.

According to the project specification, RFLAG[`SHFT] register is used to store the bit which is being shift out. Therefore after each shift operation, there is an assignment made to the Flag Register to store the bit which is shifted out.

* **How is it possible that “cnum” is not synthesized? What does synthesizer do instead?**

**Why, with only a few lines of code, is there now a bird’s nest of wires.**

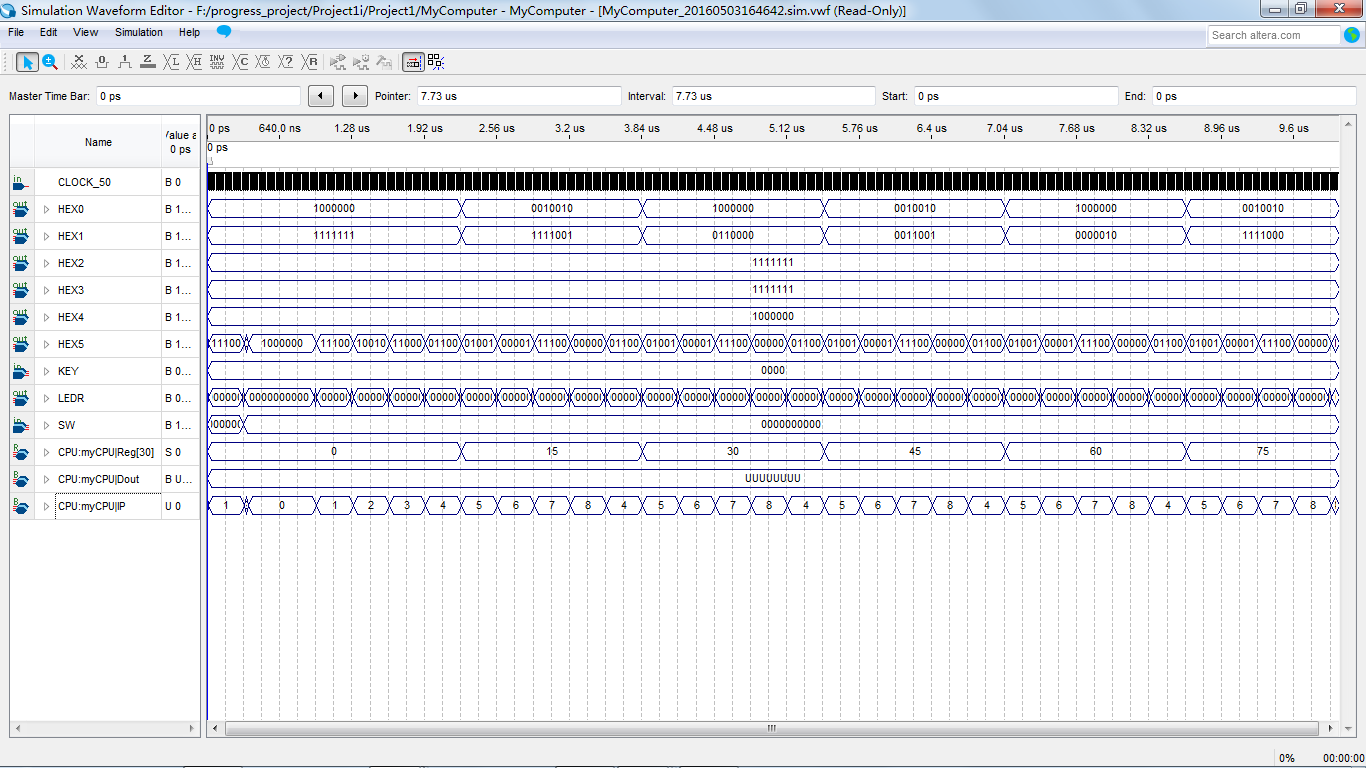
“cnum” is acting as a temporary variable. It is not involved in any real operations (e.g. storing data, getting assigned to another register). It is like a node on the wire. If we did not tell the software to express it in RTL explicitly (i.e. use (\*keep\*)), it will not be shown on the RTL diagram.

