Digital System Design 2021 Project – Computer and RPN Calculator

Lei Qin

May 2021

Contents

1 Part 1

1.1 Stage 1

Get start with the top-level module MyComputer with the RTL diagram shown in figure 1.1.

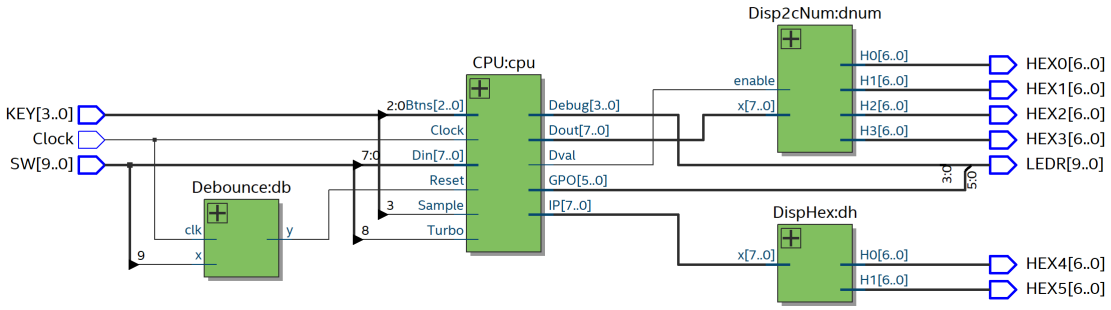

Figure 1.1: Stage 1 RTL diagram.

1.2 Stage 2

External inputs must be synchronised, to avoid metastability. Furthermore, we are told that the reset switch needs debouncing, and of a minimum 25 ms duration.

Create a module Synchroniser (in AuxMod.v) that takes as input the 50 MHz clock and an input wire, and by using a double flip-flop synchroniser, produces a synchronised version of the input signal shown in 1.2.

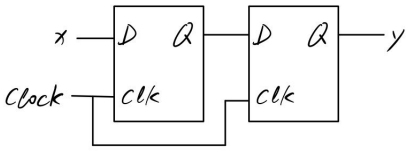

Figure 1.2: A double flip-flop synchroniser.

Fill in the Debounce module shown in figure 1.3, so that it operates as follows. First, send the input through a Synchroniser. Then implement a FSM that functions in an equivalent way to the following description. If the synchronised input remains a 1 for 30 ms then

output a 1. If the synchronised input remains a 0 for 30 ms then output a 0. Otherwise, do not change the output.

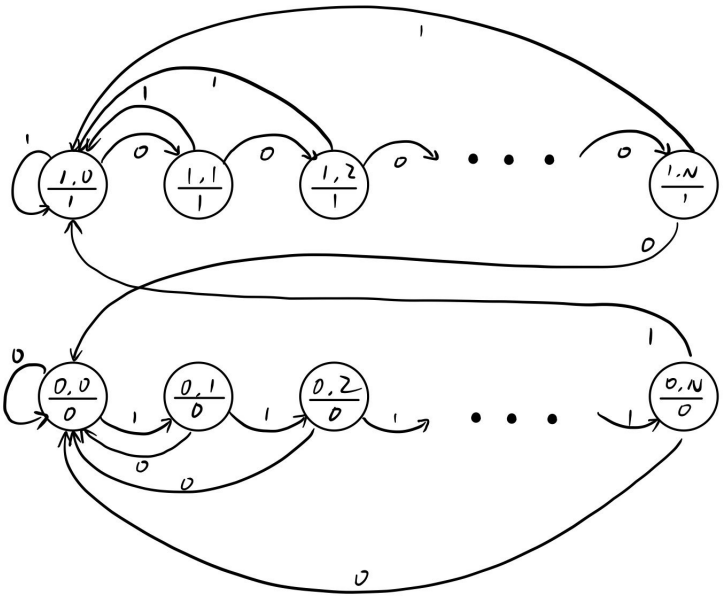

Figure 1.3: A debounce FSM.

The RTL diagram is shown in figure 1.4.

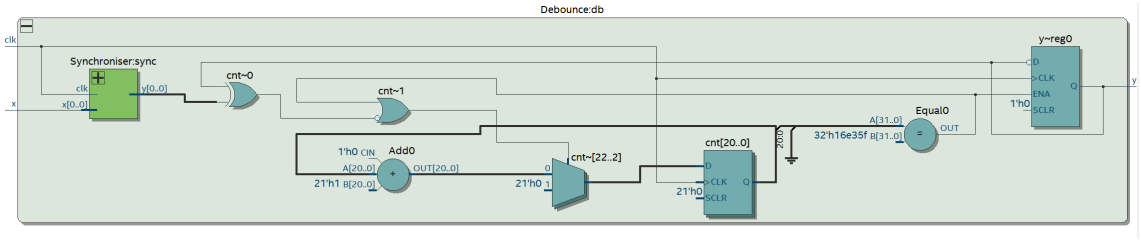

Figure 1.4: Stage 2 RTL diagram.

1.3 Stage 3

Simulate the design to see if it works as expected:

When the Reset pin is HIGH, all the output pins will be set to 0.

When the Reset pin is LOW, the CPU should run, which in this case, just means setting all the GPO pins to 1.

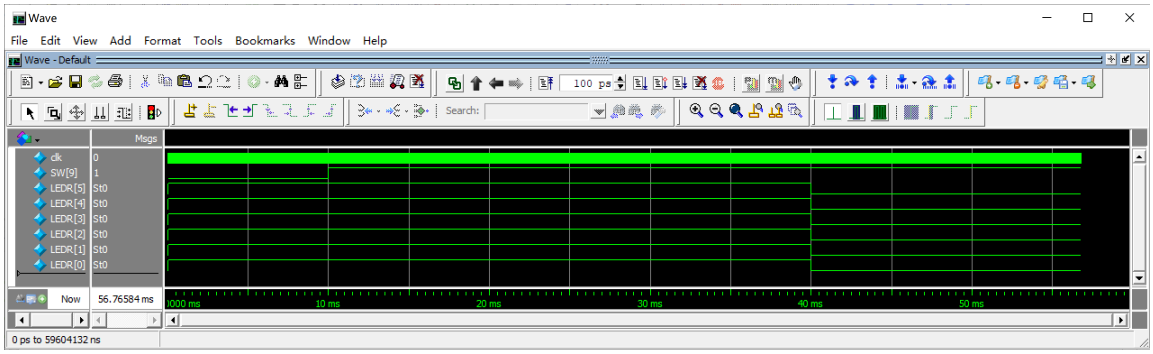

Figure 1.5: Stage 3 RTL simulation.

