课程项目报告（二）

课程名称：硬件描述语言原理与应用

项目内容：浮点单元设计

姓　　名：周开宁

学　　号：3180101148

院　　系：信息与电子工程学院

专业年级：2018 级微电子科学与工程专业

指导教师：沈海斌

完成时间：2020 年 11 月 2 日

目录

一、 设计目的及要求

本项目包括两部分：设计部分和仿真验证部分。

设计部分：实现单精度的浮点运算单元。支持 Round to Nearest 舍入模式，并实现异常处理。设计部分的指令部分包括实现 12 条浮点指令 add.s/add.ps、sub.s/sub.ps 、 mul.s/mul.ps 、 div.s 、 cvt.ps.s 、 cvt.s.w/cvt.w.s 、cvt.s.pl/cvt.s.pu，以及 4 条支持指令 lui/ori、mfc1/mtc1；设计部分的运算部分支持浮点标准协议的部分功能，实现单精度运算，包括浮点加/减/乘/除、格式转换，其中舍入模式为 round to nearest，异常的类型有 overflow/underflow/inexact/divide by 0。

仿真验证部分：要求通过 lui/ori、mtc1/mfc1 这 4 条指令完成对浮点单元的寄存器的设置与结果取回。如果要实现综合的话，可以将 Testbench 中的主处理器写成可综合形式；或者，修改综合脚本对指定的 FPU 模块进行综合。

二、 设计步骤

由于这个项目的复杂性，我采用的是自顶向下的设计思路，这样有利于理清各个模块之间的调用关系，它们的端口以及连接。

1、 理解 Processor 文件夹下的代码，确定 tb.v、procossor.v、memory.v 之间的关系。

2、 定义 fpu 模块的定位和接口，和 x_processor 模块进行通信。

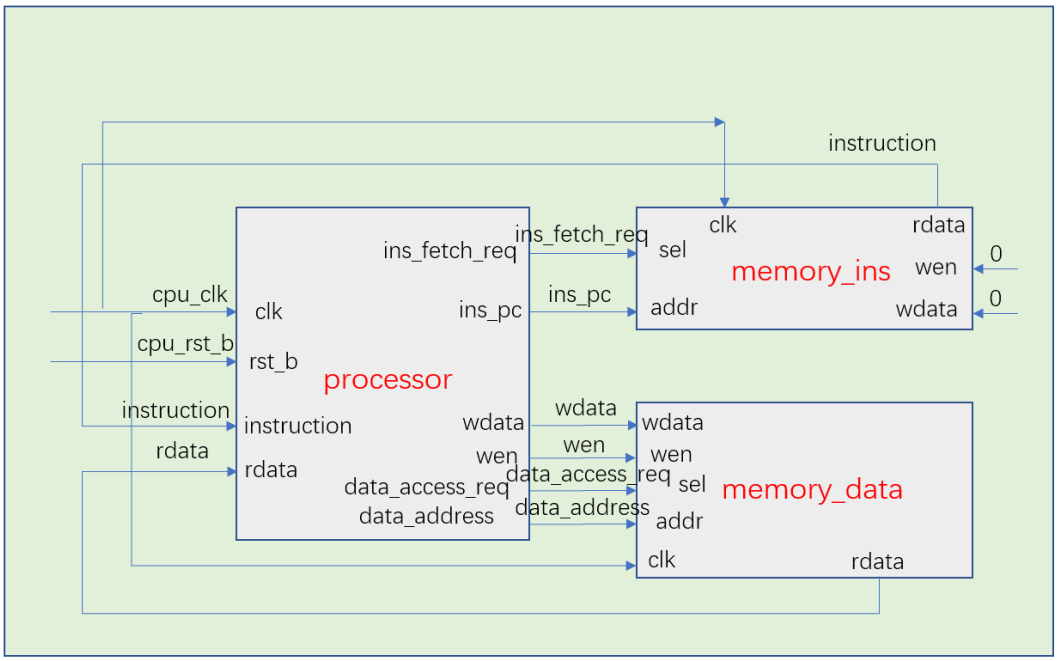
3、 定义 fpu 内部的模块和功能，主要实现浮点单元的运算。

4、 定义 fpu 的时序逻辑：含有三级流水线，能加大数据的吞吐量，加快浮点单元的运算。
5、 书写 Python 代码，可以将汇编语言转化为机器码，即一个简单的汇编器。
6、 确定汇编指令，时整个处理器能在初始化过以后自动运行。
7、 书写 C++的验证程序，能输出浮点数在内存中存储的十六进制码，以及给定某一十六进制码，将其转化为浮点数。这作为模块的验证功能，能清晰明了地看到内存中存储地浮点数的形式，加快验证的速度和准度。
8、 编写运行脚本，能统一地将上述操作合在一起，运行整个模型。
9、 进行总体仿真，查看波形。
10、 进行验证、调试。

三、 设计内容

1、 总体框图及理解

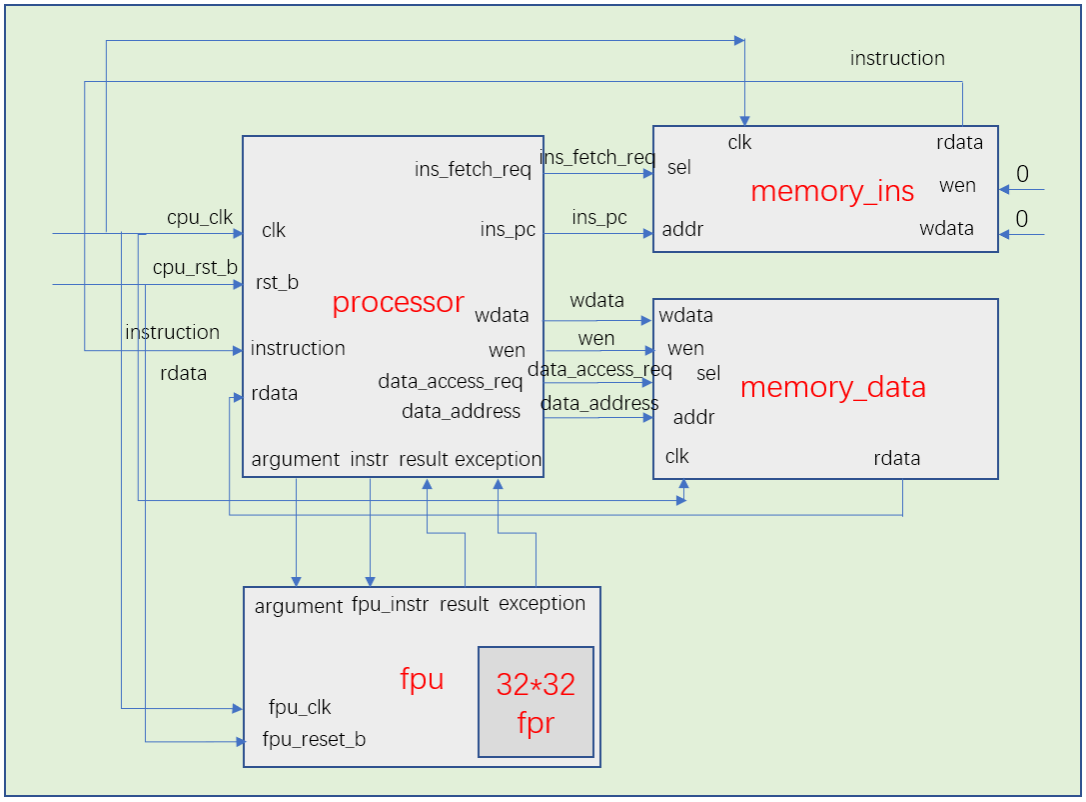
图 1 是原本 tb 内部处理器（processor）和两个存储器（x_ins_memory、x_data_memory）之间地端口连接关系。



