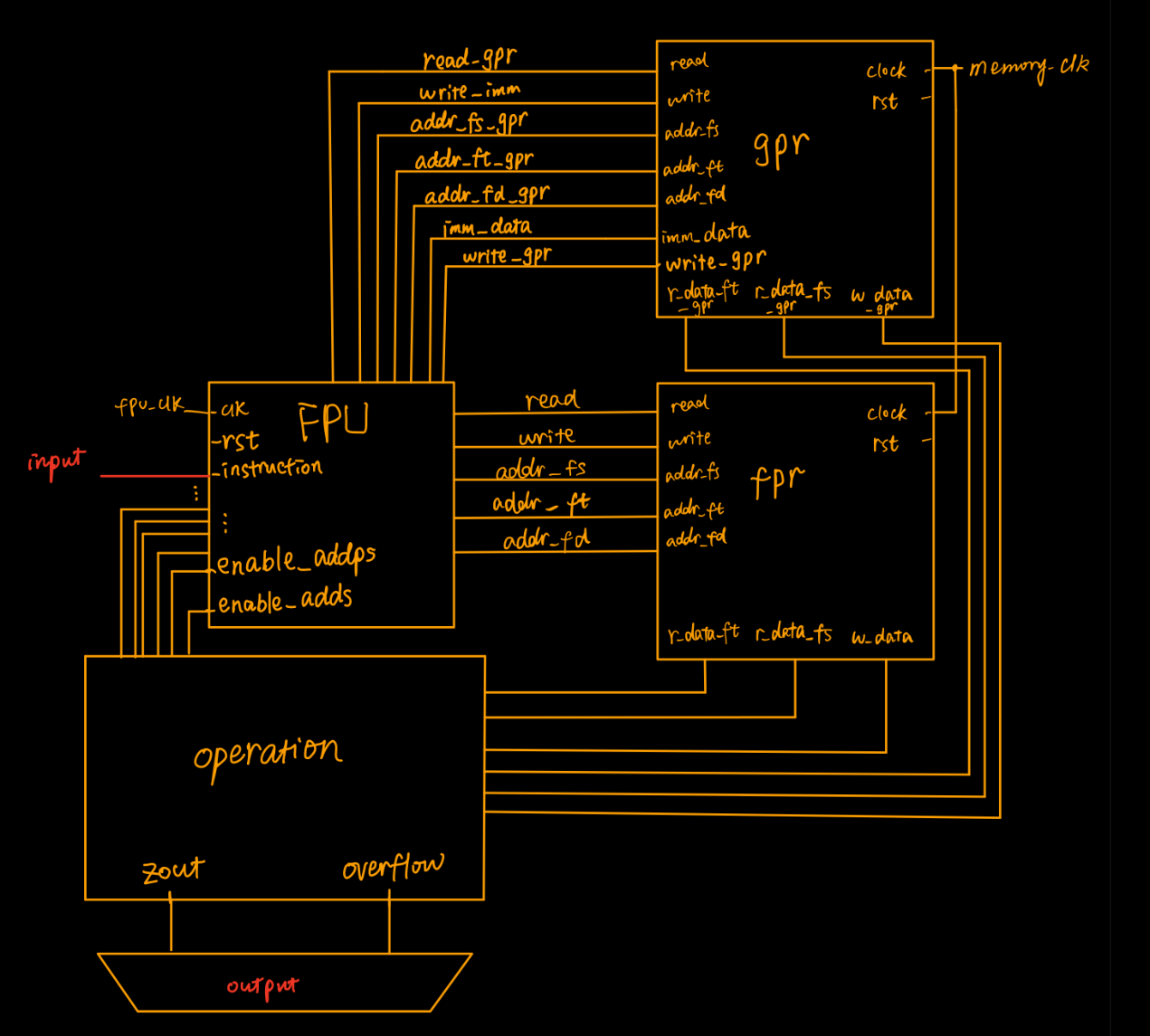
**浮点运算单元设计文档**

钱煜 3180103948

一、设计框图

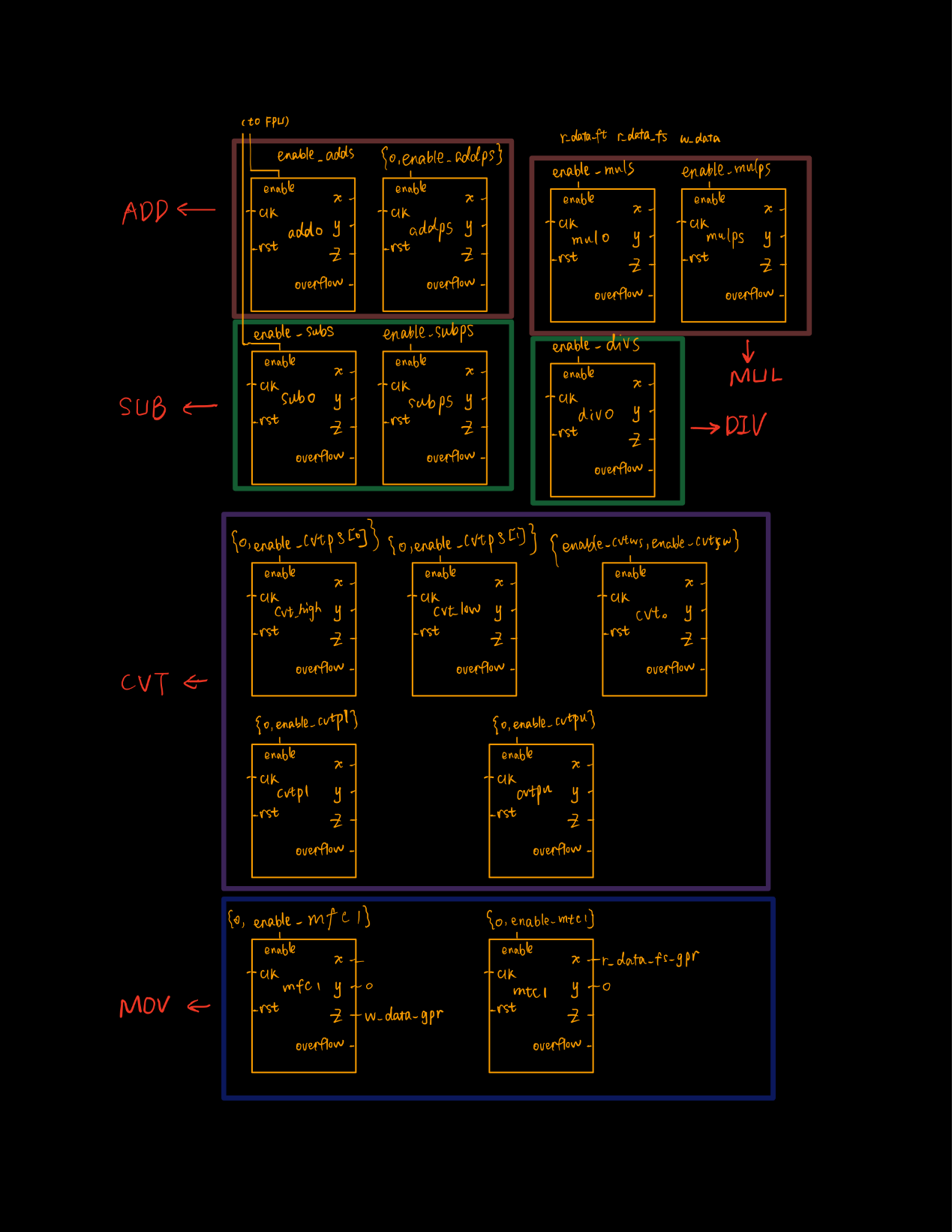
1.1整体框图（部分线因美观效果暂时省去）