From figure 1.5, the SW[9] (reset) is low, and the LEDR[5:0] (GPO) is high. At 10ms, the SW[9] (reset) goes high, then after 30ms due to the debounce, the LEDR[5:0] (GPO) becomes low.

1.4 Stage 4

Fill in the modules Disp2cNum and DispHex. The logic for DispDec is :

```
1
2 wire [3:0] digit = x % 10;  // the remainder after
    dividing by 10
3 wire n = (neg) && (x == 0);  // if display a negative
    sign
4 SSeg converter(digit, n, enable, segs);
5 always @(*) begin
6         xo = x / 10;
7         eno = (enable != 0) & ((xo != 0) | ((neg)&&(x !=
              0)));  // digit or neg sign
8 end
```

Then the RTL diagram is shown in figure 1.6.

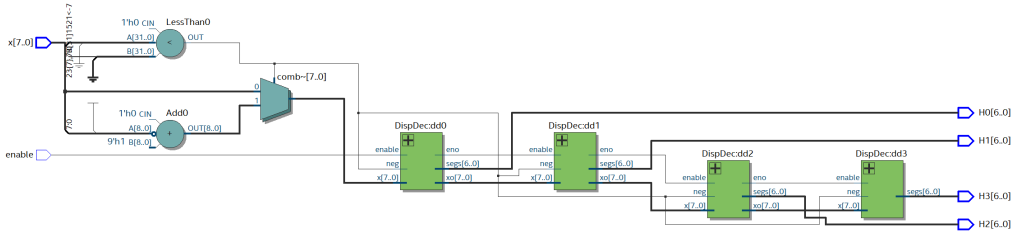

Figure 1.6: Stage 4 RTL simulation.

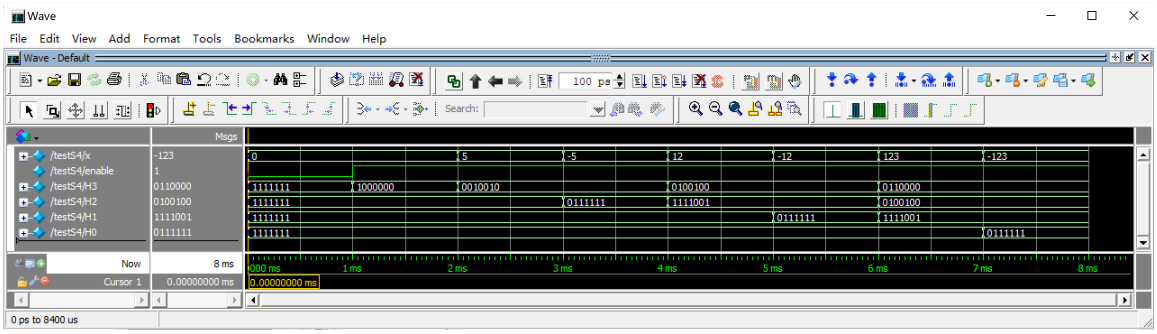

Figure 1.7: Stage 4 RTL simulation.

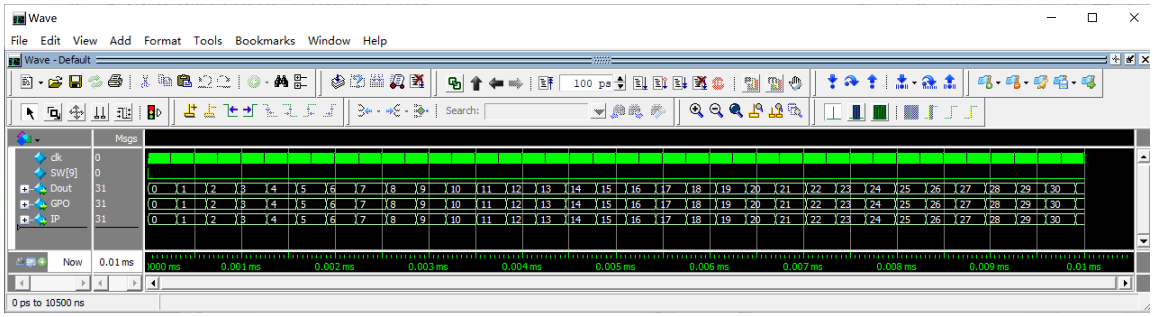

Figure 1.8: Stage 4 increment RTL simulation.

From figure 1.7, we can verify Disp2cNum module can display [0, -5, 5, 12, -12, 123, -123]. From figure 1.8, we can see when SW[9] (reset) is high, the LEDs (IP and Dout) and GPO is 0. When SW[9] (reset) is low, the LEDs (IP and Dout) and GPO counts from 0.

1.5 Stage 5

1.5.1 Step 1

The logic for counter is :

```
1 //Clock circuitry (250 ms cycle)
2 reg [23:0] cnt = 1;
3 localparam CntMax = 12500000;
4 always @(posedge Clock)
5         cnt <= (cnt == CntMax) ? 0 : cnt + 1;
```

Here is a tip that we don't want to set the wire 'go' at 0 time, so we initialise cnt as 1 and make 'go' go high when cnt is counted to 0.