图表 1 原本 tb.v 文件下三个模块的连接关系

根据设计的要求，根据我自己的理解，整个处理器分为以下几个部分：（1）中央处理器，对应框图里的 processor，用于 32-bit 的指令取指和译码。（2）数据存储单元，对应框图里的 memory_data，作为 GPR（general purpose register），存储数据并与 processor 连接相应端口。（3）指令存储单元，对应框图里的 memory_ins，作为指令存储的地方，processor 从中一条一条地读取指令，并进行译码。我把中央处理器 processor 作为主处理器，fpu 浮点单元作为协处理器

coprocessor 1，和 processor 独立来开，并定义相关连接通信端口；fpu 中内含 32*32-bit 的 FPR(floating point register),用于存储运算所需的浮点数；同时内含各种具体的浮点运算电路。(4) testbench 集成上述模块，因此重新定义 tb 的框图，如图 2 所示。



tb

图表 2 本设计 tb.v 里各个模块的关系

2、 总体架构行为描述

浮点运算单元的功能实现主要分为以下几个步骤。

(1) 第一步，在时钟上升边沿来到时，主处理器 processor 根据当前 pc，从指令存储单元 memory_ins 中读取相应的指令。

(2) 第二步，processor 对读到的指令进行译码。若是 lui/ori 指令，直接把相应数据传输到 memory_data 之中；若是其他指令，则将控制指令 instr 和相关参数 argument 传输到协处理器 fpu。

(3) 第三步，fpu 接收到来自主处理器的输入，根据控制信号和参数，进行相应的浮点加/减/乘/除及其他运算。其中运算是三级流水线的结构。

(4) 第四步，fpu 返回结果 result 和异常状态 exception 给 processor，processor 读取计算结果并观察异常情况。

(5)第五步，主处理器 processor 和协处理器 fpu 通过 mfc1 和 mtc1 进行通信，在 GPR 和 FPR 之间进行传输数据。

3、 模块代码描述

(1) memory

memory 同时做指令存储单元和数据的存储单元 GPR，有读和写两个功能。写入行为需要有效使能信号和时钟上边沿的触发才能进行，而读的行为是可以持续进行的。

由于原本的 memory 的读写地址是共用的，我发现这样的特性在实现 ori 命令的时候会有问题，一个周期内要对 data_address[6:2]进行两次赋值，这是无法综合的。所以我对 memory 进行修改，将 data_address 的[11:7]作为读指令的地址线，保持原来的[6:2]作为读数据的地址线，将两者分开。

```
always@(posedge clock)
begin
  if(sel && wen)
    // data_cell[addr[6:2]] <= wdata;
    data_cell[addr[11:7]] <= wdata;
end
```

(2) processor

processor 读取所有的指令（包括浮点运算指令，如 adds $fd,$fs,$ft），并输出控制信号。它读的指令有两种情况：一是立即数传输指令 lui 和 ori。此时不涉及 fpu 的操作；二是浮点运算指令，则输出相应控制信号和参数给 fpu。为了实现控制 fpu 的功能，我在原来的 I/O 接口基础上，增加了以下几个 I/O 接口：

```
input  [31:0] result; // return from coprocessor
input         exception;
input  [31:0] data_FPR;
output [3:0]  reg fpu_instr; // to coprocessor
output [31:0] reg argument[3-1:0]; // to coprocessor, array
output reg [32-1:0] data_GPR;
```

其中，新增的两个输出 fpu_instr 和 argument 是控制电路的输出，输出给 fpu，使其进行相应的计算。此时，经过 processor 的译码，原来读入的 32 位机器码指令已经转化为预先由 parameter 定义的一系列 4 位的控制信号，并通过 fpu_instr 输出。同时，通过读取指令中的 fs、ft、fd，也通过 argument 传输给 fpu，给后者操作数和目的数的地址，进行运算和存储。

