# Lab 4 Report
# SHEN Zijun

## 1. Explanation of firmware code

### 1-1. How does it execute a multiplication in assembly code?

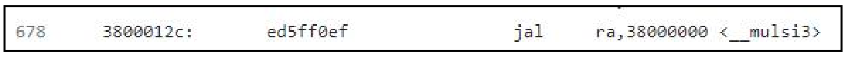
We write the FIR operation in fir.c:

*outputsignal[i] ←outputsignal[i] + taps[j]*inputbuffer[data_RAM_pointer];*

Multiplication is used in this process, so after the compile becomes assembly code, there will be a section (section) responsible for the multiplication operation. The following describes how to execute this section and the multiplication operation at this stage in the assembly code:

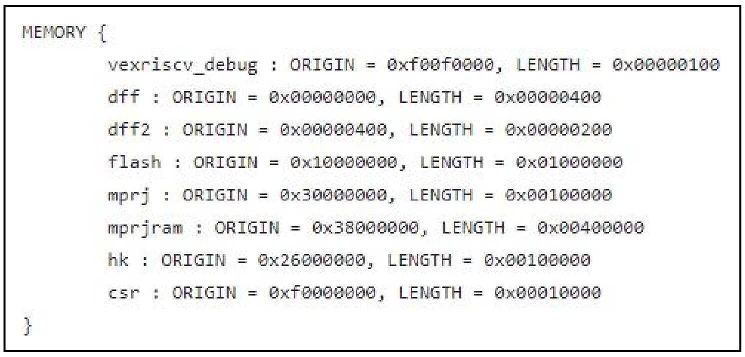
We found that the file counter_la_fir.out, obtained after compiling fir.c, clearly lists each assembly code and its corresponding segment (such as the <mulsi3> segment in Fig. 3), as well as the addresses where these codes are stored in the machine language after conversion (for example, the instruction "mv a2, a0" in Fig. 3 is stored at address 32 'h38000000). These codes in fir.c are stored at addresses corresponding to the user project BRAM (near 32' h38000000) because the function declarations in fir.c specifically include "section ('.mprjram')", and this is defined in the section.lds file (Fig. 2). mprjram "This section occupy is a memory configuration address that starts from 0x38000000 and lasts for 0x00400000, so compiler, you know, you have to store this part of the instruction code in this segment, which is stored in the user project BRAM.

When the CPU reaches the <fir> section (corresponding to fir.c code) and is about to perform the multiplication, it jumps (Using jal (jump and link) instruction, as shown in Figure 1) to perform multiplication on the <mulsi3> section.



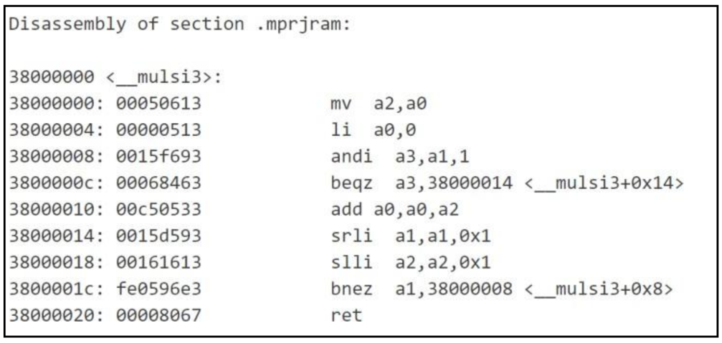
```
678     3800012c:     ed5ff0ef          jal     ra,38000000 <__mulsi3>
```

