Lab 4-1 Report

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

Explanation of your firmware code

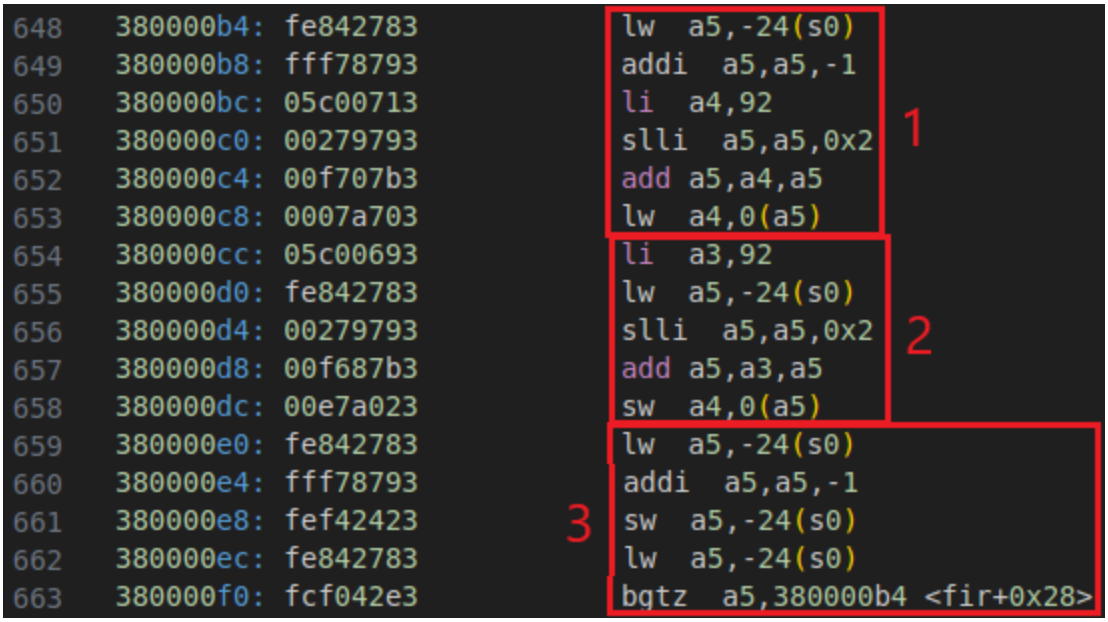
How does it execute a multiplication in assembly code

- 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.

```
11  int* __attribute__ ( ( section ( ".mprjram" ) ) ) fir(){
12      initfir();
13      //write down your fir
14      int arr[N];
15      for(int i=0; i<N; i++){
16          for(int j=N-1; j>0; j--){
17              inputbuffer[j] = inputbuffer[j-1];
18          }
19          inputbuffer[0] = inputsignal[i];
20          arr[0] = inputbuffer[0] * taps[0];
21          arr[1] = inputbuffer[1] * taps[1];
22          arr[2] = inputbuffer[2] * taps[2];
23          arr[3] = inputbuffer[3] * taps[3];
24          arr[4] = inputbuffer[4] * taps[4];
25          arr[5] = inputbuffer[5] * taps[5];
26          arr[6] = inputbuffer[6] * taps[6];
27          arr[7] = inputbuffer[7] * taps[7];
28          arr[8] = inputbuffer[8] * taps[8];
29          arr[9] = inputbuffer[9] * taps[9];
30          arr[10] = inputbuffer[10] * taps[10];
31          outputsignal[i] = arr[0] + arr[1] + arr[2] + arr[3] + arr[4] + arr[5] + arr[6] + arr[7] + arr[8] + arr[9] + arr[10];
32      }
33      return outputsignal;
34  }
```