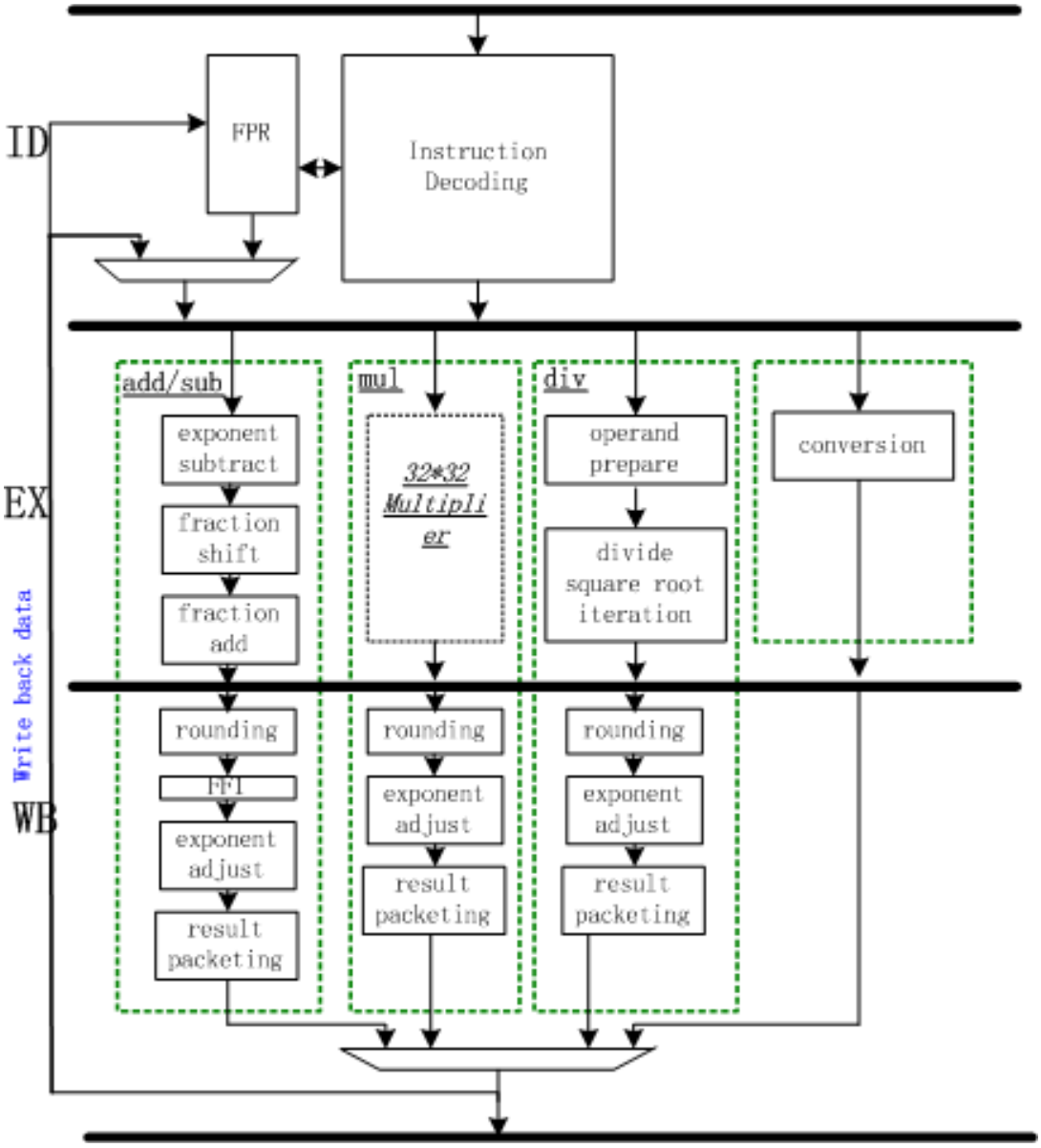
```
    // lui, Load Upper Immediate
    if(opcode==op_lui && instruction[25:21]==5'b0)
    begin
        fpu_instr        = idle;
        wen             = 1'b1;
        data_access_req   = 1'b1;
        data_address[6:2] = rt;
        wdata           = imm << 16; //GPR
        data_GPR         = 32'b0;
    end else if(opcode==op_ori) // ori, Or Immediate
    begin
        fpu_instr        = idle;
        wen             = 1'b1;
        data_access_req   = 1'b1;
        data_address[6:2] = rs; // read
        rdata_temp       = rdata;
        data_address[6:2] = rt; // write
        wdata           = rdata_temp|{16'b0,imm};
        data_GPR         = 32'b0;
    end
```

新增的两个输入是从 fpu 返回的两个值。一个是计算结果 result，一个是异常状态 exception。主处理器通过 exception 的数值来判断计算是否发生了异常。

还有一个输出和一个输入 data_FPR 和 data_GPR，是 fpu 中的 FPR 和作为存储器的 GPR 之间进行通信的通道，分别提供给 mfc1 和 mtc1 指令使用。

(3) fpu

fpu 是主要的设计模块，其中含有各种浮点运算单元。它的输入有时钟信号、复位信号、来自 processor 的指令和参数信号，输出运算结果和异常信号。大致的运算框图如下：



图表 3 fpu 3 级流水线运算框图

其中，fpu 内部放有诸多子模块并将其实例化，这些子模块包括：module_adds、module_addps、module_subs、module_subps、module_muls、module_mulps、module_divs、module_cvtpss、module_cvtsw、module_cvtws，可以执行与其名称相对应的指令的运算。fpu 作为高层结构根据其输入输出特性进行实例化。下面详述各个模块的设计思路。

module_adds

该模块执行 adds 指令。输入为两个 32 位的浮点数，输出为一个 32 位的浮点数和异常状态。加法模块完成需要以下步骤：首先将两个操作数转成正数，然后将指数位对齐进行相加减得到中间结果；再对中间结果进行修正。若尾数 M>=2，则要指数要进位，并位移一位数；若 M<1，则要减少指数，右移一位。若有溢出，则异常位置 1.

module_addps

该模块执行 addps 指令。输入为两个 64 位的数，输出为一个 64 位的浮点数和异常状态。取两个 64 位的上 32 位和下 32 位分别进行加法，因此可以直接实例化 module_adds 来简化设计，把两个 64 位数的前后 32 位分别传入两个子模块加法器 module_adds 作为输入，得到两个 32 位的浮点加法结果后拼接成一个 64 位的数作为输出。异常状态输出是两个子模块的异常的或。

module_subs

该模块执行 subs 指令。输入为两个 32 位的浮点数，输出为一个 32 位的浮点数和异常状态。由于减去一个数可以看成加上一个数的相反数，因此该模块可以直接实例化 module_adds 来简化设计，将第二个操作数的符号位取反后，与第一个操作数一起传入子模块加法器 module_adds 作为输入，得到的结果就是原来减法的结果。异常状态输出子模块的异常状态输出。

module_muls

该模块执行 muls 指令。输入为两个 32 位的浮点数，输出为一个 32 位的浮点数和异常状态。实现乘法通过以下方法：operand1 = (-1) ^ s1 * M1 * 2 ^ E1 ， operand2 = (-1) ^ s1 * M2 * 2 ^ E2 ， product = operand1 * operand2 = (s1 xor s2) * (M1 * M2) ^ (E1 + E2) ，还要处理几种异常情况：（1）如果有任意的 NaN 参与运算，就输出 NaN。（2）如果 E1+E2 上溢和下溢的时候，就要进行异常的处理。（3）经过四舍五入的时候，还要对 M1*M2 以及 E1+E2 进行修正，同样也会引起上述问题以及 inexact 的问题。此时就还要进行异常处理。若发生异常，则异常状态就置 1.

module_mulps

该模块执行 mulps 指令。输入为两个 64 位的数，输出为一个 64 位的浮点数和异常状态。取两个 64 位的上 32 位和下 32 位分别进行乘法，因此可以直接实例化 module_muls 来简化设计，把两个 64 位数的前后 32 位分别传入两个子模块加法器 module_nuls 作为输入，得到两个 32 位的浮点乘法结果后拼接成一个 64 位的数作为输出。异常状态输出是两个子模块的异常的或。

module_divs

该模块执行 divs 指令。输入为两个 32 位的浮点数，输出为一个 32 位的浮点数和异常状态。与乘法不同，被除数需要尽可能移位，以保证被除数不小于除数。相应的

module_cvtpss

该模块执行 cvt.ps.s 指令。输入为两个 32 位的浮点数，输出为一个 64 位的数和异常状态。该模块将两个单精度浮点数拼接在一起形成一个单精度对。所以输出就是两个输入的简单拼接，异常状态恒为 0。

module_cvtsw

该模块执行 cvt.s.w 指令。输入为一个 32 位的定点数，输出为一个 32 位浮点数和异常状态。该模块需要把定点数转化为对应大小的浮点数。定点数可以看成是 32-bit 的数* 2 ^ 0，所以，转化为正数以后，我们只要找到最前面的 1

（msb），兵将小数点移位到那个 1 的后面即可。由于浮点数的表示范围比定点数更大，因此不会产生上溢或者下溢的异常。

module_cvtws

该模块执行 cvt.w.s 指令。输入为一个 32 位的浮点数，输出为一个 32 位定点数和异常状态。该模块需要把浮点数转化为对应大小的定点数。由于浮点数的表示范围比定点数更大，因此可能会产生上溢或者下溢的异常。先确定尾数前置的数“0”还是“1”。如果是非规格化的数，则前面拼接一个“0”；如果是规格化的，则拼接一个“1”。之后根据指数，对尾数进行移位，考虑到可能的下溢和上溢问题，以及舍入以后产生的进位的问题。

对于 cvt.s.pl 和 cvt.s.pu 指令，只需将参数对应的写入 FPR 中即可，所以也不需要单独的模块，直接在译码阶段执行即可。不会有异常发生。需要注意的是，由于 MIPS 默认的是大端规则，所以权重更高的字存储的地址更小，lower halves 的地址要在 upper halves 的地址上加 1.