Fig. 1



```
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
}
```

Fig. 2



```
Disassembly of section .mprjram:

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

Fig. 3

Figure 3 shows that the section 1 about multiplication operation is found in the counter_la_fir.out file program counter (PC) jumps to the position of 32'h38000000 in address (that is <mulsi3> The beginning address of the section is followed by a multiplication operation, which is performed as follows: First, since jal can be regarded as a call to the multiplication function, the multiplicand and multiplier's arguments must first be placed in the register of a0 and a1, respectively, before entering this section for computation. The final return value also needs to be placed in the register of a0 so that it can be read by the upper-level function (<fir>):

| Steps | Address | instruction | Explanation |
|---|---|---|---|
| (1) | 0x38000000 | mv a2,a0 | Copy the value of a0 to a2 |
| (2) | 0x38000004 | li a0,0 | Read "0" into a0 |
| (3) | 0x38000008 | andi a3,a1,1 | Write the value of a3 as "a1 & 1" |
| (4) | 0x3800000c | beqz a3,38000014 <__mulsi3+0x14> | If the value of a3 = 0, then jump to 0x38000014 |
| (5) | 0x38000010 | add  a0,a0,a2 | a0 $\leftarrow$ a0+a2 |
| (6) | 0x38000014 | srli  a1,a1,0x1 | a1 $\leftarrow$ a1>>1 |
| (7) | 0x38000018 | slli  a2,a2,0x1 | a2 $\leftarrow$ a2<<1 |
| (8) | 0x3800001c | bneza1,38000008 <__mulsi3+0x8> | If the value of a1 = 0, then jump to 0x38000008 |
| (9) | 0x38000020 | ret | return to Upper layer function(<fir>) |

Table 1

Due to the presence of branch-related instruction, it is not necessary to follow the order in the table. After derivation, we can roughly understand that its method of performing multiplication is as follows: "The upper function <fir> places two arguments (the multiplicand and multiplier) into a0 and a1. However, since a0 ultimately needs to store return value, the multiplicand (a0) is first copied to a2 register, with a0 serving as the register for storing the product, while a1 retains the multiplier." Then proceed with the multiplication operation: check if the LSB of the multiplier is 1. If so, add the current multiplicand to the product a0; otherwise, do not add it to a0. Next, perform the operation for the next bit, but first shift the multiplier right by shift 1 bit (indicating that the value of the next bit is to be checked) and shift the multiplicand left by shift 1 bit (representing that this bit's weighting has been incremented by one order). Then, check the next LSB. Finally, when the entire multiplier is shift ed down to 0, it indicates that the operation is complete, and you can return back up one level in the function (the multiplier is now exactly in return value register (a0)). Here's an example to illustrate: when the multiplicand When a0=11011 and multiplier a1=101

(1) a2←a0=11011
(2) a0←0
(3) a3←a1[0]=1
(4) a3≠0→No jump
(5) a0← a0+a2=0+11011=11011 (Since the LSB of a1 is 1, add 11011 to the product)
(6) a1←(a1>>1)=10
(7) a2 ← (a2<<1)=110110
(8) a1≠0→jump to (3)
  (3) a3←a1[0]=0
  (4) a3=0→jump to (6) (Since the LSB of a1 is 0, 110110 is not added to the product)
  (6) a1←(a1>>1)=1
  (7) a2 ← (a2<<1)=1101100
  (8) a1≠0→jump to (3)
  (3) a3←a1[0]=1
  (4) a3≠0→No jump
      a0← a0+a2=11011+1101100=10000111 (Since the LSB of a1 is 1, 1101100 will be used Add to the product)
  (5) a1←(a1>>1)=0
  (6) a2 ← (a2<<1)= 11011000
  (7) a1=0→No jump
  (8) return

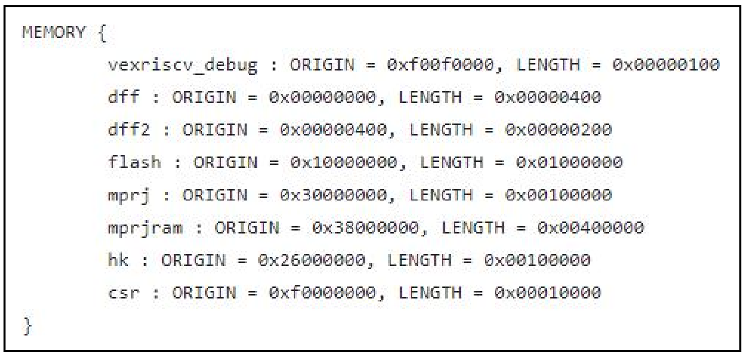
The final product is 10000111. Convert the above formula to decimal for easy observation：

$$(11011)_2 \times (101)_2 = (27)_{10} \times (5)_{10} = (135)_{10} = (10000111)_2$$

The calculation results are correct

**1-2. What address allocate for user project and how many space is required to allocate to firmware code?**

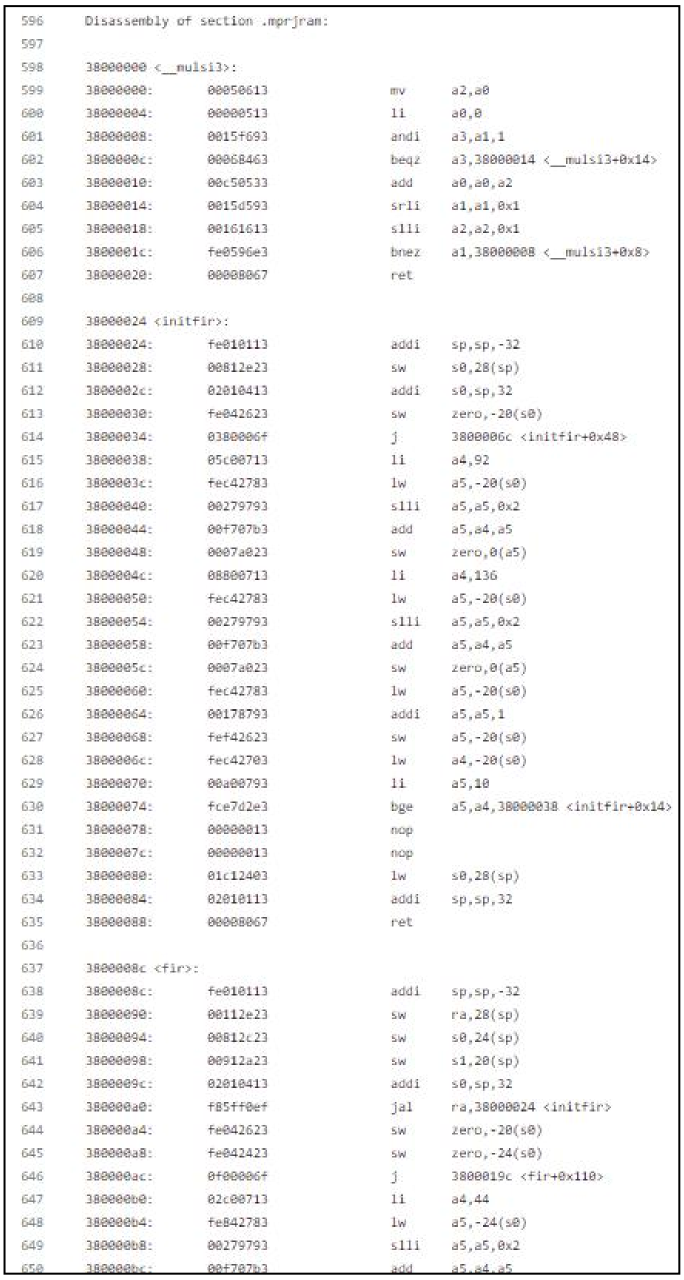
In the section.lds file (see Fig. 4 below), configuration address map is defined to explain each The allocation of a memory address block, for example, as shown in the figure, defines the address starting from 0x38000000 and lasting for 0x00400000. This way, wishbone can know which interval's address to provide to successfully obtain access at a certain location.



```
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
}
```

Fig. 4

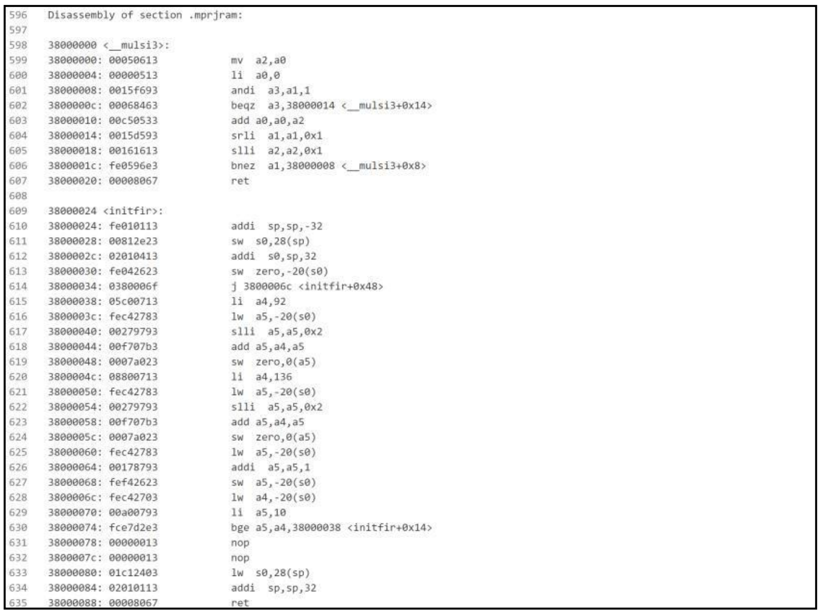
In the description above at 1-1., it is mentioned that in the fir.c file of the user project, all functions we defined have the tag "section (" mprjram ")" added when declaring the function. This indicates that we want to store this code segment as assembly code (and subsequently as machine language, hex file) in the memory block at 0x38000000. As shown in Figure 4, this section is allocated to the address space of the user project as mprjram, which is the memory block from 0x38000000 to 0x383FFFFF. (This can also be seen in Fig. 5 below: this is the counter_la_fir.out file (fir.c has been passed A screenshot from one of the output files after compile) shows a section including the multiplier The <initfir> segment corresponding to the function of <mulsi3>, fir initialization and the <fir> segment corresponding to the fir() function are both located in the 0x3800000000 segment, so it can be seen that the function we wrote will indeed be stored in these address space after being converted into assembly code/machine code.)