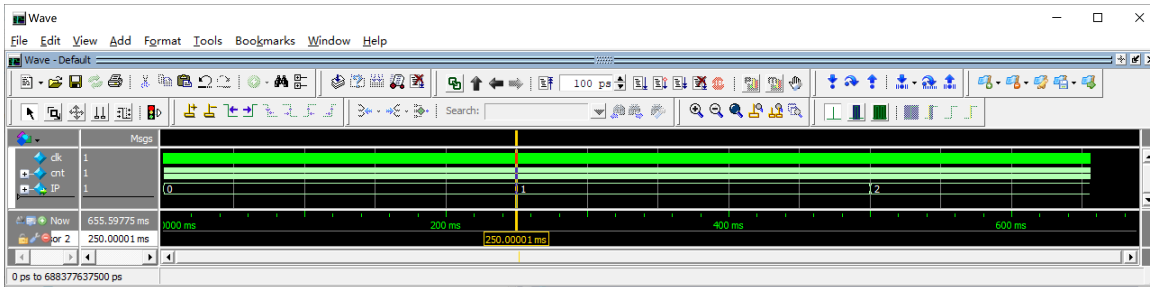

Figure 1.9: Stage 5 step 1 increment RTL simulation.

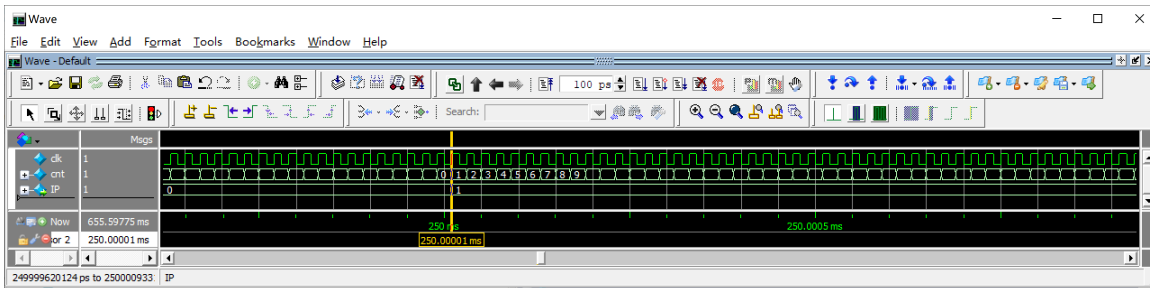

Figure 1.10: Stage 5 step 1 zoom in RTL simulation.

From figure 1.9 and 1.10, we can see IP increases from 0 correctly.

Question

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 (Reset)”?

This is because IP is a reg which should be driven by only one always block. And if it is having multiple drivers by design, it would resolved by multiple constant drivers and would suffer from race condition.

1.5.2 Step 2

We need to create the Program Memory, and place a program in there for the CPU to execute.

```
0: data = {`MOV, `PUR, `NUM, 8'd 1, `REG, `DOUT, `N8};
1: data = {`MOV, `PUR, `NUM, 8'd 3, `REG, `DOUT, `N8};
2: data = {`MOV, `PUR, `NUM, 8'd 5, `REG, `DOUT, `N8};
3: data = {`MOV, `PUR, `NUM, 8'd 7, `REG, `DOUT, `N8};
4: data = {`MOV, `PUR, `NUM, 8'd 9, `REG, `DOUT, `N8};
5: data = {`MOV, `PUR, `NUM, 8'd 11, `REG, `DOUT, `N8};
6: data = {`MOV, `PUR, `NUM, 8'd 13, `REG, `DOUT, `N8};
7: data = {`MOV, `PUR, `NUM, 8'd 15, `REG, `DOUT, `N8};
8: data = {`MOV, `PUR, `NUM, 8'd 17, `REG, `DOUT, `N8};
```

```
10  9: data = {‘MOV, ‘PUR, ‘NUM, 8’d 19, ‘REG, ‘DOUT, ‘N8};
11  default: data = 35’b0; // Default instruction is a NOP
```

Then when you run your computer, the display will show 1,3, 5, 7, 9, 11, 13, 15, 17, 19 then remain on 0 until the IP reaches 00 again shown in figure 1.11.

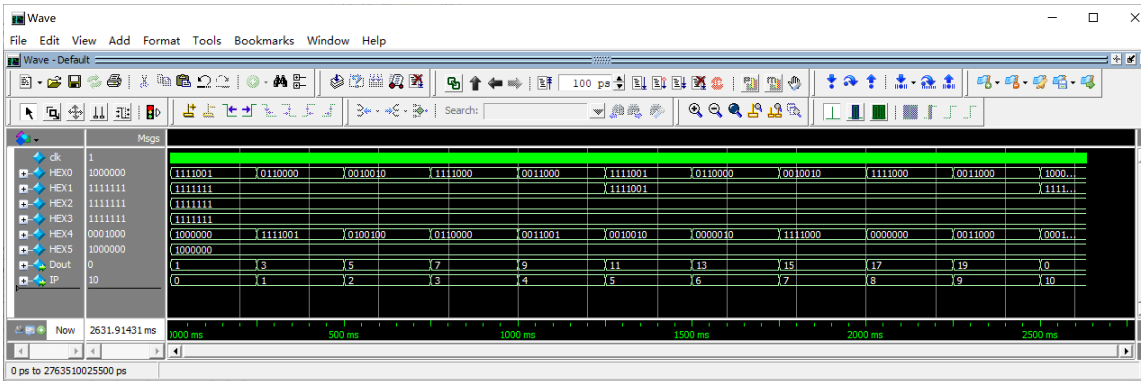

Figure 1.11: Stage 5 step 2 in RTL simulation.

Question

- How does Pmem(IP, instruction) work?

  Given an IP, such as {xx}, we can get a corresponding instruction from Pmem like {xxxx_xxx_xx_xxxxxxxx_xx_xxxxxxxxx_xxxxxxxx}, which indicates the Command Group in the first 4 bits, Command in the next 3 bits, type and data of Argument 1 in the next 10 bits, type and data of Argument 2 in the next 10 bits, address used for the JMP and ATC Command Groups in the last 8 bits.