对于 mfc1 或 mtc1 指令，只需将对应操作数作为 result 或 argument 的参数，传向 GPR 或 FPR，直接在译码阶段执行即可。这条两指令不用通过流水线，我将其设置为，在每个时钟周期上升沿将 data_GPR 的值写入 FPR；通过组合电路直接将 FPR 接到 processor 的输入，作为取出的 data_FPR 的通道。这里不会有异常发生。

流水线

流水线设计也是 fpu 实现的关键部分。流水线的作用流水线设计就是将组合逻辑系统地分割，并在各个部分（分级）之间插入寄存器，并暂存中间数据的方法。目的是提高数据吞吐率，提高处理速度。本 fpu 主要有三个执行阶段：译码、执行和写回。因此我设计了三级流水线。

第一级是译码和初始化操作。

```
// pipeline
// ID process
always @(posedge fpu_clock)
begin  if(!fpu_reset_b)
    begin
        // reset statement
    end else begin
        operand1       <= FPR[argument[0]]; // 32-bit
        // other assigning value statement
    end
```

第二级是执行操作。将各个子模块的运算后的值赋给对应的输出。

```
// execution process
always @(posedge fpu_clock)
begin
    case (reg_fpu_instr)
        idle :begin result64 <= 64'b0; exception1<=1'b0; flag_WB <=
2'b0;end
        adds :begin result64 <= wire32_result[0];
exception1<=wire_exception[0]; flag_WB <= 2'b1 ; end
        // other reg_fpu_instr
        default:begin result64 <= 64'b0; exception<=1'b1 ; flag_WB <=
2'b0 ; end
    endcase
    addr<=dest[4:0];
end
```

第三级是写回和输出操作。对于不用写回的指令，就不写回。

```
//WB process
always @(posedge fpu_clock)
begin
    if (fpu_reset_b)
    begin // determine whether write back
        if (flag_WB==2'b1) // 32-bit
            FPR[reg_dest]<=result64[31:0];
        else if (flag_WB==2'b10) // .ps
        begin
            FPR[reg_dest]   <= result64[63:32];
            FPR[reg_dest+1] <= result64[31:0];
        end
        result <= result64[31:0];
        exception<=exception1;
    end
end
```

4、 汇编指令书写

想要整个 testbench 运行起来，必须依靠正确的指令。由于直接书写机器码过于低效，我写了一个 python 的脚本，可以将上述的 MIPS 汇编语句转化成机器码。

```
with open("sti","w") as sti:
    print("@00000000",file=sti)

with open("sti","a") as sti:
    with open("instr.s") as file:
        instrs=file.read().splitlines()
        for i in range(len(instrs)):
            ins=instrs[i].split()
            print(ins)
            if ins[0]=='lui':
                op=lui
                fmt="00000"
                middle=''.join(ins[1].split(","))
                mc=op+fmt+middle
                mc_16=str(hex(int(mc,2)))[2:]
                print(mc_16,file=sti)
            elif ins[0]=='ori':
                # others
```

由于初始化的时候，memory_data 的所有值都复为 0，在现有指令情况下，必须要用 lui 和 ori 来加载最初的立即数。为了之后的测试，我将所有指令都实现了一遍。其中第一行 nop 是空操作。虽然指令在 MIPS 中不一定存在，但在译码的时候，可以被译为不执行动作。

```
nop
lui 00000,0110101011110000
mtc1 00000,00000
ori 00001,00000,0101010100001111
mtc1 00001,00001
add.s  00010,00001,00000
sub.s  00011,00001,00000
mul.s 00100,00001,00000
div.s 00101,00000,00001
add.ps 00110,00010,00000
…
mfc1 00101,00101
mfc1 00110,00110
```

四、 仿真与验证

接下来对模块进行仿真。

为了更好地展现浮点数在内存中的存储方法，我写了一个 C++程序。该程序能实现十进制浮点数和内存中的二进制数的转化。

```
int main(){
    float a=……;
    float b;
    *(int*)&b=0x……;
    printf("in binary, %x\n",*(int*)&a);
    printf("in float, %f\n",b);
return 0;
}
```

1、 模块仿真

加法的仿真结果如下图。可以看到输入涵盖了异常输入，异常输出、普通输入、加负数正数等等情况。从测试结果看，加法模块功能完全正确。

```
At time=                    0
operand1=01001110011000000001110100001100
operand2=00110110000001100011011110111101
result=01001110011000000001110100001100
exception=0

At time=                    5
operand1=01000010110010000000000000000000
operand2=00111111111111111101111100111011
result=01000010110010111111111101111101
exception=0

At time=                   10
operand1=11111111111111111111111111111111
operand2=00111111111111111101111100111011
result=11111111111111111111111111111111
exception=1

At time=                   15
operand1=01000010110010000000000000000000
operand2=10111111111111111101111100111011
result=01000010110001000000000010000011
exception=0

At time=                   20
operand1=00111111100011001100110011001101
operand2=00111111100110011001100110011010
result=01000000000100110011001100110011
exception=0

At time=                   25
operand1=00111111100011001100110011001101
operand2=10111111100011001100110011001101
result=00000000000000000000000000000000
exception=0

At time=                   30
operand1=00111111100011001100110011001101
operand2=01111111100000000000000000000000
result=01111111100000000000000000000000
exception=1
```