```
596    Disassembly of section .mprjram:
597
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
608
609    38000024 <initfir>:
610    38000024:       fe010113                addi    sp,sp,-32
611    38000028:       00812e23                sw      s0,28(sp)
612    3800002c:       02010413                addi    s0,sp,32
613    38000030:       fe042623                sw      zero,-20(s0)
614    38000034:       0380006f                j       3800006c <initfir+0x48>
615    38000038:       05c00713                li      a4,92
616    3800003c:       fec42783                lw      a5,-20(s0)
617    38000040:       00279793                slli    a5,a5,0x2
618    38000044:       00f707b3                add     a5,a4,a5
619    38000048:       0007a023                sw      zero,0(a5)
620    3800004c:       08800713                li      a4,136
621    38000050:       fec42783                lw      a5,-20(s0)
622    38000054:       00279793                slli    a5,a5,0x2
623    38000058:       00f707b3                add     a5,a4,a5
624    3800005c:       0007a023                sw      zero,0(a5)
625    38000060:       fec42783                lw      a5,-20(s0)
626    38000064:       00178793                addi    a5,a5,1
627    38000068:       fef42623                sw      a5,-20(s0)
628    3800006c:       fec42703                lw      a4,-20(s0)
629    38000070:       00a00793                li      a5,10
630    38000074:       fce7d2e3                bge     a5,a4,38000038 <initfir+0x14>
631    38000078:       00000013                nop
632    3800007c:       00000013                nop
633    38000080:       01c12403                lw      s0,28(sp)
634    38000084:       02010113                addi    sp,sp,32
635    38000088:       00008067                ret
636
637    3800008c <fir>:
638    3800008c:       fe010113                addi    sp,sp,-32
639    38000090:       00112e23                sw      ra,28(sp)
640    38000094:       00812c23                sw      s0,24(sp)
641    38000098:       00912a23                sw      s1,20(sp)
642    3800009c:       02010413                addi    s0,sp,32
643    380000a0:       f85ff0ef                jal     ra,38000024 <initfir>
644    380000a4:       fe042623                sw      zero,-20(s0)
645    380000a8:       fe042423                sw      zero,-24(s0)
646    380000ac:       0f00006f                j       3800019c <fir+0x110>
647    380000b0:       02c00713                li      a4,44
648    380000b4:       fe842783                lw      a5,-24(s0)
649    380000b8:       00279793                slli    a5,a5,0x2
650    380000bc:       00f707b3                add     a5,a4,a5
```

Fig. 5

In counter_la_fir.out (see Figure 6 below), we can see the contents of the address and instruction stored in these compile's assembly code, as well as the 16-bit conversion into machine code The code, such as the first instruction of the behavior section in Figure 599, is "mv a2,a0", and the corresponding machine code is "00050613" (in hexadecimal), which will be stored in the address of "0x38000000".



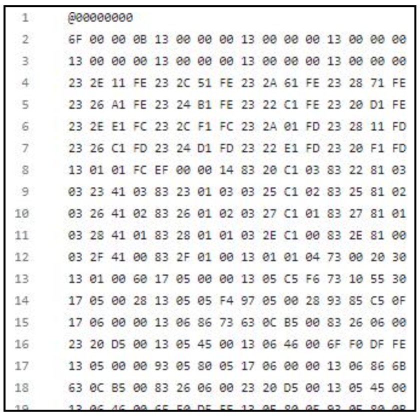
```
Disassembly of section .mprjram:

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

38000024 <initfir>:
38000024: fe010113          addi  sp,sp,-32
38000028: 00812e23          sw  s0,28(sp)
3800002c: 02010413          addi  s0,sp,32
38000030: fe042623          sw  zero,-20(s0)
38000034: 0380006f          j 3800006c <initfir+0x48>
38000038: 05c00713          li  a4,92
3800003c: fec42783          lw  a5,-20(s0)
38000040: 00279793          slli  a5,a5,0x2
38000044: 00f707b3          add a5,a4,a5
38000048: 0007a023          sw  zero,0(a5)
3800004c: 08800713          li  a4,136
38000050: fec42783          lw  a5,-20(s0)
38000054: 00279793          slli  a5,a5,0x2
38000058: 00f707b3          add a5,a4,a5
3800005c: 0007a023          sw  zero,0(a5)
38000060: fec42783          lw  a5,-20(s0)
38000064: 00178793          addi  a5,a5,1
38000068: fef42623          sw  a5,-20(s0)
3800006c: fec42703          lw  a4,-20(s0)
38000070: 00a00793          li  a5,10
38000074: fce7d2e3          bge a5,a4,38000038 <initfir+0x14>
38000078: 00000013          nop
3800007c: 00000013          nop
38000080: 01c12403          lw  s0,28(sp)
38000084: 02010113          addi  sp,sp,32
38000088: 00008067          ret
```

Fig. 6 counter_la_fir.out

After compiling in fir.c, the file counter_la_fir.hex is also generated (as shown in Fig. 7). This file is actually introduced by the testbench and will be consumed by the Caravel SoC (in this lab, the code for this part is stored in the user project BRAM). It can be seen that the addresses start from @00000000 and only machine code is written out. By comparing this with the machine code in counter_la_fir.out, we find that the file is divided into three parts, as shown in Table 2.



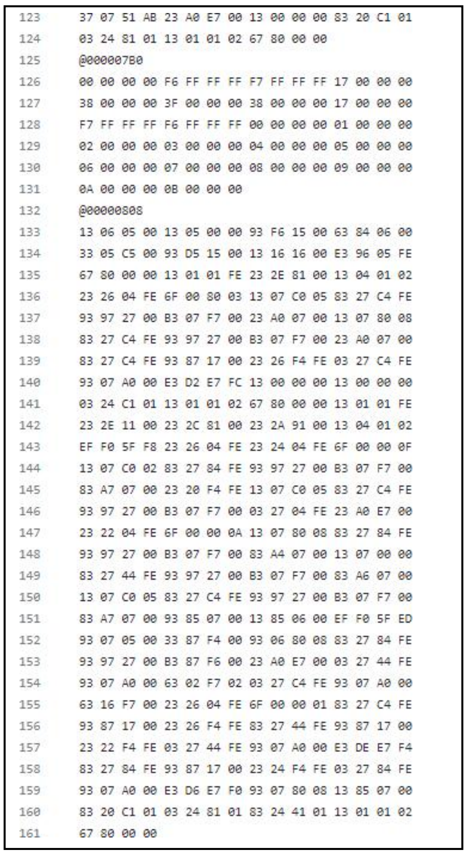
```
@00000000
6F 00 00 0B 13 00 00 00 13 00 00 00 13 00 00 00
13 00 00 00 13 00 00 00 13 00 00 00 13 00 00 00
23 2E 11 FE 23 2C 51 FE 23 2A 61 FE 23 28 71 FE
23 26 A1 FE 23 24 B1 FE 23 22 C1 FE 23 20 D1 FE
23 2E E1 FC 23 2C F1 FC 23 2A 01 FD 23 28 11 FD
23 26 C1 FD 23 24 D1 FD 23 22 E1 FD 23 20 F1 FD
13 01 01 FC EF 00 00 14 83 20 C1 03 83 22 81 03
03 23 41 03 83 23 01 03 03 25 C1 02 83 25 81 02
03 26 41 02 83 26 01 02 03 27 C1 01 83 27 81 01
03 28 41 01 83 28 01 01 03 2E C1 00 83 2E 81 00
03 2F 41 00 83 2F 01 00 13 01 01 04 73 00 20 30
13 01 00 60 17 05 00 00 13 05 C5 F6 73 10 55 30
17 05 00 28 13 05 05 F4 97 05 00 28 93 85 C5 0F
17 06 00 00 13 06 86 73 63 0C B5 00 83 26 06 00
23 20 D5 00 13 05 45 00 13 06 46 00 6F F0 DF FE
13 05 00 00 93 05 80 05 17 06 00 00 13 06 86 6B
63 0C B5 00 83 26 06 00 23 20 D5 00 13 05 45 00
```

Fig. 7 counter_la_fir.hex

| counter_la_fir.hex（Fig. 7） | counter_la_fir.out（Fig. 6） | software |
|---|---|---|
| @00000000~@000007B0 | 0x10000000~0x100007a8 section | counter_la_fir.c |
| @000007B0~@00000808 | Lines 544-594 (memory array) | fir.h (Tap[N]、inputsignal[N]…) |
| @00000808 below | 0x38000000 section | fir.c |

Table 2

The code we wrote in fir.c corresponds to the paragraph below @00000808 (such as Fig. 8). For example, the first instruction is "mv a2, a0" mentioned earlier, and the corresponding machine code is "00050613" (in hexadecimal), but here because the Endian definition of the machine reading is different (little-Endian vs. big-endian), so the order between bytes needs to be changed to "13060500", that is, the first 8 digits in line 133 of Figure 8.