- What does the test code do?

  We initialise Dval to 1 and let Dout to be the data of Arhument 1.

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

  This is because when IP is larger than 9, the default instruction is all NOP (no operations) {0000_000_00_00000000_00_00000000_000000000}. Then the data of Argument 1 is 8’b0, thus LED (Dout) is 0 after showing 19.

- Are letters or numbers being stored into the program memory?

  NO, because the program memory (read only memory) is to store instructions to operate on 32 registers which means you can’t store variables in the program memory. Data memory is a counterpart of program memory and it can store data.

1.5.3 Step 3

This should display 1,3,5 etc when you run the program, due to the instructions loaded into memory in Step 2. Add several more instructions, to get some LEDs to light up. (Move a number into the 'GOUT register.) You have built a very simple computer!

Here is a tip that we do not want IP to increase at 0 time, but we want to do the command at 0 time. So we create a new wire go_cmd to trigger command in ROM as:

```
always @(posedge Clock) begin
        // Process Instruction
        if (go) begin
                IP <= IP + 8'b1;  // Default action is to
                    increment IP
        end

        if (go_cmd) begin
                case (cmd_grp)
                        'MOV:
                                Reg[arg2] <= arg1;
                // For now, we just assumed a PUR move,
                   with arg1 a number and arg2 a register
                endcase
        end

        // Process Reset
        if (Reset) IP <= 8'b0;
end
```

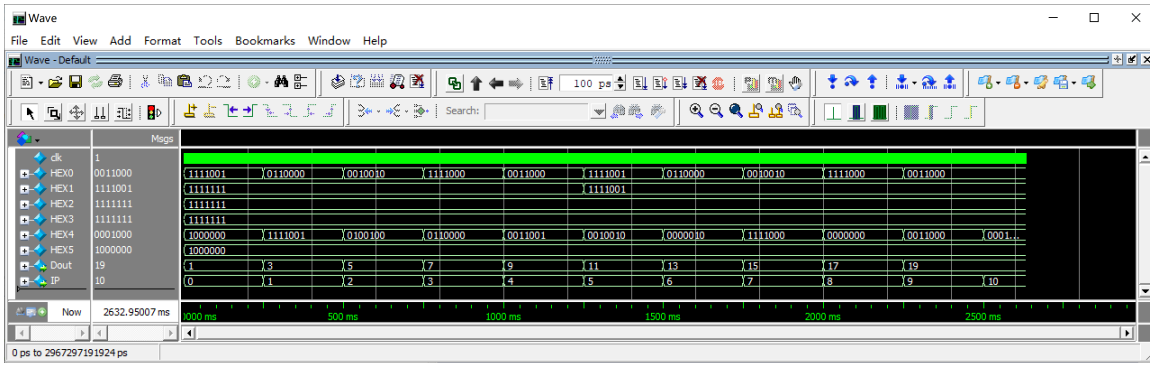

Figure 1.12: Stage 5 step 3 in RTL simulation.

From 1.12, we can find every 250ms, IP increases by 1 and Reg[30] (Dout) is updated by the instruction (pure move) from ROM: move the number from 8'd1 to 8'd19 to the Reg[30]

(Dout).

1.6 Satge 6

Have a look at the RTL for your CPU in your figure 1.14. This will be the last stage where the RTL is relatively straightforward to take in at a glance.

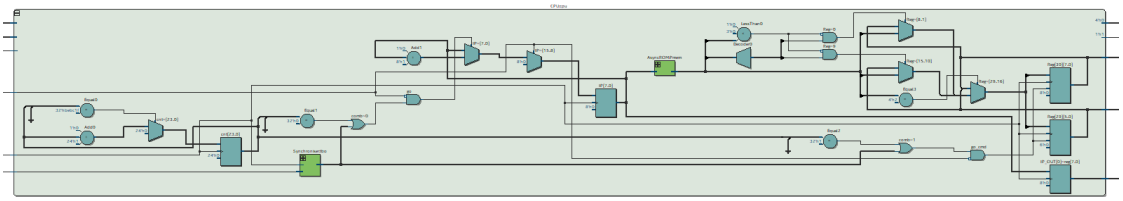

Figure 1.13: Stage 6 in RTL diagram.

Implement the Unconditional Jump instruction, and test it out by adding a JMP to your Program Memory so that your program loops after ten or so instructions. You can also implement the Turbo feature now shown in figure 1.14.

In turbo mode, we also want to do the command at 0 time, but not want IP to increase at 0 time. So we create a new delayed IP_OUT as output:

```
always @(posedge Clock) begin
        IP_OUT <= IP;
end
```

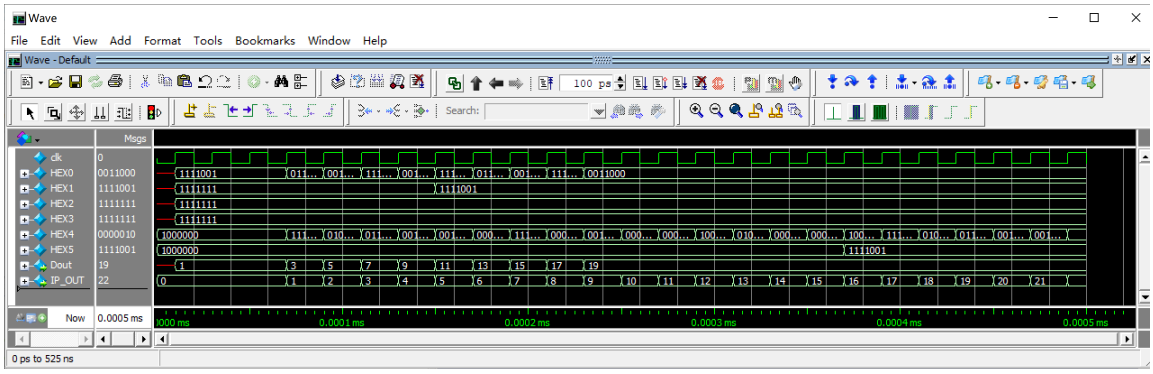

Figure 1.14: Stage 6 in RTL simulation.