* Look at the RTL. Why, with only a few lines of code, is there a bird’s nest of wires?

Because in this stage, we have created 32 registers and there are 8 bits in every register, we will connect 256 wires when we make a single assignment in the case statement.

Stage 8

Simulation and Testing



Functional Simulation for Stage 8 by University Waveform Program

The computer is expected to:

1. Assign 8’d0 to Dout at IP = 0(when it’s 1 on the HEX[5:4] display);

2. When IP = 4, do unconditional addition (UAD) to reg[`DOUT] and 8’d15, which gives 0+15=15 in this case

3. When IP=8, unconditionally assign IP=4(Jump back to 4), and therefore repeate step 2, and do addition of 15+15=30.

4. Repeat until whenever IP=8

As the waveform below shows, the output of it matches up with our expectation.

* Describe exactly how each instruction works?

Described in Simulation and Testing section.

* Why does using the addresses 0, 4 and 8 introduce a delay?

The output is delayed by a clock cycle because the instruction being executed is always the one that IP is pointing to in the previous clock cycle.

* When the displayed Dout changes on the board, is the IP 4 or 5? Why?

The instruction is obtained before IP is incremented.

If we say IP + 1 is the "next IP", and the IP the "next IP" is assigned to is "current IP". The IP passed into "AsynchROM" is "current IP", and the instruction stored under "current IP" is saved into the register in CPU module. Then the command executed in the next clock cycle is the one stored in the instruction provided by the address "current IP". Therefore, the instruction is obtained before the IP is incremented. Because of this, the output is always delayed by one cycle.

Stage 9

Simulation and Testing

The test program is designed to be

0: data = {`MOV, `PUR, `NUM, 8'd 1, `REG, `DOUT, `N8};

4: data = {`ACC, `SMT, `REG, `DOUT, `NUM, -8'd 2, `N8};

7: data = {`JMP, `SLT, `REG, `DOUT, `NUM, 8'd64, 8'd 4};

10: data = {`MOV, `PUR, `NUM, 8'd 100, `REG, `DOUT, `N8};

13: data = {`ACC, `SAD, `REG, `DOUT, `NUM, -8'd 7, `N8};

16: data = {`JMP, `SLE, `NUM, 8'd 0, `REG, `DOUT, 8'd 13};

20: data = {`JMP, `UNC, `N10, `N10, 8'd 0}; default: data = 35'b0;

It’s expected to behave:

1. When IP=0, assign reg[`DOUT] with 8’d1;

2. When IP=4, do signed multiplication(SMT) between reg[`DOUT] and -8’d2, that is 1\*(-2)=-2, in this case;

3. When IP=7, check whether reg[`DOUT](-2 at the moment) is less than 8’d64 or not. (using the SLT command), if its true, jump back to IP=8’d4