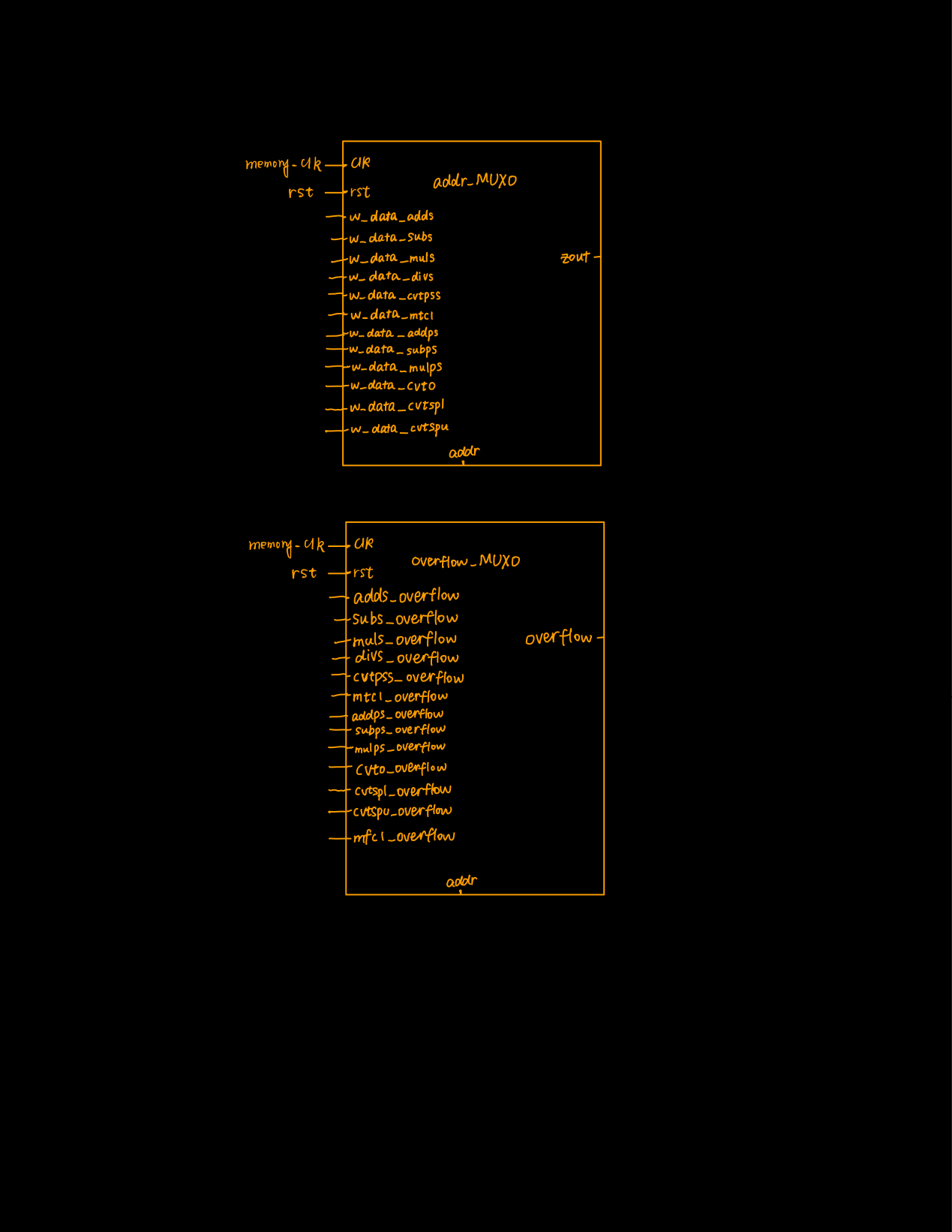
运算顺序为：

FPU中译码——gpr/fpr中存取数据（如有需要）——operation中运算——gpr/fpr中存取数据，并输出最终结果

（注：operation部分将在之后给出框图）

1.2 operation框图

上图为operation板块的一部分，未标注的clk都连接至clk，未标注的rst都连接至rst，x，y分别连接至r\_data\_ft, r\_data\_fs，enable连接到FPU的端口，z连接到addr\_MUX的输入端