Question

- How does the Turbo feature work?

  We execute the command from ROM every posedge of clock (20ns) not every 250ms.

- What does it mean to have synchronised the turbo signal?

  It is to avoid metastability of turbo signal by two flip-flops synchroniser.

- What can go wrong if we had used Turbo instead of turbo_safe?

  If we do not synchronise turbo signal, the turbo maybe metastable, thus the wire 'go' would also be metastable, which would lead the cpu totally go wrong.

1.7 Part 7

The full repertoire of the MOV Command Group will now be implemented. Make sure your original program still executes correctly. Then change some of the instructions to use the new shifting ability of MOV commands:

```
0: data = {`MOV, `SHL, `NUM, 8'd 1, `REG, `DOUT, `N8};
1: data = {`MOV, `SHL, `NUM, 8'd 3, `REG, `DOUT, `N8};
2: data = {`MOV, `SHL, `NUM, 8'd 5, `REG, `DOUT, `N8};
3: data = {`MOV, `SHL, `NUM, 8'd 7, `REG, `DOUT, `N8};
4: data = {`MOV, `SHL, `NUM, 8'd 9, `REG, `DOUT, `N8};
5: data = {`MOV, `SHR, `NUM, 8'd 11, `REG, `DOUT, `N8};
6: data = {`MOV, `SHR, `NUM, 8'd 13, `REG, `DOUT, `N8};
7: data = {`MOV, `SHR, `NUM, 8'd 15, `REG, `DOUT, `N8};
8: data = {`MOV, `SHR, `NUM, 8'd 17, `REG, `DOUT, `N8};
9: data = {`MOV, `SHR, `NUM, 8'd 19, `REG, `DOUT, `N8};
default: data = 35'b0; // Default instruction is a NOP
```

Check that it works shown in figure 1.15: left shift the first 5 numbers and right shift the next 5 numbers.

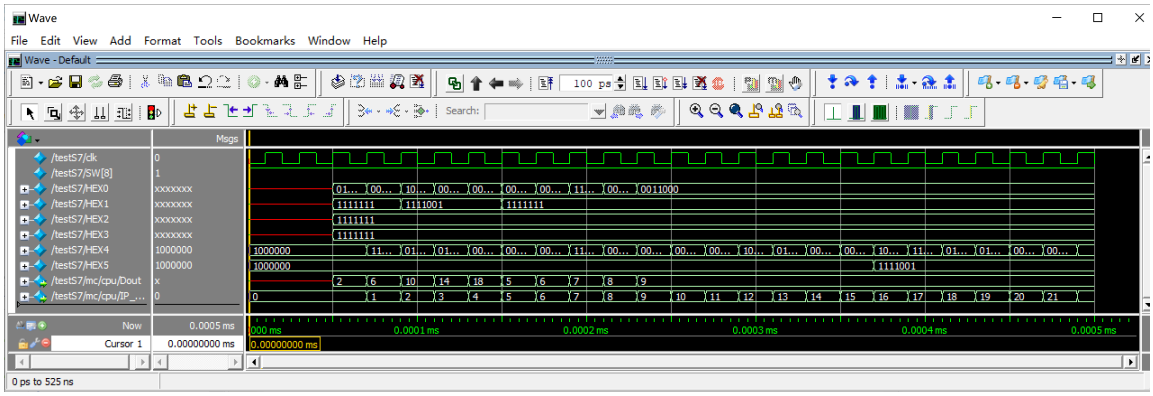

Figure 1.15: Stage 7 in RTL simulation.

Question

- What do get_number and get_location do? How do they relate to the CPU Instruction Set?

  get_number: If input is ‘REG typed, then return the content of that register; if input is ‘IND typed, then return the content of a register which is indexed by a register related to that input.

  get_location: If input is ‘REG typed, then return that register; if input is ‘IND typed, then return the register which is indexed by a register related to that input.

  We execute these two functions by arg_type and arg from ROM as inputs and return the data for CPU Instructions like ‘MOV and ‘ACC.

- What do the instructions ‘MOV ‘SHL and ‘MOV ‘SHR do exactly? Why is an assignment made to the Flag Register?

  First get the number by arg1 data and do left/right shift(multiplied/divided by 2) to the number. Then get the location by arg2 data. At last, assign the register according to the location by arg2 with the operated number.

  This is because we want to know what number is to be shifted out.

- How is it possible that “cnum” is not synthesised? What does the synthesiser do instead?

  "cnum" is not synthesised because ’cnum’ is just a intermediate variable to process with ’arg’ register.

  The synthesiser automatically converts the Verilog HDL hardware model into a gate-level implementation and maps it to the target technology. Synthesiser also optimizes the design for a given set of constraints related to area and speed. Therefore, ’cnum’ is ignored in synthesis.

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

  This is because we add two functions ’get_number’ and ’get_location’ with operations on the index of registers. Thus many combinational logics and wires used to instantiated these two functions.

1.8 Part 8

Using get_number and get_location, it is not difficult to implement the ACC Command Group. It is expedient to introduce several more temporary variables that do not get synthesised.