- 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).




```
648   380000b4: fe842783        lw  a5,-24(s0)
649   380000b8: fff78793        addi  a5,a5,-1
650   380000bc: 05c00713        li  a4,92
651   380000c0: 00279793        slli  a5,a5,0x2
652   380000c4: 00f707b3        add a5,a4,a5
653   380000c8: 0007a703        lw  a4,0(a5)
654   380000cc: 05c00693        li  a3,92
655   380000d0: fe842783        lw  a5,-24(s0)
656   380000d4: 00279793        slli  a5,a5,0x2
657   380000d8: 00f687b3        add a5,a3,a5
658   380000dc: 00e7a023        sw  a4,0(a5)
659   380000e0: fe842783        lw  a5,-24(s0)
660   380000e4: fff78793        addi  a5,a5,-1
661   380000e8: fef42423        sw  a5,-24(s0)
662   380000ec: fe842783        lw  a5,-24(s0)
663   380000f0: fcf042e3        bgtz  a5,380000b4 <fir+0x28>
```

- 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]”.

```
664   380000f4: 02c00713        li  a4,44
665   380000f8: fec42783        lw  a5,-20(s0)
666   380000fc: 00279793        slli  a5,a5,0x2
667   38000100: 00f707b3        add a5,a4,a5
668   38000104: 0007a703        lw  a4,0(a5)
669   38000108: 05c00793        li  a5,92
670   3800010c: 00e7a023        sw  a4,0(a5)
```

- 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.

```
671   38000110: 05c00793        li  a5,92
672   38000114: 0007a703        lw  a4,0(a5)
673   38000118: 00000793        li  a5,0
674   3800011c: 0007a783        lw  a5,0(a5)
675   38000120: 00078593        mv  a1,a5
676   38000124: 00070513        mv  a0,a4
677   38000128: ed9ff0ef        jal ra,38000000 <__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.

```
598   38000000 <__mulsi3>:
599   38000000: 00050613            mv  a2,a0
600   38000004: 00000513            li  a0,0
601   38000008: 0015f693            andi  a3,a1,1
602   3800000c: 00068463            beqz  a3,38000014 <__mulsi3+0x14>
603   38000010: 00c50533            add a0,a0,a2
604   38000014: 0015d593            srli  a1,a1,0x1
605   38000018: 00161613            slli  a2,a2,0x1
606   3800001c: fe0596e3            bnez  a1,38000008 <__mulsi3+0x8>
607   38000020: 00008067            ret
```

- 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.

```
678   3800012c: 00050793            mv  a5,a0
679   38000130: faf42e23            sw  a5,-68(s0)
680   38000134: 05c00793            li  a5,92
681   38000138: 0047a703            lw  a4,4(a5)
682   3800013c: 00000793            li  a5,0
683   38000140: 0047a783            lw  a5,4(a5)
684   38000144: 00078593            mv  a1,a5
685   38000148: 00070513            mv  a0,a4
686   3800014c: eb5ff0ef            jal ra,38000000 <__mulsi3>
687   38000150: 00050793            mv  a5,a0
688   38000154: fcf42023            sw  a5,-64(s0)
689   38000158: 05c00793            li  a5,92
690   3800015c: 0087a703            lw  a4,8(a5)
691   38000160: 00000793            li  a5,0
692   38000164: 0087a783            lw  a5,8(a5)
693   38000168: 00078593            mv  a1,a5
694   3800016c: 00070513            mv  a0,a4
695   38000170: e91ff0ef            jal ra,38000000 <__mulsi3>
696   38000174: 00050793            mv  a5,a0
697   38000178: fcf42223            sw  a5,-60(s0)
698   3800017c: 05c00793            li  a5,92
699   38000180: 00c7a703            lw  a4,12(a5)
700   38000184: 00000793            li  a5,0
701   38000188: 00c7a783            lw  a5,12(a5)
702   3800018c: 00078593            mv  a1,a5
703   38000190: 00070513            mv  a0,a4
704   38000194: e6dff0ef            jal ra,38000000 <__mulsi3>
705   38000198: 00050793            mv  a5,a0
706   3800019c: fcf42423            sw  a5,-56(s0)
707   380001a0: 05c00793            li  a5,92
708   380001a4: 0107a703            lw  a4,16(a5)
709   380001a8: 00000793            li  a5,0
710   380001ac: 0107a783            lw  a5,16(a5)
711   380001b0: 00078593            mv  a1,a5
712   380001b4: 00070513            mv  a0,a4
713   380001b8: e49ff0ef            jal ra,38000000 <__mulsi3>
```

- 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.

