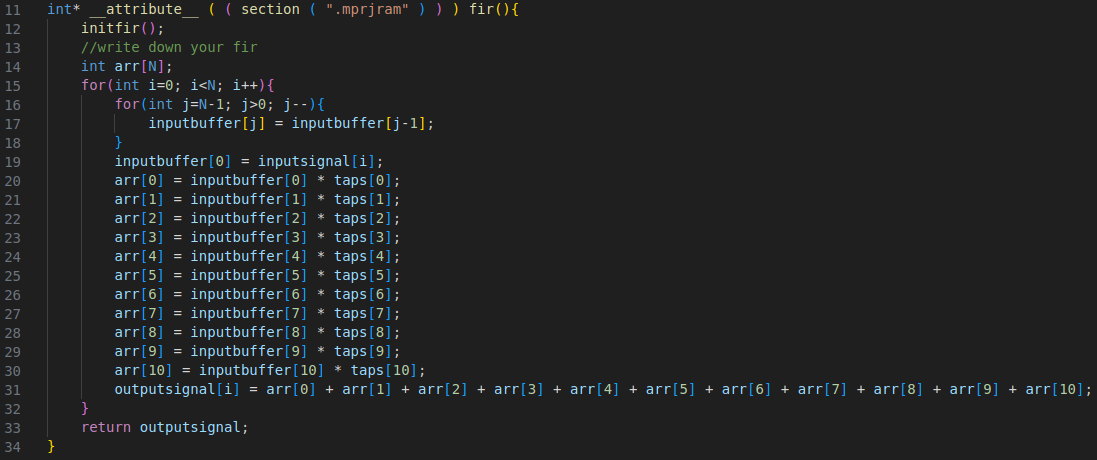
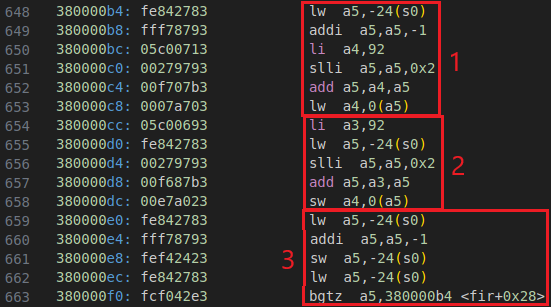
Lab 4-1 Report

王品棠(310511082) 李秉翰(311510060) 曾子鈞(311512030)

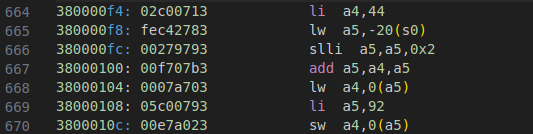
**Explanation of your firmware code**

**How does it execute a multiplication in assembly code**

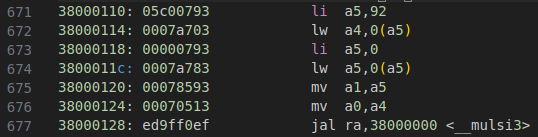
* I will first explain the multiplication from the c code (fir.c) aspect, then explain the assembly code corresponding to the c code.
* The function “initfir” initializes the values in the “inputbuffer” and “outputsignal” array.
* Line 15~32: The outer for-loop will execute N times, calculating each output value corresponding to each input signal.
* Line 16~18: The inner for-loop will execute N-1 times, shifting the value in the input buffer.
* Line 19: Then, the current input value will be stored in the first position in the “inputbuffer”.
* Line 20~30: X[n], X[n-1], …, X[n-N+1] is multiplied by the coefficient and stored in the “arr” array, respectively.
* Line 31: add the multiplication results in the “arr” array together and store in corresponding position in “outputsignal” array.
* Line 33: return “outputsignal” array.
* The following portion of assembly code is corresponding to the line 16~18 inner for-loop.
* Block 1: the index j is stored at -24(s0), so first load the j value, then subtract by 1 for the [j-1] position index and left shift by 2, because datatype int has 4 bytes. Then add the beginning position (92, stored in a4) to calculate the actual address, then load to reg: a4.
* Block 2: the following assembly code is similar as above but storing the value of “inputbuffer[j-1]” (in reg: a4) to “inputbuffer[j]”.
* Block 3: subtract the index j by 1 (j--), and check the loop is over or not (equal to zero or not).