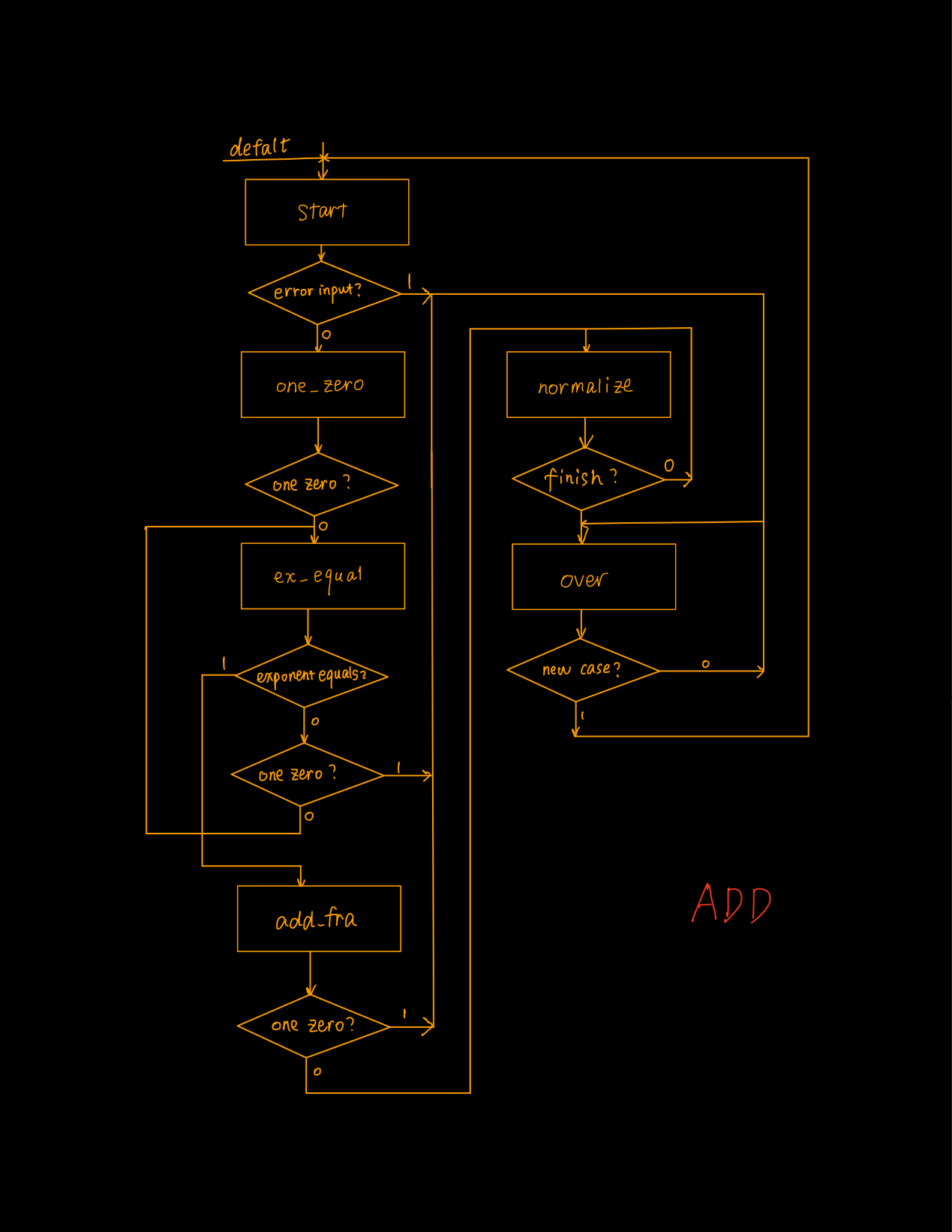
上图为operation板块的后半部分，addr为FPU译码后输出的选择地址，zout和overflow为最终的输出