4. Repeat 2 again until reg[`DOUT] is NOT less than 64, therefore in DOUT, 4, -8, 16, -32, 64 is expected

5. When IP=10, assign DOUT with 8’d100

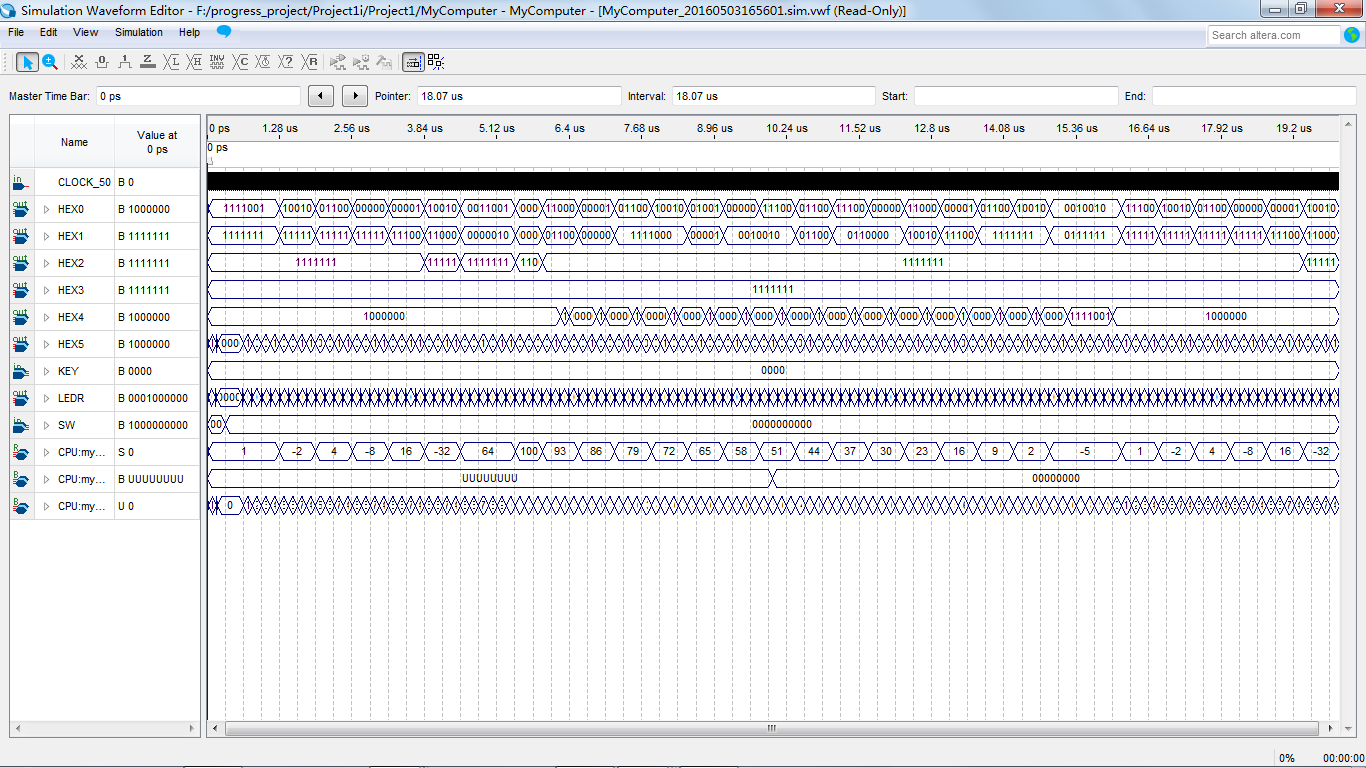
6. Move on to IP=13, do signed addition between DOUT(100 at the moment) with -8’d7, with is 100-7=93

7. When IP=16, check whether 8’d0 is less than DOUT or not, if it is, jump back to IP=13

8. Repeat step 6 until DOUT is less than 0, therefore, 93-7=86, 79, 72 until -5 is expected

9. When IP=20, unconditionally jump back to IP=0 and repeat everything

As the waveform shown below, it matched up with the expectation above completely



Functional Simulation for Stage 9 by University Waveform

* The thirty two registers of the CPU are all unsigned. Yet we allow signed comparisons. How does this work?