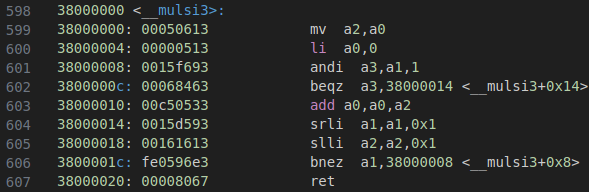
* The following portion of assembly code is corresponding to the line 19.
* Calculate the [i-1] position for “inputbuffer[i-1]” and load to reg: a4, then store the value to “inputbuffer[0]”.



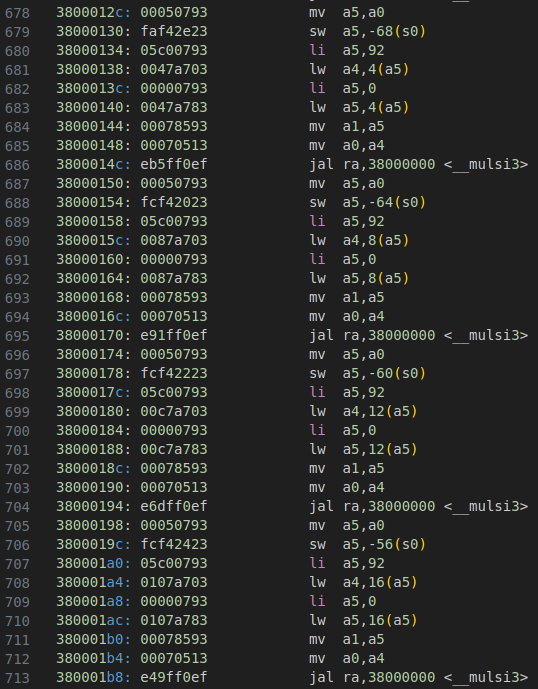
* The following portion of assembly code is corresponding to the line 20.
* Load the “inputbuffer[0]” and the “taps[0]” and stored the values to reg: a0 and reg: a1, then jump to \_\_mulsi3.



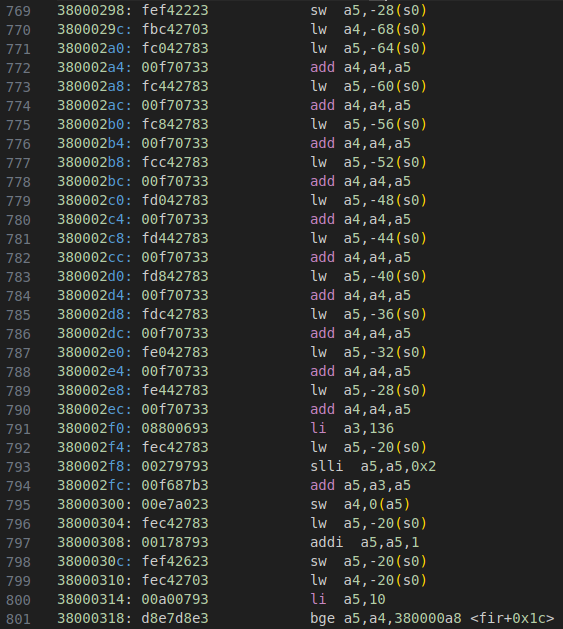
* Copy the “inputbuffer[0]” value to reg: a2, then use reg: a0 as an accumulator later. Use “andi” to chech the lower bit of reg: a1 (taps[0]) is 1 or not, add reg: a2 (inputbuffer[0]) to the accumulator: a0. Then, right shift reg: a1 and left shift reg: a2, so that the “inputbuffer[0]” value will be correctly accumulated corresponding to the lowest bit of “taps[0]” in the next round. End the accumulation if reg: a1 is zero, otherwise go to the next round of accumulation.