1.3 operation子模块框图

（注：为了防止跑飞，定义了当除了工作状态外的其他状态出现时，default ：nextstate = START）

1.3.1 ADD

1.3.1.1 ADD状态图



1.3.1.2 ADD 运行过程说明

（1）START

将输入的两个加数x，y拷贝到cmp\_x，cmp\_y，用于之后比对是否要进行新的一轮运算；

将x和y拆分成xe，ye，xm，ym。其中xe，ye为指数（考虑到可能溢出，故先不减去BIAS），zm，ym为带1的小数部分（即在本身的0.xxxxx上变成1.xxxxx）；

判断输入是否规范，具体为：指数是否为8‘d255，以及指数为零且底数不为零，若是，则令overflow为2’b11（代表数值不符合规范），并直接跳转到OVER，反之，进入ONE\_ZERO；

（2）ONE\_ZERO

判断是否至少有一个加数为0（即指数和底数都为0），若是，则可以将另一个加数直接赋值给z作为输出，同时跳转到OVER，反之，进入EX\_EQUAL；

（3）EX\_EQUAL

判断xe和ye是否相等，若相等，说明可以直接底数相加，进入ADD\_FRA；

若不相等，将小的指数+1，对应底数右移一位（使得数值不变），若此时底数为零，由于0的任意次幂都为0，则只需将大的指数对应的数赋值给输出即可，同时跳转到OVER；

若没有以上情况，则说明两个加数还没有变换成指数相等且底数不为0的情况，继续跳转到EX\_EQUAL自我循环；

（4）ADD\_FRA

判断输入使能enable是2‘b01还是2’b10来判断是加法操作还是减法操作；

如果是加法操作，则判断xm和ym的大小（注意，此时指数已经相等，比较底数实际上就是比较绝对值大小），来进一步判断z的符号位，同时将大数减/加小数，得到最终的小数值。若结果为0，则直接跳到OVER，即两个数相等。（注意：这里实际上会出现相加时溢出或者相减时数位降低的情况，这也将在下个状态中解决该问题）；