图表 4 加法器仿真

乘法的仿真结果如下图。可以看到输入涵盖了普通的输入输入，异常输入，异常输出、相同符号数相乘，不同符号数相乘等等情况。从测试结果看，乘法模块功能完全正确。

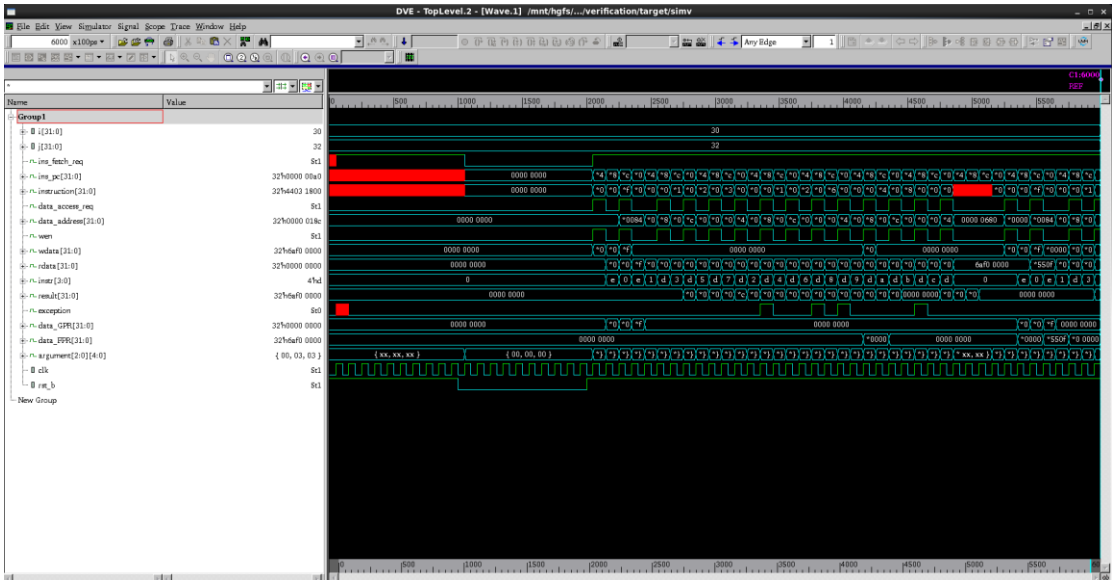
```
Compiler version G-2012.09; Runtime version G-2012.09;  Nov  4 17:13 2020
At time=                    0
operand1=01000010110010000000000000000000
operand2=00111111111111111111101111100111011
result=01000011010001111111001100110
exception=0

At time=                    5
operand1=01000010110010000000000000000000
operand2=10111111111111111111101111100111011
result=11000011010001111111001100110
exception=0

At time=                   10
operand1=11111111111111111111111111111111
operand2=10111111111111111111101111100111011
result=01111111100000000000000000000001
exception=1

At time=                   15
operand1=00000000000000000000000000000000
operand2=10111111111111111111101111100111011
result=10000000100000000000000000000000
exception=0

At time=                   20
operand1=00000000000000000000000000000000
operand2=11111111111111111111111111111111
result=01111111100000000000000000000001
exception=1

At time=                   25
operand1=01111111011111000000000000001011
operand2=01111111011111000000000000001011
result=01111111100000000000000000000000
exception=1

At time=                   30
operand1=00000000110011000000000000001011
operand2=00000000110011000000000000001011
result=01111111100000000000000000000000
exception=1
```

经测试，由加法和乘法衍生的几个模块都功能都基本正确，在此就不赘述。

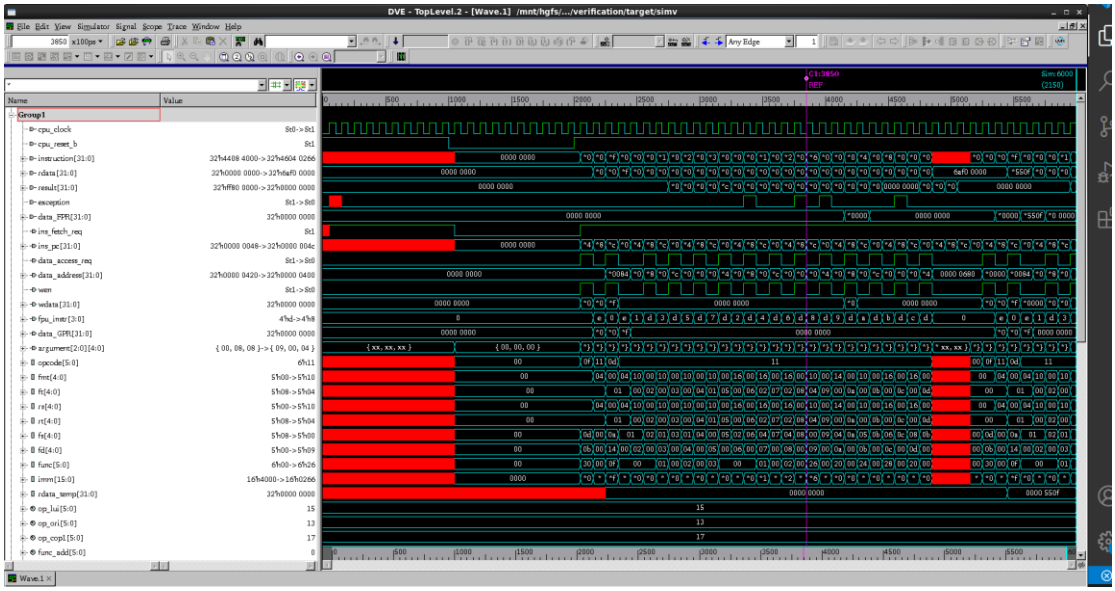
2、 整体仿真

为了测试整套流程，我设计了一套指令，主要是先用 lui 和 ori 指令在 GPR 中产生初始的非 0 值，通过 mtc1 写入 fpu 的 FPR 中，在 FPR 中进行相关运算，之后再通过 mfc1 将 FPR 中的计算结果全部写回 GPR。