```
38000298: fef42223           sw  a5,-28(s0)
3800029c: fbc42703           lw  a4,-68(s0)
380002a0: fc042783           lw  a5,-64(s0)
380002a4: 00f70733           add a4,a4,a5
380002a8: fc442783           lw  a5,-60(s0)
380002ac: 00f70733           add a4,a4,a5
380002b0: fc842783           lw  a5,-56(s0)
380002b4: 00f70733           add a4,a4,a5
380002b8: fcc42783           lw  a5,-52(s0)
380002bc: 00f70733           add a4,a4,a5
380002c0: fd042783           lw  a5,-48(s0)
380002c4: 00f70733           add a4,a4,a5
380002c8: fd442783           lw  a5,-44(s0)
380002cc: 00f70733           add a4,a4,a5
380002d0: fd842783           lw  a5,-40(s0)
380002d4: 00f70733           add a4,a4,a5
380002d8: fdc42783           lw  a5,-36(s0)
380002dc: 00f70733           add a4,a4,a5
380002e0: fe042783           lw  a5,-32(s0)
380002e4: 00f70733           add a4,a4,a5
380002e8: fe442783           lw  a5,-28(s0)
380002ec: 00f70733           add a4,a4,a5
380002f0: 08800693           li  a3,136
380002f4: fec42783           lw  a5,-20(s0)
380002f8: 00279793           slli  a5,a5,0x2
380002fc: 00f687b3           add a5,a3,a5
38000300: 00e7a023           sw  a4,0(a5)
38000304: fec42783           lw  a5,-20(s0)
38000308: 00178793           addi  a5,a5,1
3800030c: fef42623           sw  a5,-20(s0)
38000310: fec42703           lw  a4,-20(s0)
38000314: 00a00793           li  a5,10
38000318: d8e7d8e3           bge a5,a4,380000a8 <fir+0x1c>
```

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.

```
MEMORY {
    vexriscv_debug : ORIGIN = 0xf00f0000, LENGTH = 0x00000100
    dff : ORIGIN = 0x00000000, LENGTH = 0x00000400
    dff2 : ORIGIN = 0x00000400, LENGTH = 0x00000200
    flash : ORIGIN = 0x10000000, LENGTH = 0x01000000
    mprj : ORIGIN = 0x30000000, LENGTH = 0x00100000
    mprjram : ORIGIN = 0x38000000, LENGTH = 0x00400000
    hk : ORIGIN = 0x26000000, LENGTH = 0x00100000
    csr : ORIGIN = 0xf0000000, LENGTH = 0x00010000
}
```

- 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).

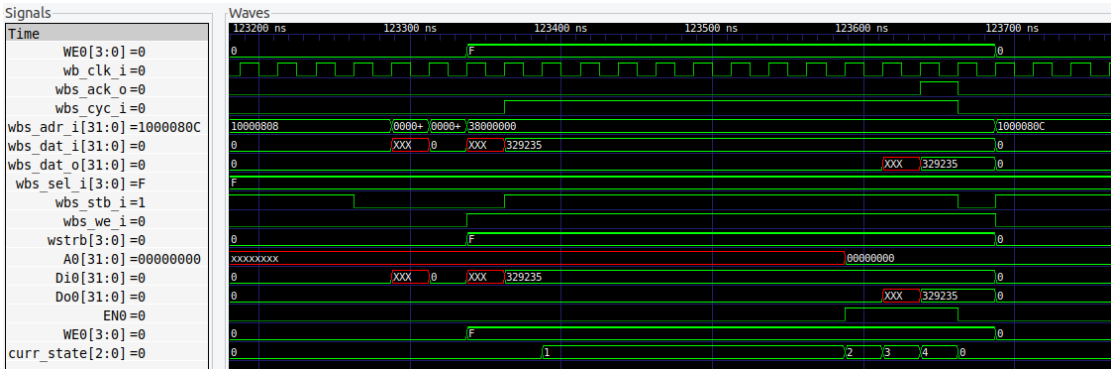
```
801    38000318: d8e7d8e3           bge a5,a4,380000a8 <fir+0x1c>
802    3800031c: 08800793           li  a5,136
803    38000320: 00078513           mv  a0,a5
804    38000324: 04c12083           lw  ra,76(sp)
805    38000328: 04812403           lw  s0,72(sp)
806    3800032c: 05010113           addi  sp,sp,80
807    38000330: 00008067           ret
```