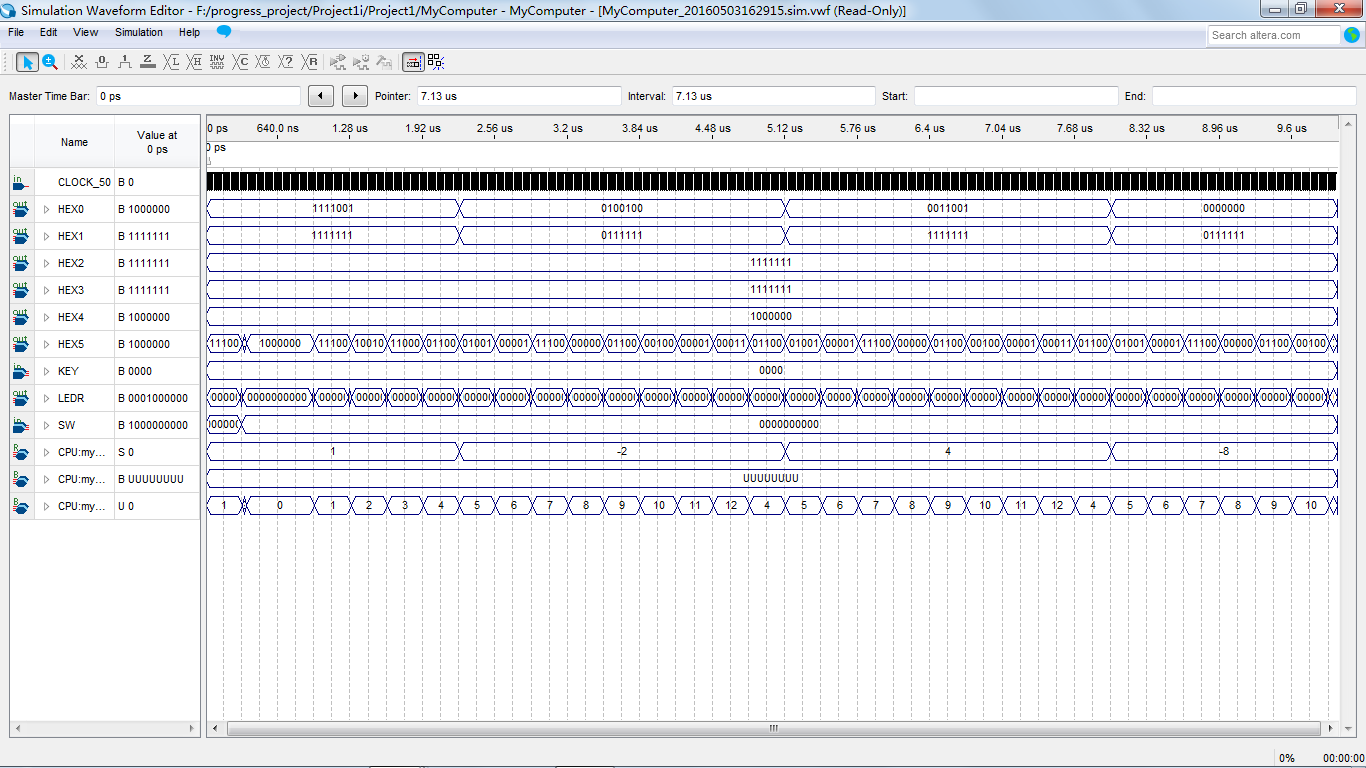
We use $signed() function to cast an unsigned value to a signed one.

* Explain why the above test program produces what it does.

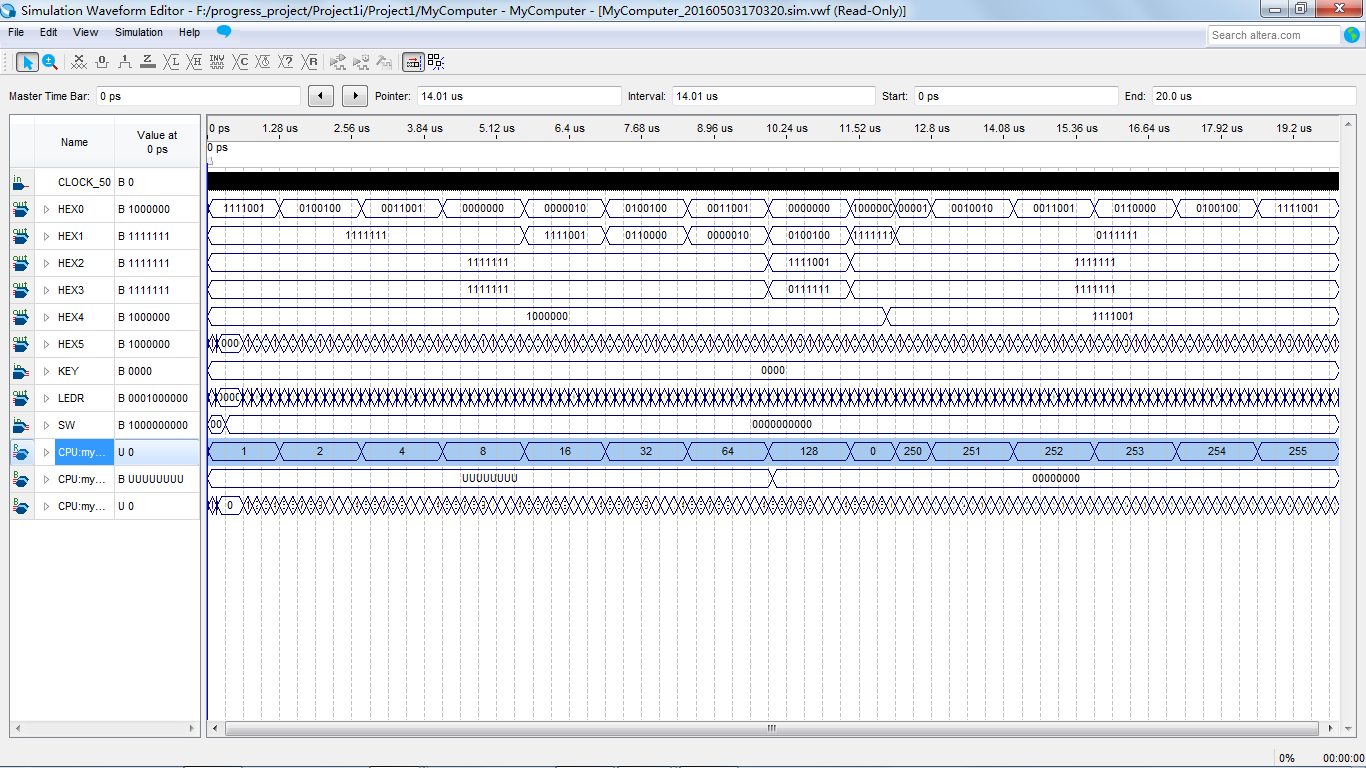
Described in Simulation and Testing section above.