```
123    37 07 51 AB 23 A0 E7 00 13 00 00 00 83 20 C1 01
124    03 24 81 01 13 01 01 02 67 80 00 00
125    @000007B0
126    00 00 00 00 F6 FF FF FF F7 FF FF FF 17 00 00 00
127    38 00 00 00 3F 00 00 00 38 00 00 00 17 00 00 00
128    F7 FF FF FF F6 FF FF FF 00 00 00 00 01 00 00 00
129    02 00 00 00 03 00 00 00 04 00 00 00 05 00 00 00
130    06 00 00 00 07 00 00 00 08 00 00 00 09 00 00 00
131    0A 00 00 00 0B 00 00 00
132    @00000808
133    13 06 05 00 13 05 00 00 93 F6 15 00 63 84 06 00
134    33 05 C5 00 93 D5 15 00 13 16 16 00 E3 96 05 FE
135    67 80 00 00 13 01 01 FE 23 2E 81 00 13 04 01 02
136    23 26 04 FE 6F 00 80 03 13 07 C0 05 83 27 C4 FE
137    93 97 27 00 B3 07 F7 00 23 A0 07 00 13 07 80 08
138    83 27 C4 FE 93 97 27 00 B3 07 F7 00 23 A0 07 00
139    83 27 C4 FE 93 87 17 00 23 26 F4 FE 03 27 C4 FE
140    93 07 A0 00 E3 D2 E7 FC 13 00 00 00 13 00 00 00
141    03 24 C1 01 13 01 01 02 67 80 00 00 13 01 01 FE
142    23 2E 11 00 23 2C 81 00 23 2A 91 00 13 04 01 02
143    EF F0 5F F8 23 26 04 FE 23 24 04 FE 6F 00 00 0F
144    13 07 C0 02 83 27 84 FE 93 97 27 00 B3 07 F7 00
145    83 A7 07 00 23 20 F4 FE 13 07 C0 05 83 27 C4 FE
146    93 97 27 00 B3 07 F7 00 03 27 04 FE 23 A0 E7 00
147    23 22 04 FE 6F 00 00 0A 13 07 80 08 83 27 84 FE
148    93 97 27 00 B3 07 F7 00 83 A4 07 00 13 07 00 00
149    83 27 44 FE 93 97 27 00 B3 07 F7 00 83 A6 07 00
150    13 07 C0 05 83 27 C4 FE 93 97 27 00 B3 07 F7 00
151    83 A7 07 00 93 85 07 00 13 85 06 00 EF F0 5F ED
152    93 07 05 00 33 87 F4 00 93 06 80 08 83 27 84 FE
153    93 97 27 00 B3 87 F6 00 23 A0 E7 00 03 27 44 FE
154    93 07 A0 00 63 02 F7 02 03 27 C4 FE 93 07 A0 00
155    63 16 F7 00 23 26 04 FE 6F 00 00 01 83 27 C4 FE
156    93 87 17 00 23 26 F4 FE 83 27 44 FE 93 87 17 00
157    23 22 F4 FE 03 27 44 FE 93 07 A0 00 E3 DE E7 F4
158    83 27 84 FE 93 87 17 00 23 24 F4 FE 03 27 84 FE
159    93 07 A0 00 E3 D6 E7 F0 93 07 80 08 13 85 07 00
160    83 20 C1 01 03 24 81 01 83 24 41 01 13 01 01 02
161    67 80 00 00
```

Fig. 8 counter_la_fir.hex

Therefore, firmware code is divided into three regions with the following sizes:

| software | counter_la_fir.out | counter_la_fir.hex | address | memory size |
|---|---|---|---|---|
| counter_la_fir.c | 0x10000000~ 0x100007a8 section | @00000000~@000007B0 | 1964 | **1964 Bytes** |
| fir.h (Tap[N]、inputsignal[N]…) | Lines 544-594 (memory array) | @000007B0~@00000808 | 88 | **88 Bytes** |
| fir.c | 0x38000000 section | @00000808 below | 452 | **452 Bytes** |

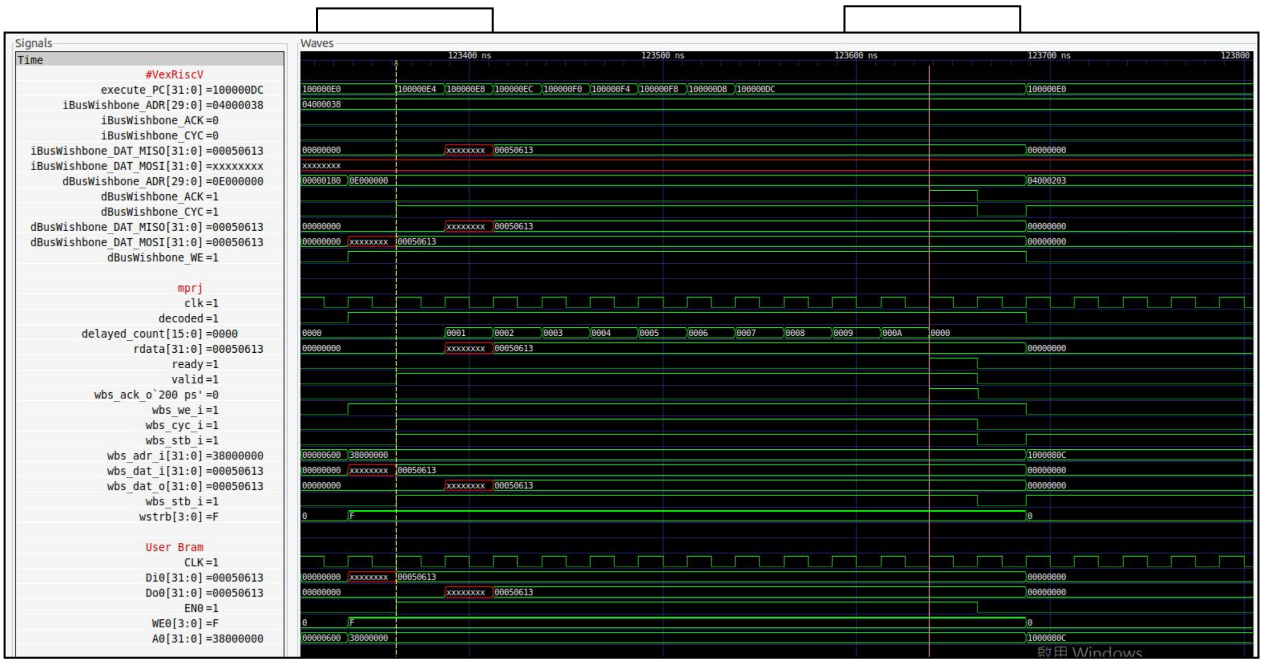
Table 3

The calculation method for the two column on the far right of this table is as follows: Taking the previous example "mv a2, a0" as an example, the corresponding machine code is "00050613" (in hexadecimal) (written as "13 06 05 00" in counter_la_fir.hex). Since this is a 16-bit representation, the digits are "1", "3", "0", "6", and so on "0", "5", "0", "0" each represent 4 bits, and every two digits form a 1 byte. Therefore, by counting the number of groups (each consisting of two digits) in the counter_la_fir.hex file, we can determine how many memory units are required to store the firmware code/machine code for that region, which gives us the value in the rightmost column. Additionally, from Fig. 3, where instructions, machine code, and their corresponding addresses are compared, it is known that one line of assembly code requires 4bytes (32 bits, 4 addresses) to store, and each address stores 1 byte (=8 bits). This allows us to calculate how many addresses are needed to store these firmware codes.

To calculate the total allocate memory space required for all these firmware code, add up the three: a total of 1964+88+452=2504 (Bytes), and if you only want to look at fir.h and fir.c, the amount of memory space used is 88+452=540 (Bytes).

## 2. Interface between bram and wishbone

### 2-1. Waveform from xsim