```
module bram(
    CLK,
    WE0,
    EN0,
    Di0,
    Do0,
    A0
);

    input   wire            CLK;
    input   wire    [3:0]   WE0;
    input   wire            EN0;
    input   wire    [31:0]  Di0;
    output  reg     [31:0]  Do0;
    input   wire    [31:0]   A0;

    // 16 kB
    parameter N = 8;
    (* ram_style = "block" *) reg [31:0] RAM[0:2**N-1];


    always @(posedge CLK)
        if(EN0) begin
            Do0 <= RAM[A0[N-1:0]];
            if(WE0[0]) RAM[A0[N-1:0]][7:0] <= Di0[7:0];
            if(WE0[1]) RAM[A0[N-1:0]][15:8] <= Di0[15:8];
            if(WE0[2]) RAM[A0[N-1:0]][23:16] <= Di0[23:16];
            if(WE0[3]) RAM[A0[N-1:0]][31:24] <= Di0[31:24];
        end
        else
            Do0 <= 32'b0;
endmodule
```

Interface between BRAM and wishbone

Waveform from xsim

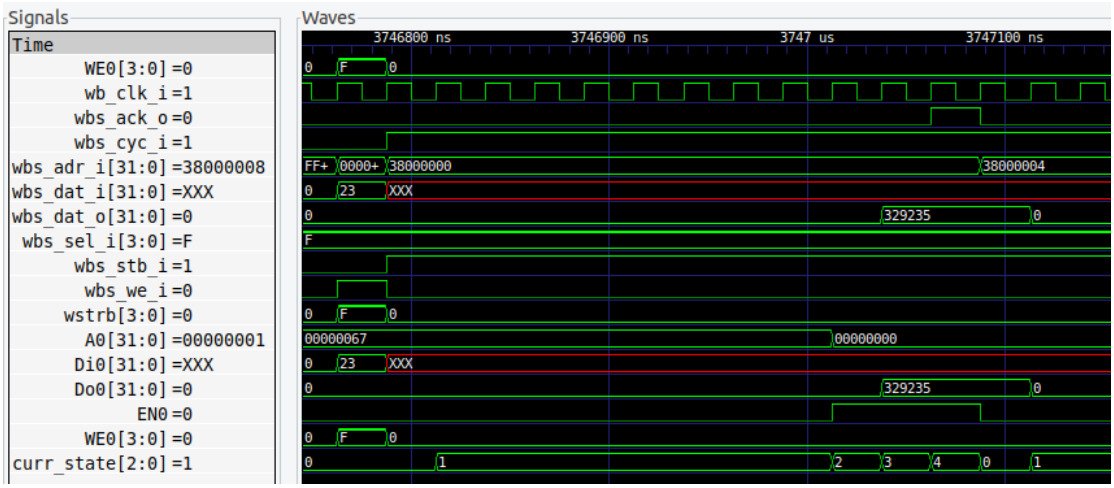