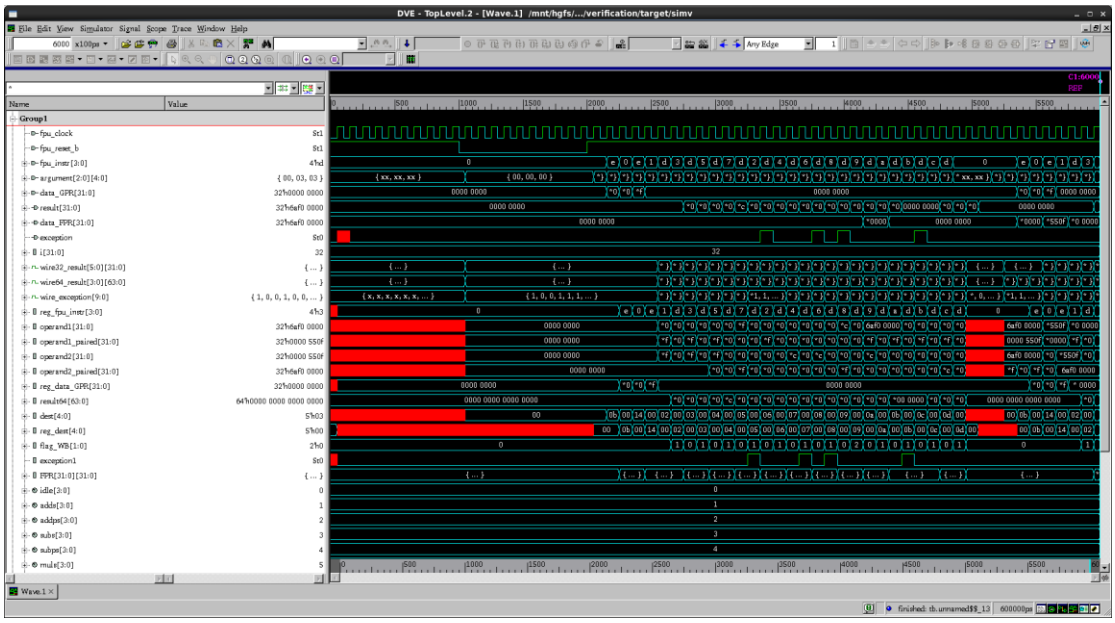
最终 testbench、processor 和 fpu 的波形图如下图所示。



图表 5 testbench 波形图

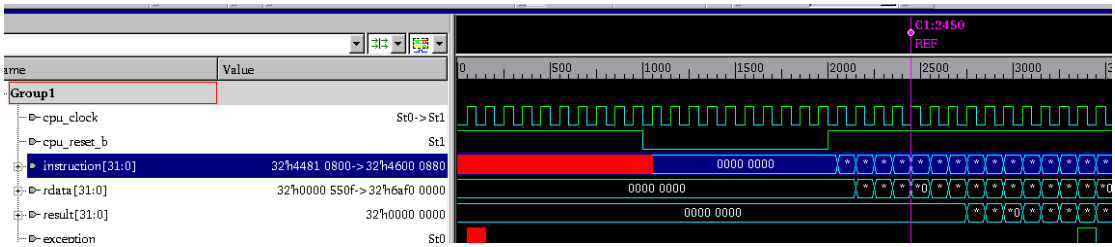


图表 6 processor 波形图

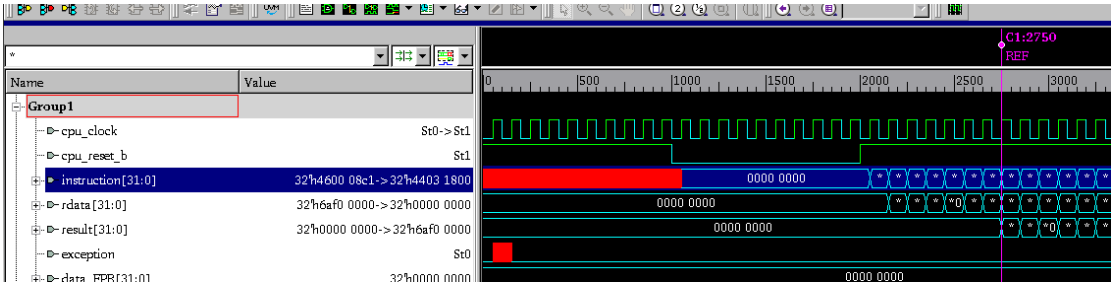


图表 6 fpu 波形图

以 processor 为例，其波形体现了几个关键的设计：在数据初始化以后，processor 在 245ns 收到第一条运算指令 adds，经过三个时钟周期，即 275ns 后，得到来自 result 的计算结果。之后所有的结算结果都是流水线式被 fpu 发送，被 processor 接收的。这就是协处理器的三级流水线设计。



图表 7-1

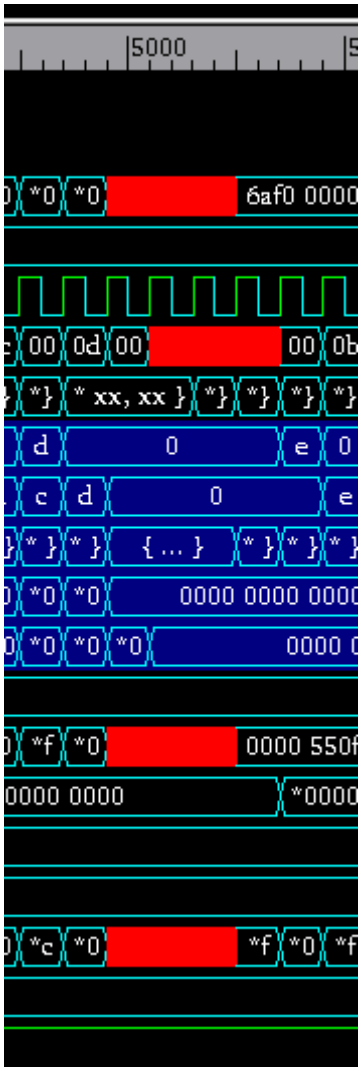


图表 7-2

之后由于执行 mfc1 传输指令，result 结果呈现周期新性的零。

在这套指令周期内，exception 异常出现了四个周期，分别出现在计算乘法、除法的时候。由于我的第一个操作数取得很大，所以会导致上溢情况的发生。这也是 processor 观察到异常的方式。

需要说明的是，在第 485ns 后出现了几个连续的 x，这是因为我的指令书不够 32 条，指令内存中还有些空余存储单元没有被赋值。这关系不大，因为当 pc 持续增加，所有的指令会再次从头按顺序执行一遍。

