**Ali Sencer Irmak – CSE224.1**

**20200702096**

**Very Simple Cpu Verilog Design**

I have implemented 2 modules in my design. One of them is VerySimpleCPU and the other one is blram. In the VerySimpleCPU module I have 7 ports. These are clk, rst, interrupt and data\_fromRAM (32-bit) as inputs. Also, wrEn, data\_toRAM (32-bit), addr\_toRAM (14-bit) as outputs. Moreover, this design is consisting of st (3-bit), stN (3-bit), PC (3-bit), PCN (3-bit), IW (32-bit), IWN (32-bit), A (32-bit), AN (32-bit), intr, intrN, isr, isrN as reg type variables. In every posedge signal, each variable will be equal to their next value.  
This is 8-state designed code but we just use 6 of them. Also, we have synchronous reset. In the zero state, we fetch instruction from memory to IW. In the first state, we read opcode specified on each instruction. In the second state, if it is enough to read one opcode for instruction, program do operations on each specified instruction, enable writing mode, increment PC value, set next state as zero state and finish the instruction. However, if it is not enough to read one opcode, program goes next state which is fourth state and program do operations on each specified instruction, enable writing mode, increment PC value, set next state as zero state and finish the instruction. In addition, this design can handle interruption. If any interruption is encountered, program complete the execution of the current instruction, get the ISR address which is stored at address #5 in memory. Next, store the address of the next instruction at address #6 in memory. Run ISR, return from ISR and continue from where left off. In the fifth and sixth state for interruption.

In the blram module which is memory I have 5 ports. These are clk, i\_we, i\_addr (14-bit), as inputs i\_ram\_data\_in (32-bit), o\_ram\_data\_out (32-bit) as outputs. Also, reg type variable which is memory (64KByte).  
In this module, if ram is in reading mode, data which is specified in VerySimpleCPU is loaded to o\_ram\_data\_out from memory in every posedge signal. If ram is in writing mode, data which is specified in VerySimpleCPU is loaded to memory from i\_ram\_data\_in in every posedge signal.

This very simple cpu that i designed can handle ADD, ADDi, NAND, NANDi, SRL, SRLi, LT, LTi, MUL, MULi, CP, CPi, CPI, CPIi, BZJ, BZJi.

**Synthesis Area and Timing Report**

Minimum clock frequency is 7.158ns  
Maximum clock frequency is 139.709MHz

|  |  |  |  |
| --- | --- | --- | --- |
| **Logic Utilization** | **Used** | **Available** | **Utilization** |
| Number of Slices | 705 | 960 | 73% |
| Number of Slice Flip Flops | 92 | 1920 | 4% |
| Number of 4 input LUTs | 1289 | 1920 | 67% |
| Number of bonded IOBs | 82 | 83 | 98% |
| Number of MULT18X18SIOs | 2 | 4 | 50% |
| Number of GCLKs | 1 | 24 | 4% |

**Full synthesis report is available in the attachment.**

**EDA PLAYGROUND LINK:** [**https://edaplayground.com/x/fWaJ**](https://edaplayground.com/x/fWaJ)

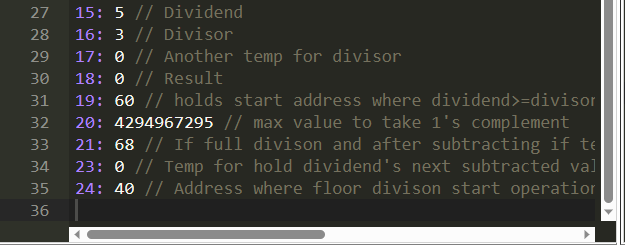
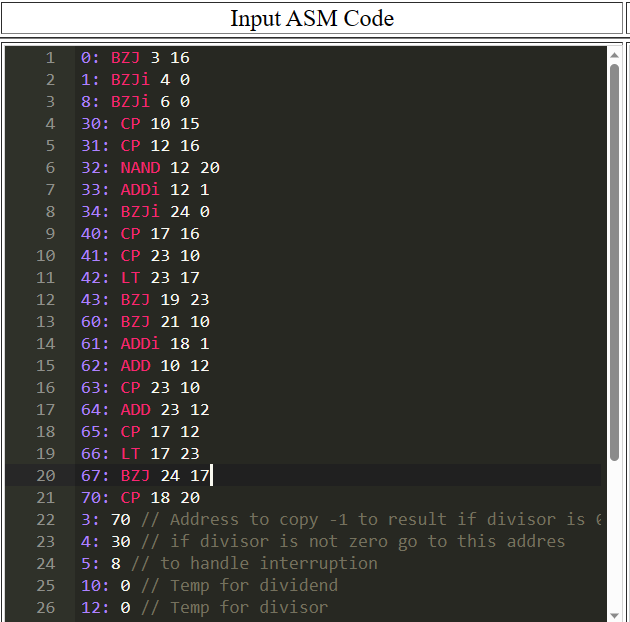
**FLOOR DIVISION ALGORITHM**