- The following waveform shows the interface between BRAM and wishbone during writing data to BRAM from wishbone interface.
- The wbs_ack_o is raised with 10 cycles delay after the wbs_cyc_i rising edge.



- The following waveform shows the interface between BRAM and

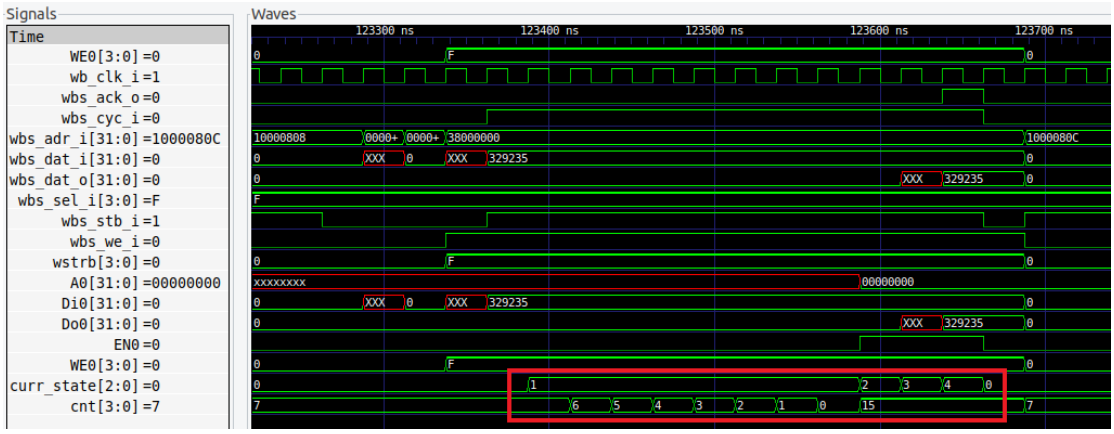
wishbone during reading data from BRAM to wishbone interface.

- The wbs_ack_o is raised with 10 cycles delay after the wbs_cyc_i rising edge.

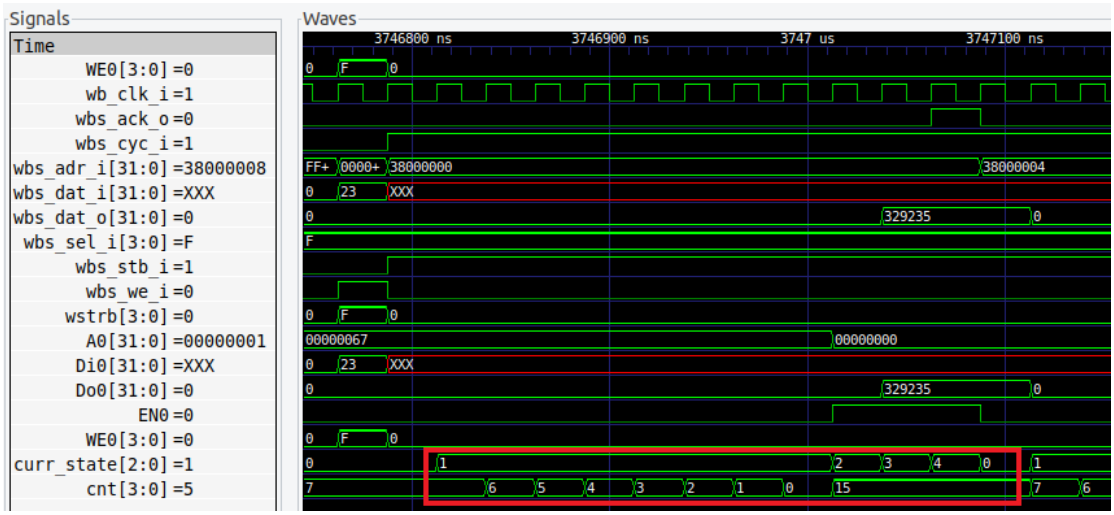


FSM

- 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.


Signals
Waves
Time
WE0[3:0] =0
wb_clk_i =1
wbs_ack_o =0
wbs_cyc_i =1
wbs_adr_i[31:0] =38000008
wbs_dat_i[31:0] =XXX
wbs_dat_o[31:0] =0
wbs_sel_i[3:0] =F
wbs_stb_i =1
wbs_we_i =0
wstrb[3:0] =0
A0[31:0] =00000001
Di0[31:0] =XXX
Do0[31:0] =0
EN0 =0
WE0[3:0] =0
curr_state[2:0] =1
cnt[3:0] =5
3746800 ns
3746900 ns
3747 us
3747100 ns
FF+
0000+
38000000
38000004
329235
00000067
00000000
XXX