Stage 10

Simulation and Testing



Simulation result obtained by University Waveform Program



Simulation Result Obtained by the University Waveform Program (After two lines of change)

* What is being stored into program memory now?

Binary numbers are stored into the program memory.

* What does the first program do?

If one bit of the RFLAG register is HIGH, jump to a specific address. After this, turn off that bit of the RFLAG register.

* What about the two lines of change

These two lines keep left shifting DOUT register until the bit is shifted out is one. If the bit is one, jump to address 16.

* Why

Modify the instruction pointer under a certain condition, depending on a specific bit of a specific register.

Stage 11

* What does the line “assign Dval = Rgout[`DVAL];” do?

This line connects Dval output pin to Register Rgout[`DVAL].

* Why do we want that line?

By doing this, we can control the output of Dval by setting up commands in ROM.

Without this line, Dval cannot be modified in the always block because it is not a register.

* What change did you have to make to your program?

After making Dval a wire and assigning it to a register Rgout[`DVAL], a command to turn on the display is added to the first address.

* Does your computer have to evaluate 8’b1 << bit every time you want to set or clear a bit?

Yes. Although “<<” is a left shift operation (like a shift register), there is no register storing the value of the shifted version. Therefore, 8’b1 << bit is evaluated every time when it is called.

Mechanism: Microprocessor

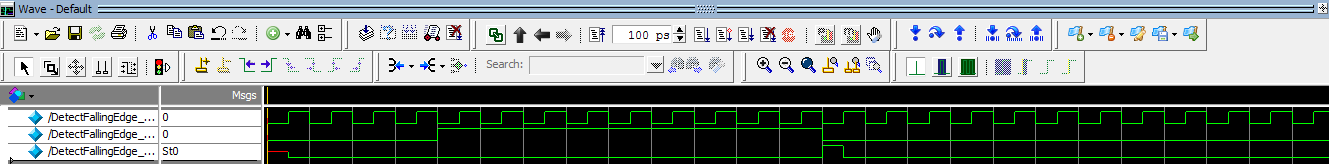
Microprocessor architecture is learnt over the development of the CPU module.

Generates instruction pointer,

The CPU module is designed to be able to read command which written in a specific format. Inside the CPU module there’s an instruction loop with different cases for the computer to understand machine code, and perform tasks over these codes. For instance in the cmd\_grp case the microprocessor recognise this is going to be a move command and enter this case, if SHR is detected shift right will be performed.