```
Signals
Time
#VexRiscV
execute_PC[31:0] =100000DC
iBusWishbone_ADR[29:0] =04000038
iBusWishbone_ACK =0
iBusWishbone_CYC =0
iBusWishbone_DAT_MISO[31:0] =00050613
iBusWishbone_DAT_MOSI[31:0] =xxxxxxxx
dBusWishbone_ADR[29:0] =0E000000
dBusWishbone_ACK =1
dBusWishbone_CYC =1
dBusWishbone_DAT_MISO[31:0] =00050613
dBusWishbone_DAT_MOSI[31:0] =00050613
dBusWishbone_WE =1

mprj
clk =1
decoded =1
delayed_count[15:0] =0000
rdata[31:0] =00050613
ready =1
valid =1
wbs_ack_o`200 ps' =0
wbs_we_i =1
wbs_cyc_i =1
wbs_stb_i =1
wbs_adr_i[31:0] =38000000
wbs_dat_i[31:0] =00050613
wbs_dat_o[31:0] =00050613
wbs_stb_i =1
wstrb[3:0] =F

User Bram
CLK =1
Di0[31:0] =00050613
Do0[31:0] =00050613
EN0 =1
WE0[3:0] =F
A0[31:0] =38000000
```

Fig 1 waveform of xsim

The red box in the figure is the signal of WB, and the bottom part of the figure is the access signal of BRAM. The signal is operated according to the waveform in submission guide. Its operation starts when the STB and CYC of WB are 1, and the position of check address is indeed the address assigned to the beginning of user project 0x380 BRAM requires dat (a package containing the instruction and tap[N], inputbuffer[N] and other data that the CPU wants to execute, The addresses are stored separately in the upper Table 3). When WE (write enable) is 1, it writes to BRAM; When WE is 0, data is read from BRAM, and address is given addres (s 0x380 by WB (Start). The process of accessing the BRAM will take 2 cycles, and according to the problem setting, it will take another delay 10 One cycle can only return ACK and data from BRAM in WB, and these 10 cycle are used The register "delayed_count" will return to 0 when reset and start counting upwards while accessing the BRAM, until it reaches 10. At this point, it will ACK to allow external access to wbs_dat_o and reset to 0, preparing for the next access. Therefore, from "STB and CYC = 1" to "ACK, pulled up," a total of 11 cycles have passed (the 12th cycle is pulled up).

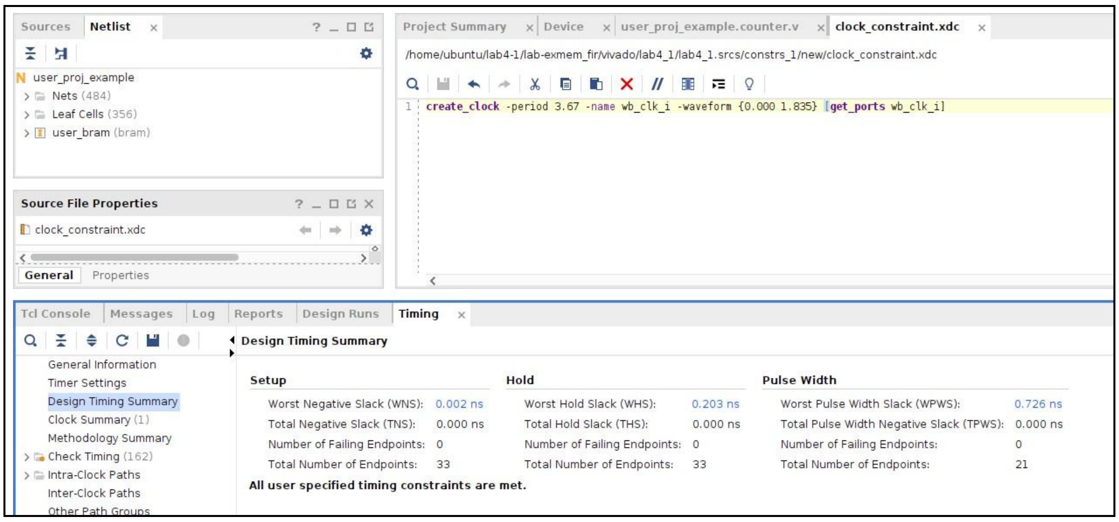
## 2-2. FSM

In this lab, the FSM is not specifically utilized. Instead, the output signal is obtained through logical OR operations on the input signal and the BRAM's output signal, or by direct wiring. This includes using the address decode to confirm that the WB's address is 0x380, indicating that the slave is accessing data at the address of user project, before outputting the ACK and output data signals. For example:

```
assign decoded = wbs_adr_i[31:20] == 12'h380 ? 1'b1 : 1'b0;
assign valid = wbs_cyc_i && wbs_stb_i && decoded;
assign wbs_dat_o = rdata;
assign wbs_ack_o = ready;
```

## 3. Synthesis report

We will add design (including fir.c, fir.h, user_proj_example.counter.v, bram.v, and modify the testbench to detect the output signal and verify its correctness) to the Vivado software (simulation is performed through the run_sim script, without using Vivado), and after adding timing constraints, execute synthesis. The result shown in the figure below can be seen, with the highest speed approximately equal to the clock period ≈ 3.67ns, corresponding to a speed of about 272.48MHz. At this point, the slack is positive.



/home/ubuntu/lab4-1/lab-exmem_fir/vivado/lab4_1/lab4_1.srcs/constrs_1/new/clock_constraint.xdc