Here is a change in Command ‘ACC:

```
‘RFLAG[‘OFLW] <= ($signed(s_word) > 127 || $signed(s_word
    ) < -128);
```

Test out your computer with the following instruction.

```
0: data = {‘MOV, ‘PUR, ‘NUM, 8’d 0, ‘REG, ‘DOUT, ‘N8};
4: data = {‘ACC, ‘UAD, ‘REG, ‘DOUT, ‘NUM, 8’d 15, ‘N8};
8: data = {‘JMP, ‘UNC, ‘N10, ‘N10, 8’d 4};
```

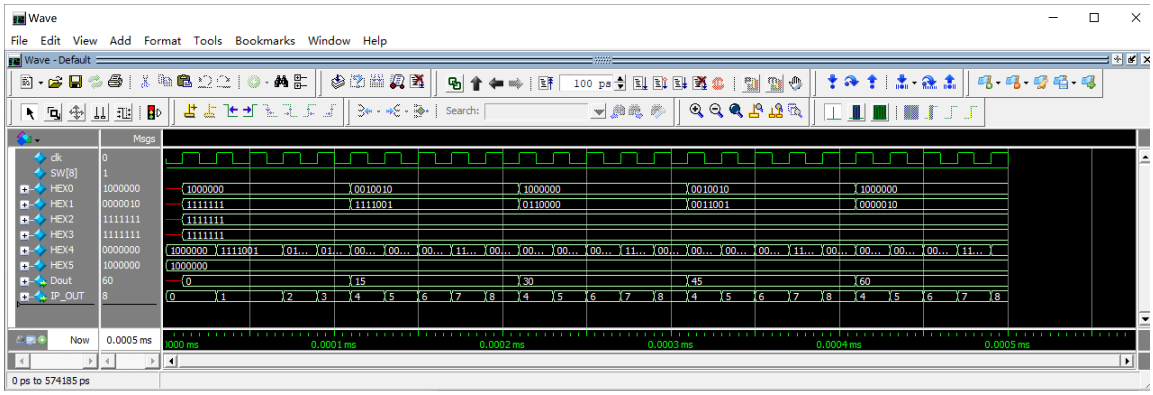

Figure 1.16: Stage 8 in RTL simulation.

From figure 1.16, we can see Dout is added by 15 every IP increases by 4.

Question

- Describe exactly how each Instruction works.

  0: purely move a number 8’d0 to a register Reg[‘Dout(30)];

  4: do a unsigned addition to the content of a register Reg[‘Dout(30)] by a number 8’d15;

  8: unconditionally jump the instruction to 8’d4.

- Why does using the addresses 0, 4 and 8 introduce a delay? This is because in one block, they are non-blocking assigned, so the command is executed from the last IP.

  This is solved by IP_OUT in Stage 6.

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

  IP = 5

1.9 Part 9

Implement in full the JMP Command Group. Test it out with the following instruction.

```
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;
```

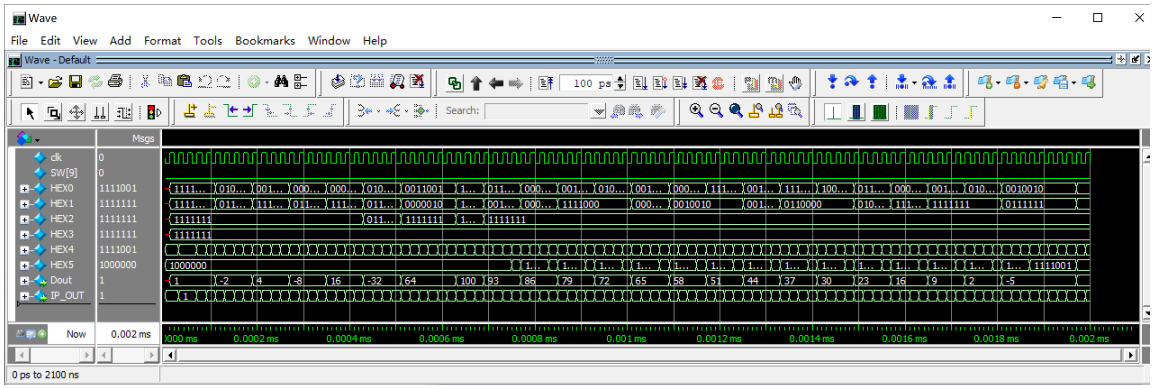

Figure 1.17: Stage 9 in RTL simulation.

From figure 1.17, we can see the output firstly is multiplied by (-2) until it is not smaller than 64, than it is added by (-7) from 100 until it is smaller or equal to 0. Next, it would jump to 0th instruction and run for the next loop.

Question

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

  We convert these to signed value by $signed function.

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

  The output firstly is multiplied by (-2) until it is not smaller than 64, than it is added by (-7) from 100 until it is smaller or equal to 0. Next, it would jump to 0th instruction and run for the next loop.

1.10 Part 10

The ATC Command Group is designed for reading the status of the push buttons. Notice the use of functions for syntactic sugar. From figure 1.18, we can see the LEDR[4](‘OFLW) is high after the 8 bit digits overflows.

After the Dout reaches 0, it is jumped to IP = 4 and run in a loop of multiplication by 2.

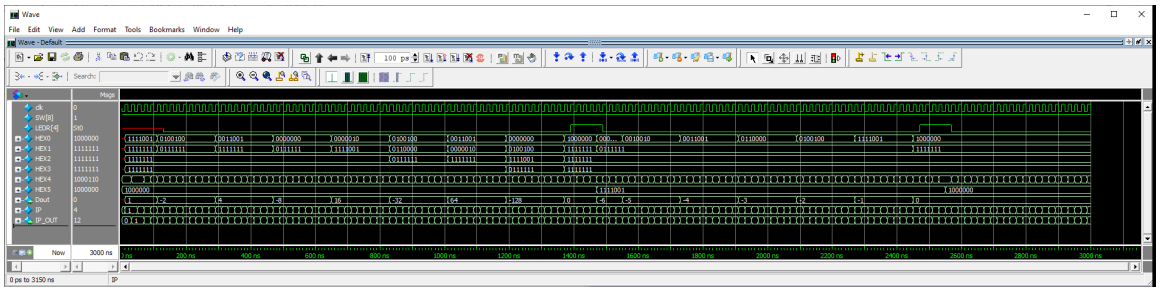

Figure 1.18: Stage 10 in RTL simulation.

Also try the move-with-shift instruction by making the changes to the above. From figure 1.19, we can see the LEDR[4](‘OFLW) is high after the 8 bit digits overflows and LEDR[5](‘SHFT) is high after the 8 bit digits shifts out 1.

After the Dout reaches 0, it is jumped to IP = 4 and run in a loop of multiplication by 2.