Step 2 Stage 12

Simulation



Simulation of DetectFallingEdge module in ModelSim

* Why is a flip-flop required

We need a flip flop to store the old value (the value one clock cycle before the current value).

The output is HIGH (when a falling edge is detected) when the old value is HIGH and the current value is LOW.

* Why do we not want to use two or more flip flops?

The value representing the status of button has already been passed into a synchronizer (i.e. this is a safe version of button value). There is no need to put extra flip flops in this module.

Cascading flip flops will create delays. Therefore only one flip flop is used.

* Why not use negedge?

The whole design is a synchronized design. Every new value is sampled on the rising edge of the clock (either the fast clock or the slow clock). It is very unlikely to sample the negedge of the button on the posedge of the clock (The transition happens instantly).

Step 3 Stage 12

* What might we want the CPU to automatically clear the Flag Register after a Reset?

After Reset, all the registers should be cleared. The Instruction Pointer should be pointed to the default address in ROM (usually address 0). RDINP register should be set back to 0. RFLAG should also be cleared, as well as DOUT and GOUT.

* What does the above verilog code do?

The generate block generates four DetectFallingEdge blocks. The inputs are four synchronized push buttons and the outputs represents whether the buttons are pressed or not.

The if-else statement is basically connecting the button to registers.

According to the previous specification, 0~3 bits are recording the status of the KEYs (i.e. buttons) on the board. Therefore when a button is pressed (pb\_activated is HIGH), the corresponding bit in the RFLAG register is HIGH. Among the four buttons, the fourth one is the special one. When the fourth button is pressed, the number on the switches (input pins) are copied into the RDINP register, which is the register recording the input values.

* What happens if the Program Memory contains an instruction to store a value into Register 28 (`RDINP)? Will the value stay there? For how long? Which takes precedence if both a move instruction and the hardware try to change `RDINP at the same time?

If both a move instruction and the hardware try to change `RDINP at the same time, the change brought by the hardware will overwrite the values moved in `RDINP by the instruction. According to the implementation of CPU, after execution of the instruction, the values on the switches are registered in the RDINP. Therefore the previous value in RDINP moved in by the instruction is lost.

The value moved in the register by instruction will stay there until it is overwritten by the hardware change.

* Is it possible for `RDINP to change halfway through when the programmer tries to move `RDINP into another register? When Turbo mode is off?

It is highly unlikely for `RDINP to change halfway through when the programmer tries to move ‘RDINP into another register. Because we have connected the input from switches to the debounce blocks, the probability of metastability is reduced.

* Why might we have chosen to use the falling edge rather than the rising edge, or in other words, why do we want to detect push button releases rather than push button presses?

Detecting push button releases will prevent multiple times of assignment to a register when the buttons are pressed.

Part 2 RPN Calculator

Implementation of the RPN Calculator

* STACK1: the register to store the most recent number

STACK2: the register to store the second recent number

STACK3: the register to store the second oldest number

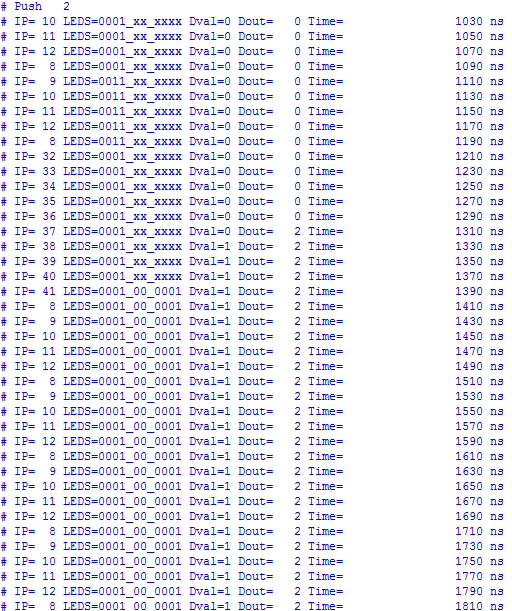
STACK4: the register to store the oldest number