```
create_clock -period 3.67 -name wb_clk_i -waveform {0.000 1.835} [get_ports wb_clk_i]
```

Netlist: user_proj_example — Nets (484), Leaf Cells (356), user_bram (bram)

Design Timing Summary

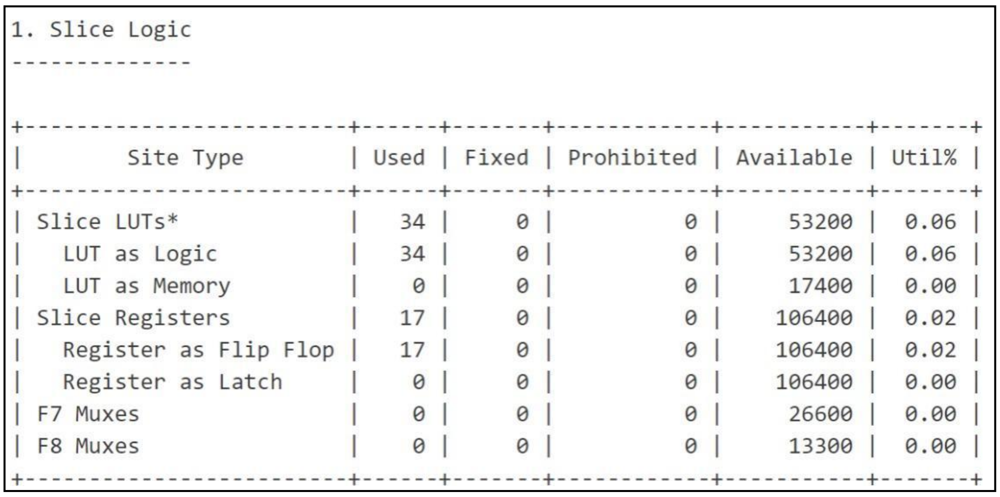
| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.002 ns | Worst Hold Slack (WHS): | 0.203 ns | Worst Pulse Width Slack (WPWS): | 0.726 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 33 | Total Number of Endpoints: | 33 | Total Number of Endpoints: | 21 |

All user specified timing constraints are met.

### (1) User_proj_example_utilization_synth.rpt

After executing synthesis, the synthesized utilization report, namely user_proj_example, can be obtained _utilization_synth.rpt. Opening this file will give you the following resource usage:

FF & LUT:

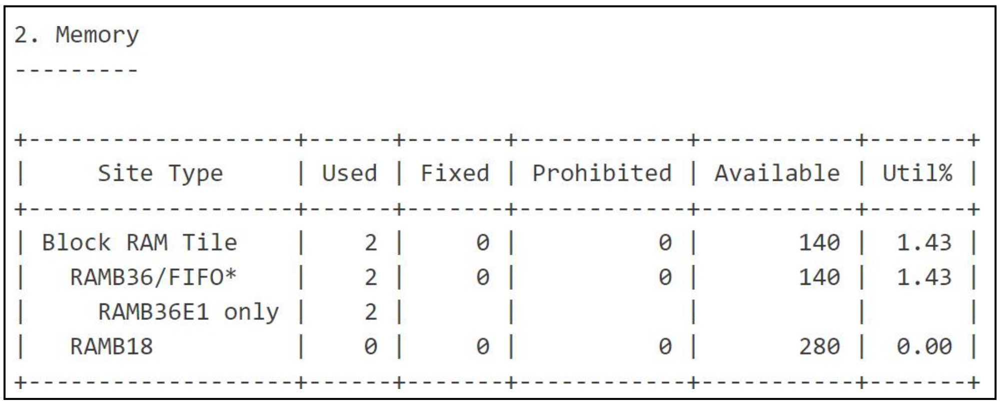


```
1. Slice Logic
--------------

+-------------------------+------+-------+------------+-----------+-------+
|        Site Type        | Used | Fixed | Prohibited | Available | Util% |
+-------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*             |   34 |     0 |          0 |     53200 |  0.06 |
|   LUT as Logic          |   34 |     0 |          0 |     53200 |  0.06 |
|   LUT as Memory         |    0 |     0 |          0 |     17400 |  0.00 |
| Slice Registers         |   17 |     0 |          0 |    106400 |  0.02 |
|   Register as Flip Flop |   17 |     0 |          0 |    106400 |  0.02 |
|   Register as Latch     |    0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                |    0 |     0 |          0 |     26600 |  0.00 |
| F8 Muxes                |    0 |     0 |          0 |     13300 |  0.00 |
+-------------------------+------+-------+------------+-----------+-------+
```

Fig 2 Slice Logic

BRAM:

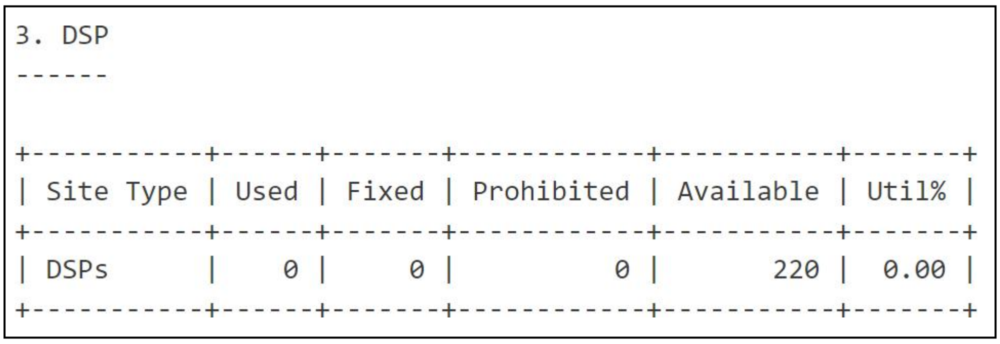


```
2. Memory
---------

+-------------------+------+-------+------------+-----------+-------+
|     Site Type     | Used | Fixed | Prohibited | Available | Util% |
+-------------------+------+-------+------------+-----------+-------+
| Block RAM Tile    |    2 |     0 |          0 |       140 |  1.43 |
|   RAMB36/FIFO*    |    2 |     0 |          0 |       140 |  1.43 |
|     RAMB36E1 only |    2 |       |            |           |       |
|   RAMB18          |    0 |     0 |          0 |       280 |  0.00 |
+-------------------+------+-------+------------+-----------+-------+
```

Fig 3 Memory

DSP:

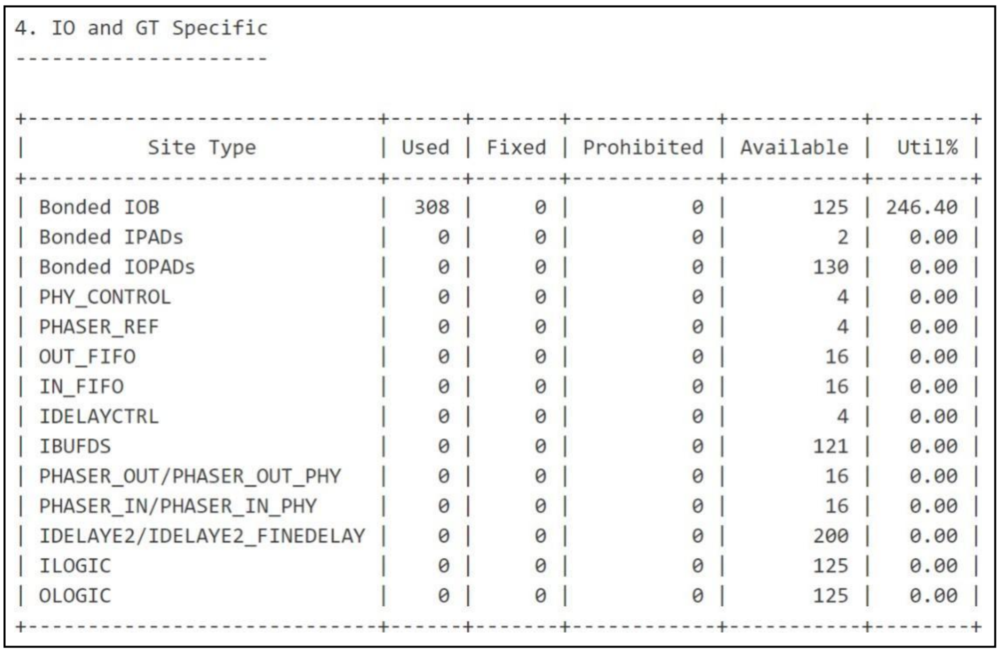