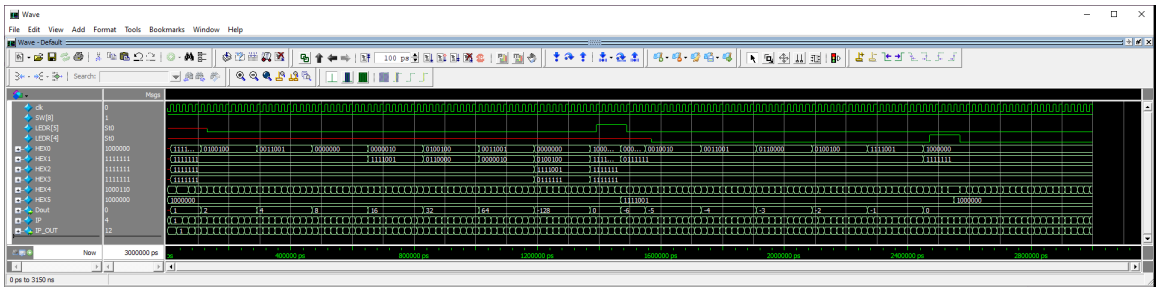

Figure 1.19: Stage 10 change in RTL simulation.

Question

- What is being stored into program memory now? Functions? Letters? Or binary numbers?

  Binary numbers. It is still a 35 bits binary number as an instruction.

- What does the first program do, and why?

  It is designed for reading the status of the push buttons by test the ‘RFLAG and clear it.

- What about the two lines of changes, related to move-with-shift? What should they do?

  Move with left shift is the same as signed multiplication by two. It shifts the input number left by 1 bit and show the shifted bit in Gout.

- Why might an atomic instruction, such as ATC, be important in a CPU?

  Test-and-set helps us fix that problem by checking that the value your overwriting is what you think it should be. In this case, you can check that the balance was the original value that you read. Since it's atomic, it's non-interruptible so no-one can pull the rug out from under you between the read and the write.

1.11 Part 11

There is a loose end to tidy up. Remove the line "initial Dval = 1;", change Dval to a wire. Setting and Clearing Individual Bits. Modify your program so that the display continues to work. We set the Reg[‘FLAG][‘DVAL] as IP = 2.

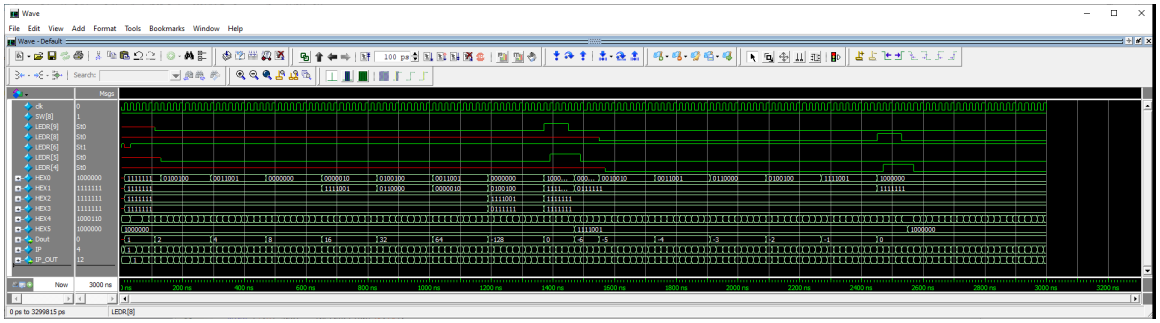

Figure 1.20: Stage 11 in RTL simulation.

From figure 1.20, LEDR[6](go) is high after IP is 2. LEDR[9] and LEDR[5](‘SHFT) goes high after bit 1 is shifted out while LEDR[8] and LEDR[4](‘OFLW) goes high after there is a overflow in the addition or multiplication.

Question

- What does the line "assign Dval = Rgout[‘DVAL];" do? Why do we want that line?

  Assign the value of Rgout[‘DVAL] to Dval otherwise Dval is unknown.

  Connect Dval to Reg[Rgout]. This is because we want to control whether to display by operating the Reg[Rgout]. We want to know if the output is valid to show on the board. If there is an overflow then we cannot store the correct answer - but we can light a LED to warn the user there is an overflow.

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

  Add setting the Reg[‘FLAG][‘DVAL] as IP = 2:

```
2: data = set_bit(‘FLAG, ‘DVAL);
```

- Does your computer have to evaluate 8’b1 « bit every time you want to set or clear a bit? Explain.

  No. If bit = 0, there is no need to left shift 8’b1.

1.12 Part 12

1.12.1 Step 1

Create a wire [7:0] din_safe and assign it to the outputs of eight Synchroniser modules whose inputs are the respective bits of Din. Also create a wire [3:0] pb_safe and assign its bits to the outputs of four Synchroniser modules whose inputs are Sample, Btns[2], Btns[1] and Btns[0].

1.12.2 Step 2

Create a module DetectFallingEdge (in AuxMod.v). As inputs, it should have a clock and a btn_sync. Here, btn_sync must already have been synchronised to the clock, and should also be debounced.

Question

- Why is a flip-flop required?

  This is because we want to remember the previous value of x.

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

  There would be more delay if we add more flip-flops.

- Why not use negedge?

  If we use negedge, we cannot set it back to 0 after it detects the falling edge.