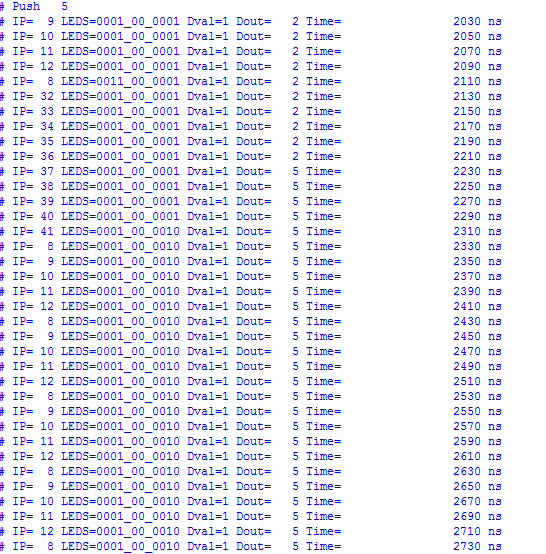
STACKF: the register to keep record of the number of numbers in the stacks

* Using the test bench provided
* The result below shows that the push function was correctly implemented.
* Some modification was made to the original design:
  + ¡ `DVAL bit, which is used to indicate turning on the display, was assigned to the 8th bit of the RGOUT register. RGOUT register linked with LEDRs, while LEDRs are also related to the register storing the number of numbers in the stack. Because the maximum number of numbers can be stored in the stack is 4, using one-hot-encoding style, the most four significant bits in the register recording this value is always zero. When we move this register to RGOUT (LEDRs), the most significant bit (DVAL) will be turned off. To avoid this, DVAL bit is changed to the most significant bit of RFLAG register. By doing this, DVAL will always be on after it is switched on until it is turned off by some instructions in the program.
* l When one number is pushed into the stack, the register recording the number of the numbers entered is shifted to the left by one shift. However, according to the implementation of the SHL command, the result from the shift operation will remain in 0s if the original number is 0. Therefore, an “if” condition is added to the `MOV `SHL implementation. If the number to be shifted is 0, then the bit shift in is 1, otherwise it will just operate in the ways it is implementation before.

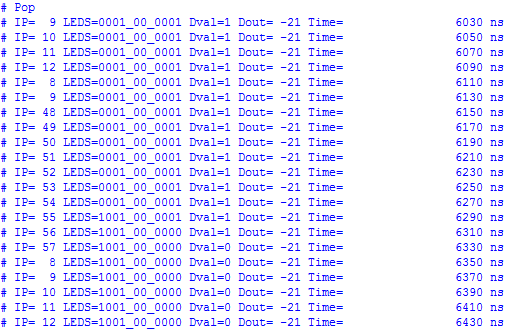
Simulation

Push:





Pop:



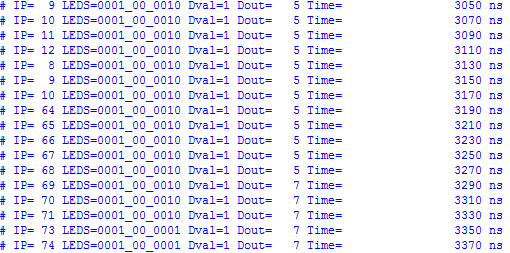
Verification:

|  |  |  |  |
| --- | --- | --- | --- |
| STACK0 | STACK1 | STACK2 | STACK3 |
| -21 | x | x | x |

After:

|  |  |  |  |
| --- | --- | --- | --- |
| STACK0 | STACK1 | STACK2 | STACK3 |
| x | x | x | x |

Addition:



Verification:

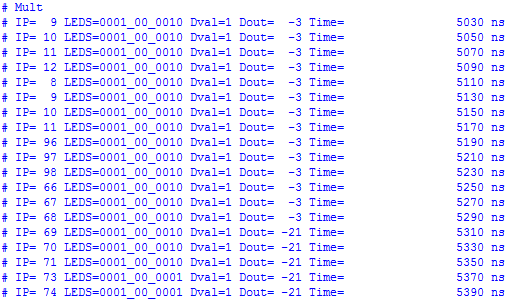
Before addition:

|  |  |  |  |
| --- | --- | --- | --- |
| STACK0 | STACK1 | STACK2 | STACK3 |
| 2 | 5 | x | x |

After addition:

|  |  |  |  |
| --- | --- | --- | --- |
| STACK0 | STACK1 | STACK2 | STACK3 |
| 7 | x | x | x |

Multiplication:



Verification:

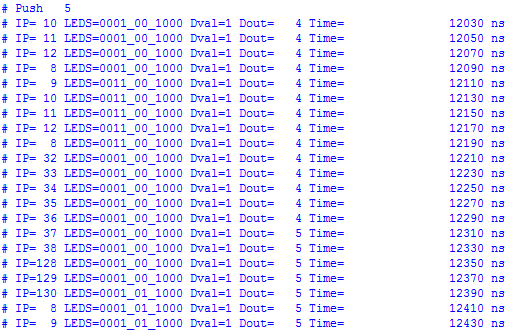
Before multiplication

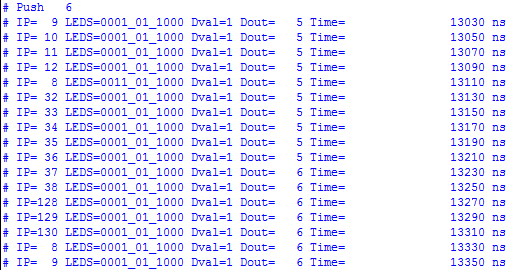
|  |  |  |  |
| --- | --- | --- | --- |
| STACK0 | STACK1 | STACK2 | STACK3 |
| 7 | -3 | x | x |

After

|  |  |  |  |
| --- | --- | --- | --- |
| STACK0 | STACK1 | STACK2 | STACK3 |
| -21 | x | x | x |

Stack overflow:





Before:

|  |  |  |  |
| --- | --- | --- | --- |
| STACK0 | STACK1 | STACK2 | STACK3 |
| 4 | 3 | 2 | 1 |

After push 5:

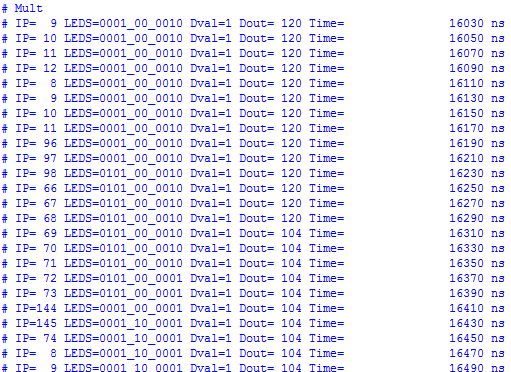
|  |  |  |  |
| --- | --- | --- | --- |
| STACK0 | STACK1 | STACK2 | STACK3 |
| 5 | 4 | 3 | 2 |

The 5th bit of GOUT should be lit on.

After push 6:

|  |  |  |  |
| --- | --- | --- | --- |
| STACK0 | STACK1 | STACK2 | STACK3 |
| 6 | 5 | 4 | 3 |

Arithmetic Overflow:



Verification:

|  |  |  |  |
| --- | --- | --- | --- |
| STACK1 | STACK2 | STACK3 | STACK4 |
| 120 | 3 | X | X |

After:

|  |  |  |  |
| --- | --- | --- | --- |
| STACK1 | STACK2 | STACK3 | STACK4 |
| 104 | X | X | X |

360 = 101101000 which is a 9 bit long number, but the register is 8 bit long, therefore, it is truncated to a 8 bit number 0110100, which is 104 in decimal.

Mechanism:

Program

A program is a collection of command containing machine code to drive the CPU to perform tasks. It was implemented as different states in this project. By changing from state to state, the microprocessor is able to read and execute different command contained in the stage perform tasks (do calculations) as a RPN calculator.

Overall slack

The speed of the program (computer) is limited by Fmax given by Quartus software. It is important that the speed of the clock do not go over Fmax, which is considered as the maximum speed of the design. This measure depends on the longest delay along any path, called the critical path, between two registers clocked by the same clock.