```
3. DSP
------

+-----------+------+-------+------------+-----------+-------+
| Site Type | Used | Fixed | Prohibited | Available | Util% |
+-----------+------+-------+------------+-----------+-------+
| DSPs      |    0 |     0 |          0 |       220 |  0.00 |
+-----------+------+-------+------------+-----------+-------+
```

Fig 4 DSP

IO:



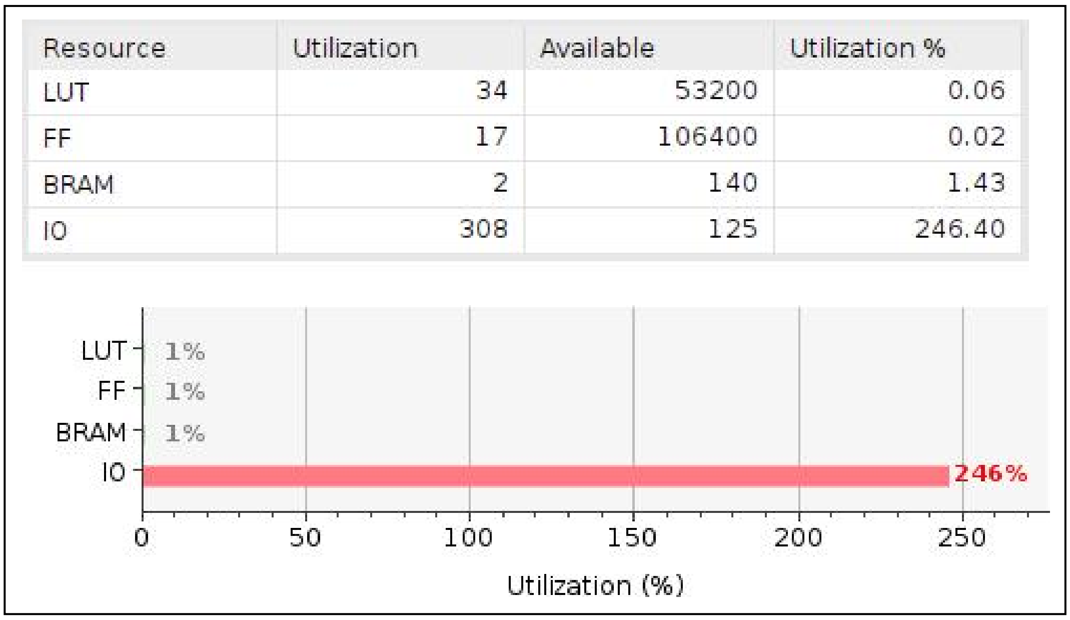
```
4. IO and GT Specific
---------------------

+-----------------------------+------+-------+------------+-----------+--------+
|          Site Type          | Used | Fixed | Prohibited | Available |  Util% |
+-----------------------------+------+-------+------------+-----------+--------+
| Bonded IOB                  |  308 |     0 |          0 |       125 | 246.40 |
| Bonded IPADs                |    0 |     0 |          0 |         2 |   0.00 |
| Bonded IOPADs               |    0 |     0 |          0 |       130 |   0.00 |
| PHY_CONTROL                 |    0 |     0 |          0 |         4 |   0.00 |
| PHASER_REF                  |    0 |     0 |          0 |         4 |   0.00 |
| OUT_FIFO                    |    0 |     0 |          0 |        16 |   0.00 |
| IN_FIFO                     |    0 |     0 |          0 |        16 |   0.00 |
| IDELAYCTRL                  |    0 |     0 |          0 |         4 |   0.00 |
| IBUFDS                      |    0 |     0 |          0 |       121 |   0.00 |
| PHASER_OUT/PHASER_OUT_PHY   |    0 |     0 |          0 |        16 |   0.00 |
| PHASER_IN/PHASER_IN_PHY     |    0 |     0 |          0 |        16 |   0.00 |
| IDELAYE2/IDELAYE2_FINEDELAY |    0 |     0 |          0 |       200 |   0.00 |
| ILOGIC                      |    0 |     0 |          0 |       125 |   0.00 |
| OLOGIC                      |    0 |     0 |          0 |       125 |   0.00 |
+-----------------------------+------+-------+------------+-----------+--------+
```

Fig 5 IO and GT Specific

**(2) In vivado utilization summary**

In addition, pressing "Report Utilization" in Vivado will produce the following (more visual organization):



| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 34 | 53200 | 0.06 |
| FF | 17 | 106400 | 0.02 |
| BRAM | 2 | 140 | 1.43 |
| IO | 308 | 125 | 246.40 |

From point (1) and (2) of report, it can be seen that this design is used after synthesis

1. The number of FF is 17, and there are 106400 in the board of this FPGA, so utilization is 0.02%
2. The LUT has 34 numbers, and the board of this FPGA has 53200, so utilization is 0.06%
3. BRAM uses 2, and this FPGA, the board has a total of 140, so utilization is 1.43%

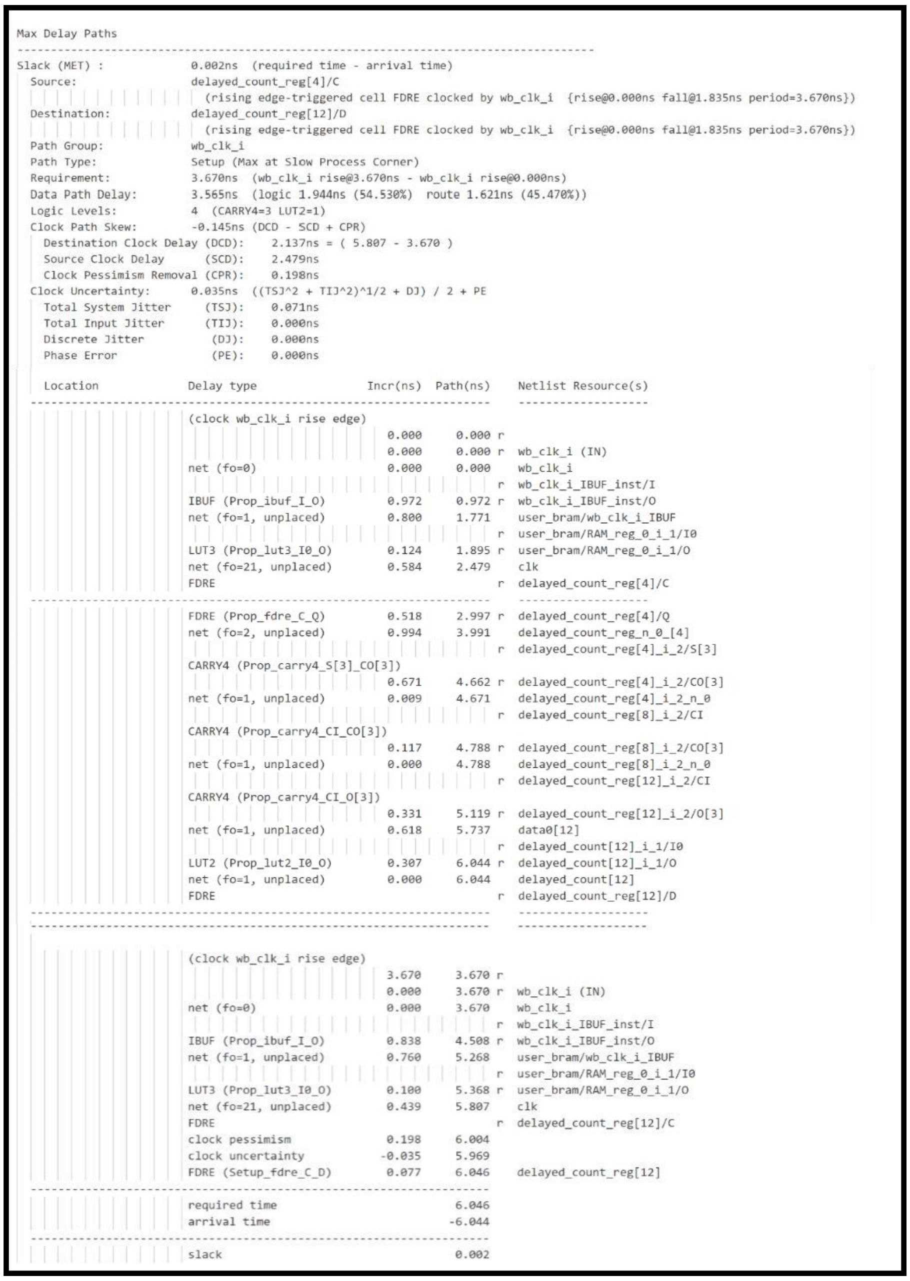
**(3) User_proj_example.vds**