1.12.3 Step 3

Connect all four synchronised push buttons to instantiations of the DetectFallingEdge module in the CPU module. After the “if (go) begin … end” in the Instruction Cycle block,

add the following code, and fix any compile errors.

```
0: data = set('FLAG, 128);
1: data = mov('FLAG, 'GOUT);
2: data = mov('DINP, 'DOUT);
3: data = jmp(1);
default: data = 35'b0;
```

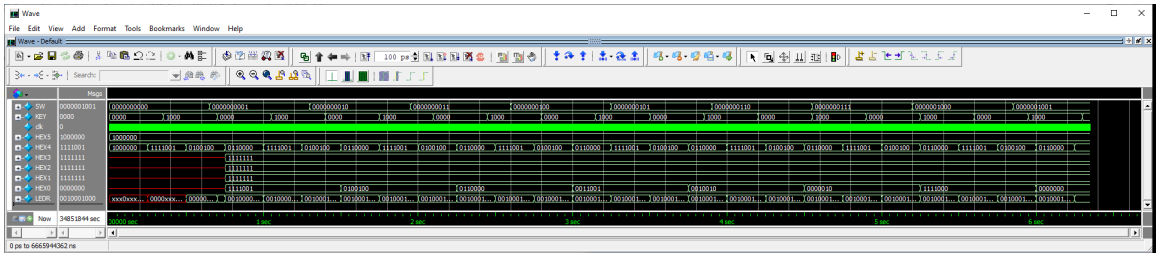

Figure 1.21: Stage 12 in RTL simulation.

Question

- Why might we want the CPU to automatically clear the Flag Register after a Reset?

  This is because we do not want the wrong message before reset influence our operation after reset.

- What does the above Verilog code do?

  To detect a falling edge of pressing a button and clear Reg['FLAG] after reset.

- What does the test program do? Why is 128 loaded into the Flag Register?

  0: set the Reg['FLAG] as 8'b1000_0000;

  1: move the contents in Reg['FLAG] to Reg['GOUT];

  2: move the contents in Reg['DINP] to Reg['DOUT];

  3: jump to IP = 1 unconditianally.

  This is because we want Dval to 1.

- 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?

  Move the value to Register 28 ('RDINP). The value will be stored in Register 28 ('RDINP). It depends on next sample or move instruction. Move instruction takes precedence because hardware changes would cost more time due to the synchroniser.

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

  No, due to the time delay, it is not possible.

- 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?

  It is a more user-friedly way to detect the push button releases rather than push button presses. Suppose you have encountered a situation that you push the buttons by accident, you can record the digits before you release the button.

- When you first started, did you think you would ever finish Part 1?

  Yes.

2 Part 2

- Implement the 'reset module' using 'set' in ROM, including: clear stacks elements and size, register GOUT and Dout. Then add jump to wait at the last.
- Implement the 'wait module' with jumping to other modules using 'atc' function, including: 'push', 'pop', 'add' and 'mult'. Then add jump to wait at the last.
- Implement 'push module': move the stack one by one upwards; move DINP to stack[0]; display stack[0] on 7-segment display; jump to "overflow" if Register[4] equals 8 using 'jmp_eq'; turn off the overflow LEDRs. Then implement jump to 'overflow module' if Register[4] equals 8 using 'jmp_eq' and show the stack size by move the last 4 bit of Register to 'Gout register(this can be done as a 'to jump module' used several times after). Last update the new stack size. Then add jump to wait at the last.
- Implement 'overflow module' by turning Stack Overflow LED on and turn Arithmetic Overflow LED off. Then add jump to wait at the last.
- Implement 'pop module' including: turn off overflow LEDRs; move the stack one by one downwards; update the new stack size; update the new number to dispaly. Then add jump to wait at the last.
- Implement 'add module': turn off Arithmetic Overflow LED and jump to "wait" if Register [4] equals 0 or 1; take the signed addition and update the stack; turn on Arithmetic Overflow LED if an overflow occurred; move the stack one by one downwards. Then add jump to wait at the last.
- Implement 'mult module': turn off Arithmetic Overflow LED and jump to "normal module", otherwise jump to 'wait module'; take the multiplication and update the stack; turn on Arithmetic Overflow LED if an overflow occurred; move the stack one by one downwards. Then add jump to wait at the last.

There are some tricks:
1) When implementing the instructions, we can write them discontinuously so that we can implement more instruction without changing instruction address in other modules.
2) We can use 'jmp' or 'atc' as 'if...else' function.
3) If there are some similar steps like 'if ..., then jump to wait', we can implement that once at the end and use jump to call this 'function'.

4) If there is no number in the stack, we cannot display '0'. Then first we should initialise the 'Gout register with all 'o'; next set the 'Dval bit of 'Gout '1' after pushing. The second one is in 'pop module', we shold also turn off the display by set the 'Dval bit of 'Gout '0' and if stack size is '0', jump to wait(avoid set the 'Dval bit of 'Gout '1'); else set the 'Dval bit of 'Gout '1'.

5) Then the most struggling question for me: how to display the stack size? My solution is to set the other bits of Register [4] same as 'Gout register and move the Register [4] to 'Gout avoiding changing other bits of 'Gout. Next reset the Register [4] as before the setting. this is meant to only change the bits representing stack size in 'Gout.

The figure 2.1 shows the final computer test in RTL simulation.

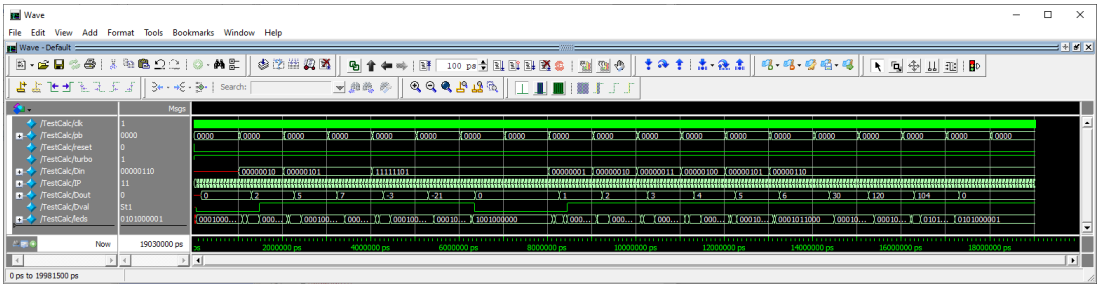

Figure 2.1: RPN computer test in RTL simulation.

From the above figure, we can find the computer works correctly as expected.