* The following portion of assembly code is corresponding to the line 21~24.
* The following code is similar as above. Executing the c code from “arr[1] = inputbuffer[1] \* taps[1]” to “arr[4] = inputbuffer[4] \* taps[4]”.
* Line 20~30 in c code is also executed similarly as above.

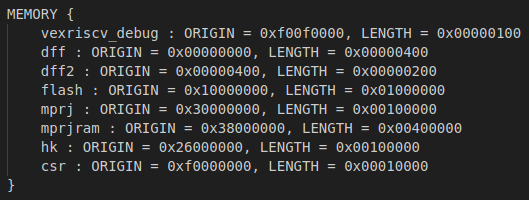


* The following portion of assembly code is corresponding to the line 31.
* Load and accumulate the values of arr[0] to arr[10], then store the accumulated result in the corresponding position of “outputsignal” array. Then, check for-loop index i to decide jump or not.

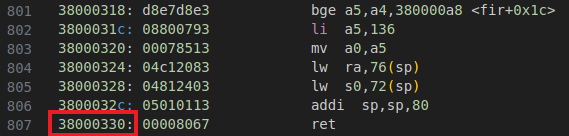


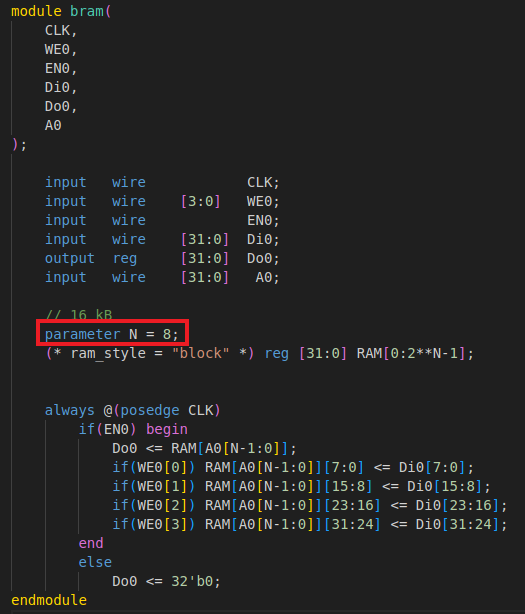
**What address allocate for user project and how many space is required to allocate to firmware code**

* User project is allocated in the section of “mprj” with the address of 0x3000000.

****

* The user specified section “mprjram” is allocated to place the fir firmware code. However, the space is limited by the ram size specified in “bram.v”.
* According to “counter\_la\_fir.out”, the last instruction in “fir.c” is placed at 0x38000330, so the length of firmware code is 0x334 bytes (820 bytes). Since the BRAM has 32-bit data width, so the RAM size should be larger than 205 (820/4). Therefore, the minimum N should be set to 8, providing size of 1KB (2^8 \* 4 bytes).