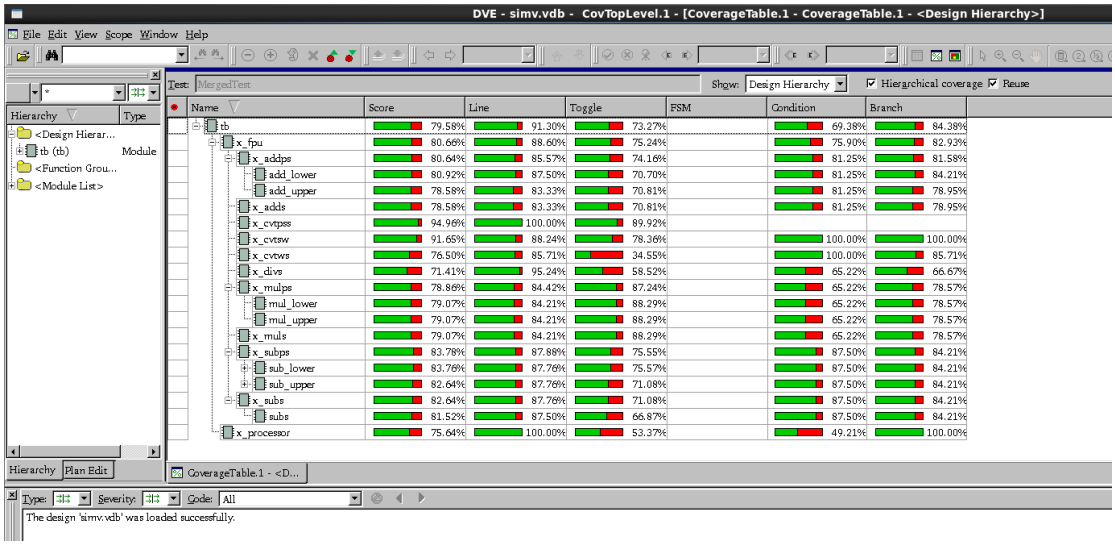


图表 8

几张波形图可以说明，本设计在运行当前这套指令的时候，时序和运算结果正确。这说明数据传输的指令的行为设计也是正确的。

3、 覆盖率

测试覆盖率得到如下结果：



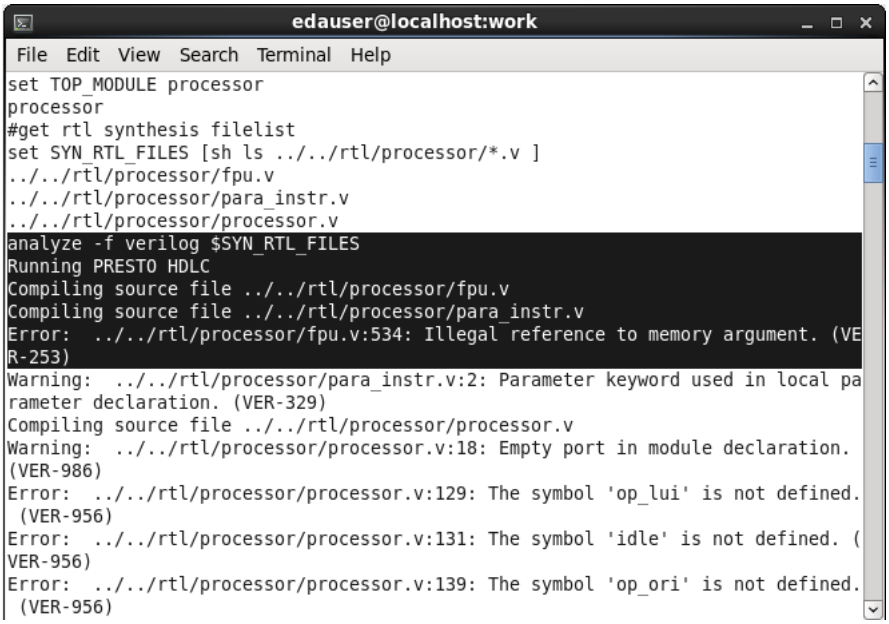
图表 9

由于目前的指令较少，所以覆盖率不是特别高。但是主要的情况已经涉及到了。

五、 综合

我没有直接用 tb.v 综合，而是直接写了一个顶层模块 module top，其中包含了 fpu、memory 和 processor，没有输入激励。我修改综合脚本中的相关配置文件，将 module top 作为顶层的综合模块。

在用项目已有的代码进行综合命令后，发现出现这样的问题：



报错显示对 memory，不能进行综合。我详细看了 tb.v 里面有 mem_temp [integer]这种写法，这是一种动态范围的写法，Verilog 是不支持这种的综合

的，要用 System Verilog 综合。故我修改了综合脚本 syn_script.tcl 中的命令，将一处“verilog”改成“sverilog”，解决了这个错误。

```
67
68 analyze -f sverilog $SYN_RTL_FILES
69 elaborate $TOP_MODULE -lib work
```

图表 10-1 改为 sverilog

修改完后出现了新的错误。

```
164 Error:  ../../rtl/processor/processor.v:41: Net 'data_access_req' or
    a directly connected net is driven by more than one source, and not
    all drivers are three-state. (ELAB-366)
165 Error:  ../../rtl/processor/processor.v:41: Net 'data_address[6]' or
    a directly connected net is driven by more than one source, and not
    all drivers are three-state. (ELAB-366)
166 Error:  ../../rtl/processor/processor.v:41: Net 'data_address[5]' or
    a directly connected net is driven by more than one source, and not
    all drivers are three-state. (ELAB-366)
167 Error:  ../../rtl/processor/processor.v:41: Net 'data_address[4]' or
    a directly connected net is driven by more than one source, and not
    all drivers are three-state. (ELAB-366)
168 Error:  ../../rtl/processor/processor.v:41: Net 'data_address[3]' or
    a directly connected net is driven by more than one source, and not
    all drivers are three-state. (ELAB-366)
169 Error:  ../../rtl/processor/processor.v:41: Net 'data_address[2]' or
    a directly connected net is driven by more than one source, and not
    all drivers are three-state. (ELAB-366)
```

图表 11 多次赋值的错误

我重新检查所有的代码，发现在初始化的时候，我在 always 块中，当 reset 触发时，所有输出值进行赋值，但是是不对的，应该把初始化阶段写在 initial 块中。虽然 initial 本身是不可综合的（或者说会被忽略），但却可以在可综合模块中对存储器加载初始化文件，这属于一种可综合的行为。

调整代码后，发现出现 fatal error。

```
   ELAPSED            WORST NEG TOTAL NEG  DESIGN
    TIME      AREA      SLACK     SLACK   RULE COST         ENDPOINT
  --------- --------- --------- --------- --------- -------------------------
    0:01:11  171401.3      7.48     200.9     476.3
    0:01:11  171401.3      7.48     200.9     476.3
    0:01:12  176609.3      7.14     197.6     454.4
    0:01:13  190669.4      7.14     186.2     122.3
    0:01:14  203061.6      6.48     126.2      39.1
    0:01:25  205324.3      1.43      45.1       0.3
    0:01:26  205394.4      1.53      48.0       0.3
    0:01:26  205394.4      1.53      48.0       0.3
    0:01:26  205405.4      1.52      47.8       0.3
    0:01:26  205405.4      1.52      47.8       0.3
    0:01:26  205405.4      1.52      47.8       0.3

Abort at 951


The tool has just encountered a fatal error:

If you encountered this fatal error when using the most recent
Synopsys release, submit this stack trace and a test case that
reproduces the problem to the Synopsys Support Center by using
Enter A Call at http://solvnet.synopsys.com/EnterACall.

* For information about the latest software releases, go to the Synopsys
  SolvNet Release Library at http://solvnet.synopsys.com/ReleaseLibrary.

* For information about required Operating System patches, go to
  http://www.synopsys.com/support


Fatal: Internal system error, cannot recover.

Release = 'G-2012.06-SP2'  Architecture = 'linux'  Program = 'dc_shell'
Exec = '/tools/synopsys/syn_vG-2012.06-SP2/linux/syn/bin/common_shell_exec'

'284564231 284567148 284702571 284988170 284988334 183620273 183816479 183903139 183967108 183975466 184022478 16160
2022 161623744 161822411 160381743 160417508 160499348 155377915 154033211 283781521 283902357 286528373 286536842 2
86543093 283809712 283843229 283902357 286528373 286536842 286543093 283892211 283906010 149609332 149612317 1496412
91 149641677 149643052 134629115 134614971 6454582'
```