```
Detailed RTL Component Info :
+---Adders :
|   |    2 Input   16 Bit       Adders := 1
+---Registers :
|   |   |   |   |     32 Bit    Registers := 1
|   |   |   |   |     16 Bit    Registers := 1
|   |   |   |   |      1 Bit    Registers := 1
+---RAMs :
|   |   |   |   |    64K Bit   (2048 X 32 bit)          RAMs := 1
+---Muxes :
|   |    2 Input   32 Bit        Muxes := 6
|   |    2 Input   16 Bit        Muxes := 1
|   |    2 Input    8 Bit        Muxes := 1
|   |    2 Input    1 Bit        Muxes := 2
```

## (4) Timing_report_3_67ns.txt

After incorporating the design into Vivado, perform synthesis and set the maximum speed as much as possible (we initially used a clock period of 10ns for synthesis, which resulted in a slack of over +6 points, so we can further increase the operation speed: the maximum speed this design can achieve is approximately clock period = 3.67ns, translating to a maximum frequency of about 272.48 MHz). Following the procedures outlined in Lab3, the SYN_Workflow.pptx document, you can obtain the post-synthesis timing report, namely timing_report.txt, which includes information on the longest path: (due to the length of the file, it will be captured in multiple screenshots)

**Clock period: 3.67ns**



```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :             0.002ns  (required time - arrival time)
  Source:                 delayed_count_reg[4]/C
                            (rising edge-triggered cell FDRE clocked by wb_clk_i  {rise@0.000ns fall@1.835ns period=3.670ns})
  Destination:            delayed_count_reg[12]/D
                            (rising edge-triggered cell FDRE clocked by wb_clk_i  {rise@0.000ns fall@1.835ns period=3.670ns})
  Path Group:             wb_clk_i
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            3.670ns  (wb_clk_i rise@3.670ns - wb_clk_i rise@0.000ns)
  Data Path Delay:        3.565ns  (logic 1.944ns (54.530%)  route 1.621ns (45.470%))
  Logic Levels:           4  (CARRY4=3 LUT2=1)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.137ns = ( 5.807 - 3.670 )
    Source Clock Delay      (SCD):    2.479ns
    Clock Pessimism Removal (CPR):    0.198ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns

    Location             Delay type                Incr(ns)  Path(ns)    Netlist Resource(s)
  -------------------------------------------------------------------    -------------------
                         (clock wb_clk_i rise edge)
                                                      0.000     0.000 r
                                                      0.000     0.000 r  wb_clk_i (IN)
                         net (fo=0)                   0.000     0.000    wb_clk_i
                                                                      r  wb_clk_i_IBUF_inst/I
                         IBUF (Prop_ibuf_I_O)         0.972     0.972 r  wb_clk_i_IBUF_inst/O
                         net (fo=1, unplaced)         0.800     1.771    user_bram/wb_clk_i_IBUF
                                                                      r  user_bram/RAM_reg_0_i_1/I0
                         LUT3 (Prop_lut3_I0_O)        0.124     1.895 r  user_bram/RAM_reg_0_i_1/O
                         net (fo=21, unplaced)        0.584     2.479    clk
                         FDRE                                         r  delayed_count_reg[4]/C
  -------------------------------------------------------------------    -------------------
                         FDRE (Prop_fdre_C_Q)         0.518     2.997 r  delayed_count_reg[4]/Q
                         net (fo=2, unplaced)         0.994     3.991    delayed_count_reg_n_0_[4]
                                                                      r  delayed_count_reg[4]_i_2/S[3]
                         CARRY4 (Prop_carry4_S[3]_CO[3])
                                                      0.671     4.662 r  delayed_count_reg[4]_i_2/CO[3]
                         net (fo=1, unplaced)         0.009     4.671    delayed_count_reg[4]_i_2_n_0
                                                                      r  delayed_count_reg[8]_i_2/CI
                         CARRY4 (Prop_carry4_CI_CO[3])
                                                      0.117     4.788 r  delayed_count_reg[8]_i_2/CO[3]
                         net (fo=1, unplaced)         0.000     4.788    delayed_count_reg[8]_i_2_n_0
                                                                      r  delayed_count_reg[12]_i_2/CI
                         CARRY4 (Prop_carry4_CI_O[3])
                                                      0.331     5.119 r  delayed_count_reg[12]_i_2/O[3]
                         net (fo=1, unplaced)         0.618     5.737    data0[12]
                                                                      r  delayed_count[12]_i_1/I0
                         LUT2 (Prop_lut2_I0_O)        0.307     6.044 r  delayed_count[12]_i_1/O
                         net (fo=1, unplaced)         0.000     6.044    delayed_count[12]
                         FDRE                                         r  delayed_count_reg[12]/D
  -------------------------------------------------------------------    -------------------
  -------------------------------------------------------------------    -------------------

                         (clock wb_clk_i rise edge)
                                                      3.670     3.670 r
                                                      0.000     3.670 r  wb_clk_i (IN)
                         net (fo=0)                   0.000     3.670    wb_clk_i
                                                                      r  wb_clk_i_IBUF_inst/I
                         IBUF (Prop_ibuf_I_O)         0.838     4.508 r  wb_clk_i_IBUF_inst/O
                         net (fo=1, unplaced)         0.760     5.268    user_bram/wb_clk_i_IBUF
                                                                      r  user_bram/RAM_reg_0_i_1/I0
                         LUT3 (Prop_lut3_I0_O)        0.100     5.368 r  user_bram/RAM_reg_0_i_1/O
                         net (fo=21, unplaced)        0.439     5.807    clk
                         FDRE                                         r  delayed_count_reg[12]/C
                         clock pessimism              0.198     6.004
                         clock uncertainty           -0.035     5.969
                         FDRE (Setup_fdre_C_D)        0.077     6.046    delayed_count_reg[12]
  -------------------------------------------------------------------
                         required time                          6.046
                         arrival time                          -6.044
  -------------------------------------------------------------------
                         slack                                  0.002
```

As can be seen from the figure above, the timing/delay of this longest path is 3.565ns, while its slack is 0.002ns, which is a positive value