**#15 holds dividend, #16 holds divisor and #18 holds result.**

**Address #5, Address #6 and Address #8 for interrupt handling**

My floor division code start with checking whether divisor is 0 or not. If it is zero, set result as -1 in signed decimal representation (FFFF FFFF in hex). Also, program checks divisor > dividend between #40 and #43, if it is not, it checks for dividend >= divisor between #60 and #67. Division operation is done by subtracting divisor from dividend consecutively.

If divisor is not zero, it jumps to address #30 and get prepared for division. Firstly, program copy original dividend value at address #15 to address #10 and copy original divisor value at address #16 to address #12 **not to lose original data**. After copying values, program takes 2’s complement of temp divisor value at address #12 and jump to address #40.  
At address #40, program copy original divisor value to address #17 and temp dividend value to address #23. After copying values, program checks whether divisor is greater than temp dividend value (for the first time check original dividend, next time check for subtracted dividend value). If it is greater than dividend, program is stopped and keep holding result as 0 at address #18.   
If it is less than, program jumps to address #60 to check whether temp dividend value is equal to 0. It checks this equality to decide whether division is full or not. If it is equal, program is stopped and protect result at address #18.   
If it is not equal, program keeps doing next instructions. That’s mean, dividend>=divisor.   
Respectively, add +1 to address #18.   
Subtract divisor from dividend.   
Copy subtracted value (temp dividend) to address #23.   
Subtract divisor from new dividend.   
Copy negative divisor value to #17. Compare whether next subtracted (1 more time subtracted) value at #23, is greater than negative divisor. **(For example, 4 // 3 = 1. 4-3-3 = -2 at address #23) (For example, 10 // 3 = 3. 10-3-3 = 4 at address #23)  
(You have to think signed and unsigned decimal values as HEXADECIMAL)**If it is, program set value at #17 to 1 and finish.  
If it is not, program set value at #17 to 0 and go to address #40 to do the same operations with new subtracted dividend (just 1 time subtracted) and divisor.

**  
It is also available as .txt format in the attachment  
You can see example next page ( 7 // 3 = 2)**

(Example: 7 // 3 = 2)

First checks whether divisor is 0 or not. It is 3 so it jumps to address #30 and copy original divisor and dividend values. Also, create negative divisor. After doing that it jumps address #40 to check for 3>7 or not.

It is not therefore, it jumps address #60 and checks whether divisor = dividend or not. It is not so program subtract 3 from 7 and increment value at #18 by 1. After that, it subtracts divisor value from new dividend value which is 4 (From first subtract 7 – 3 = 4) and checks if 1 (4 – 3 = 1) is greater than -3 (negative divisor) or not. (Important you should be aware this comparison is done in hexadecimal) (1 in hex is 1, -3 in hex is FFFF FFFD). It is not greater so program use divisor and new dividend value which is 4 and jumps to address #40 to do the same operations that I mentioned above. After doing operation between addresses #40 and #43

it jumps address #60 and checks whether divisor = dividend or not. It is not, so program subtract 3 from 4 and increment value at #18 by 1. After that, it subtracts divisor value from new dividend value which is 1 (From first subtract 4 – 3 = 1) and checks if -2 (1 – 3 = -2) is greater than -3 (negative divisor) or not. (Important you should be aware this comparison is done in hexadecimal) (-2 in hex is FFFF FFFE, -3 in hex is FFFF FFFD). FFFF FFFE is greater than FFFF FFFD so program finishes with result as 2 at address #18