图表 12 综合时的 fatal error

经过查阅资料后发现，要在 dc_shell-t 后面加-64bit，问题解决，终于综合完毕。

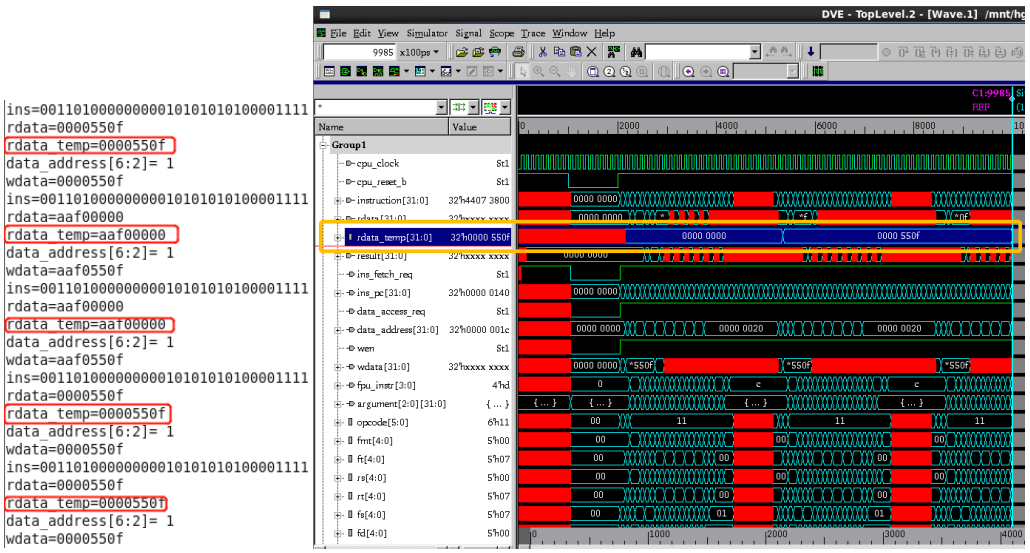
图表 13 go 脚本加上-64bit

六、 调试技巧

1、 vpd 文件已经生成，可以代替 vcd 文件查看波形。

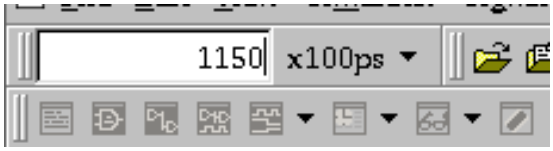
2、 tb.v 里面有 mem_temp [integer]这种写法，其实是动态分配空间的写法（dynamic range），所以能根据命令的条数来决定 mem_temp 的大小。这是 SystemV erilog 才有的写法。

3、 遇到的问题：在 vcs 进行编译仿真的时候，用 display 输出某个变量的值，输出有一个值；但在波形图中，又看不到这样值变。原因：其实是某一个变量在一个周期内被赋值多次，所以用 display 能输出不同的值；而在波形图中只会显示这个时钟周期内变化最后的值，所以两者不对应。



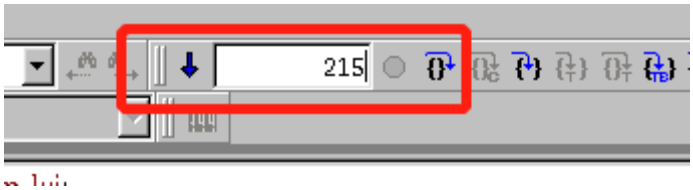
图表 14

4、 vcs 支持从某个时间节点开始的仿真。比如我想在 115ns 开始进行仿真，我可以在仿真运行之前，在空格处填 115 然后开始仿真。这样就可以实现在不同时间段的仿真。



图表 15

还可以设置仿真的时间步长。比如如果想仿真 215 个时间单位，可以在这里填 215，然后启动仿真。



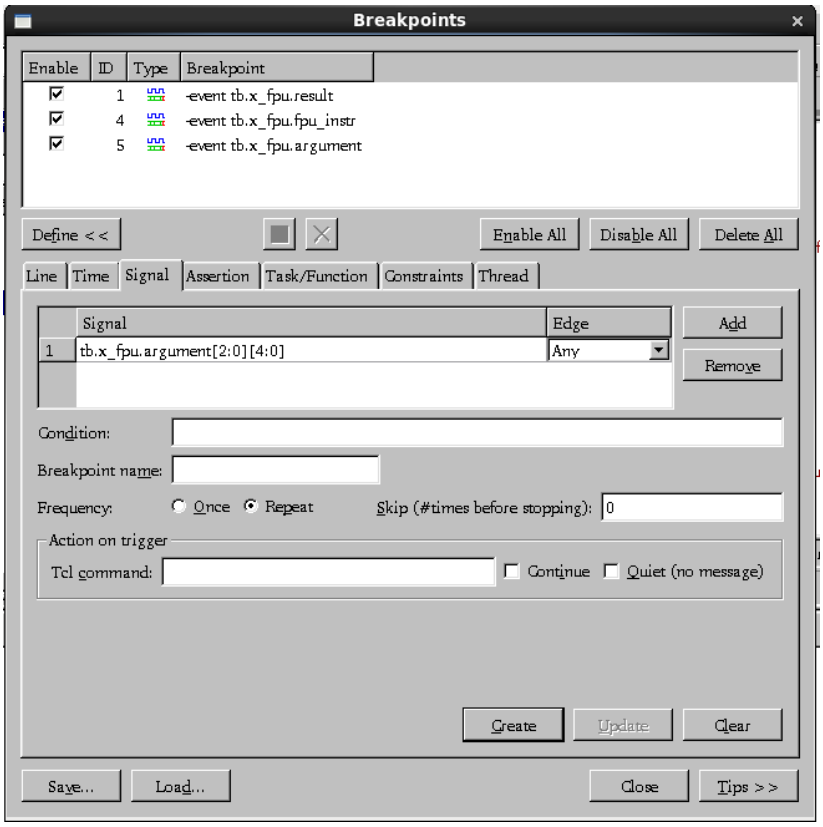
图表 16

5、 vcs 有时会报错,vcs Error-[TCF-CETE] Cannot evaluate the expression，这是由于 Verilog 不允许在取下标的时候使用变量。使用非常量下标的电路设计无法实现的。比如：

if(shift<=23) frac=pos_int[msb-1:0]; //不行 msb 是 reg 类型

是不被允许的。如果我们要取和某一位有关的具体的几位，我们可以采用先移位（移位的位数可以是变量），再取下标的方法。这样就能取到某些位的值了。

6、vcs 支持对多个变量设断点进行调试。这样就能运行的时候看到不同的变量的变化。



图表 17

七、 结论

我对项目要求的 16 条指令进行相关建模,写出了所有计算或数据传输模型。根据我对项目的理解，我将 processor 和 fpu 独立开来，processor 作为主处理

器，fpu 作为协处理器，其中协处理器是用三级流水线；我还修改了 memory，读和写的通道分开；用 Python 书写简单 MIPS 编译器；设计指令集进行验证。经验证，大部分模块功能和时序正确，基本完成原定的设计目标，并且完成了综合。存在的不足有：还没有用更多的指令去验证不同的情况。