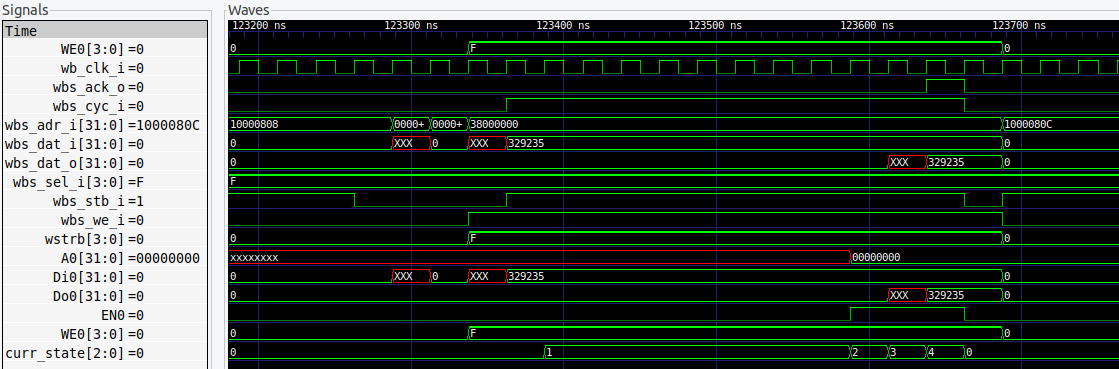
****

****

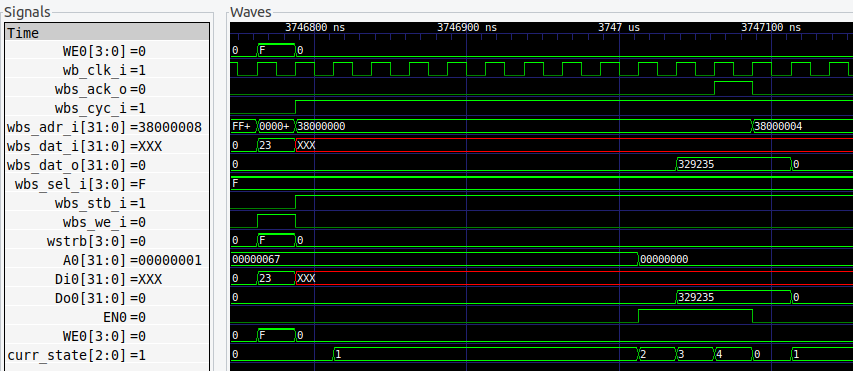
**Interface between BRAM and wishbone**

**Waveform from xsim**

* The following waveform shows the interface between BRAM and wishbone during writing data to BRAM from wishbone interface.
* The wbs\_ack\_o is raised with 10 cycles delay after the wbs\_cyc\_i rising edge.

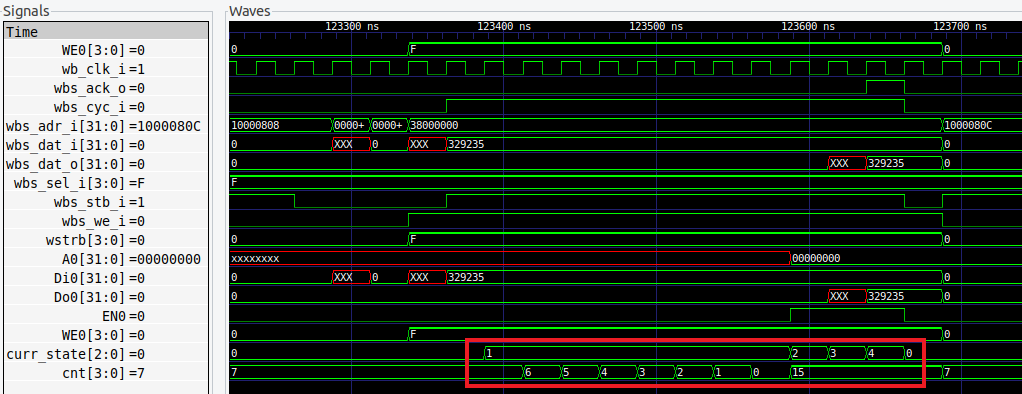


* The following waveform shows the interface between BRAM and wishbone during reading data from BRAM to wishbone interface.
* The wbs\_ack\_o is raised with 10 cycles delay after the wbs\_cyc\_i rising edge.



**FSM**

* The following waveform shows the interface between BRAM and wishbone during writing data to BRAM from wishbone interface.
* In the state “1”, the FSM uses the “cnt” to count for the delay cycles. In the state “2” ~ state “4”, FSM write the data in the BRAM then output wbs\_ack\_o back to wishbone interface.



* The following waveform shows the interface between BRAM and wishbone during reading data from BRAM to wishbone interface.
* In the state “1”, the FSM uses the “cnt” to count for the delay cycles. In the state “2” ~ state “4”, FSM write the data in the BRAM then output wbs\_ack\_o back to wishbone interface.