（5）NORMALIZE

判断zm是否超过24位（上个状态的遗留问题），若是，则进行移位和指数自增，跳转到OVER（因为只需一次操作）；

判断zm是否低于23位（上个状态的遗留问题）若是，则进行移位和指数自检，跳转到NORMALIZE自我循环（可能多次操作），直到位数正常为止，进入OVER；

（6）OVER

判断是否上溢（指数为8‘d255）或者非规格（指数为0而底数不为0），以此决定输出的overflow值；

判断cmp\_x，cmp\_y与x，y是否一致，若一致，则跳转到OVER不断自我循环，反之，跳转到START进行新的一轮运算。

1.3.2 SUB

1.3.2.1 SUB状态图

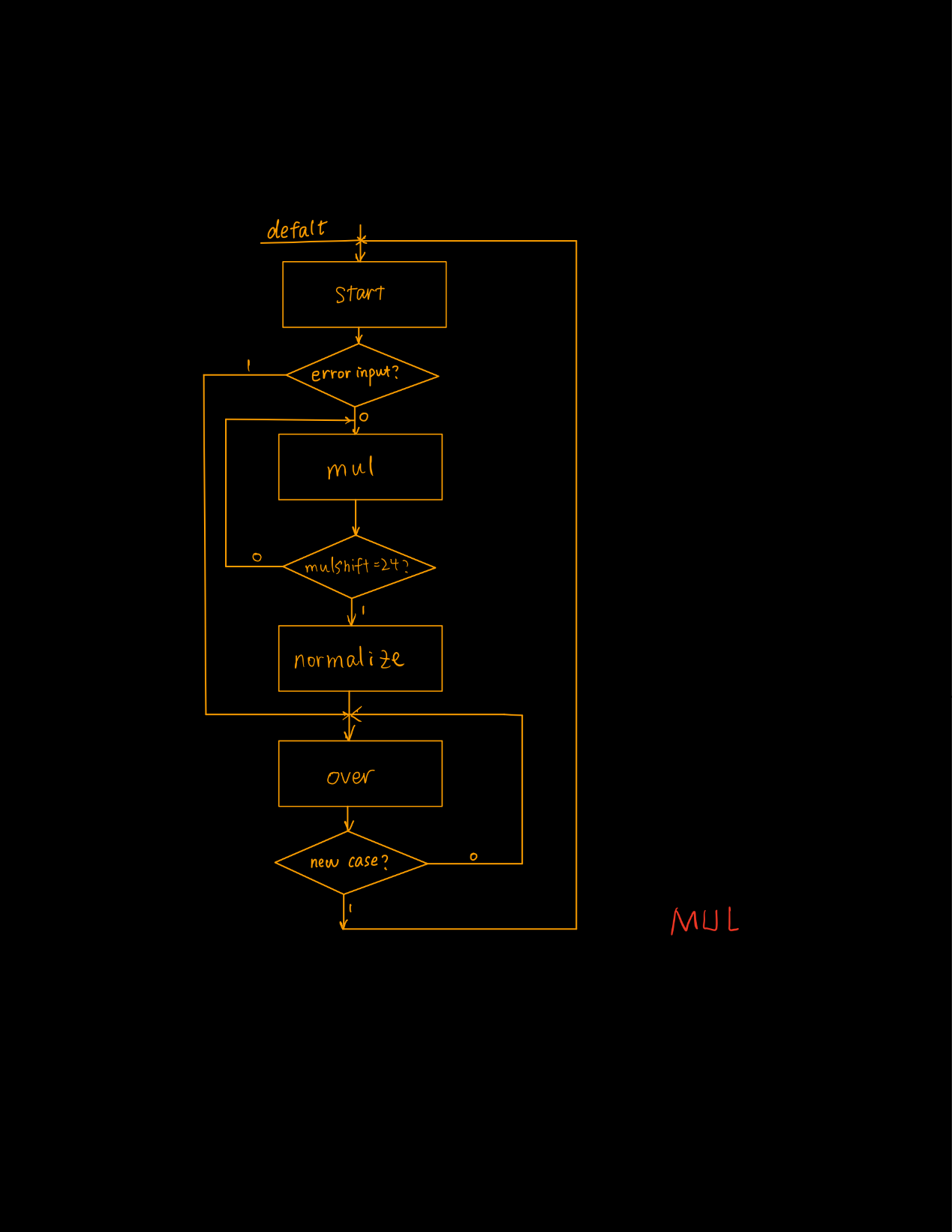
参见1.3.1.1 ADD状态图

1.3.2.2 SUB运行过程说明

参见1.3.1.2 ADD运行过程说明

1.3.3MUL

1.3.3.1 MUL 状态图



1.3.3.2 MUL 运行过程说明

（1）START

将输入的两个乘数x，y拷贝到cmp\_x，cmp\_y，用于之后比对是否要进行新的一轮运算；

将x和y拆分成xe，ye，xm，ym。其中xe，ye为指数（考虑到可能溢出，故先不减去BIAS），xm，ym为带1的小数部分（即在本身的0.xxxxx上变成1.xxxxx）；

创建50位的xm1，ym1，为了后续乘法需要的移位准备（考虑了worstcase的移位情况，故选择了50位）；

判断输入是否规范，具体为：指数是否为8‘d255，以及指数为零且底数不为零，若是，则令overflow为2’b11（代表数值不符合规范），并直接跳转到OVER，反之，进入ONE\_ZERO；

（2）MUL

根据输入两个乘数的符号位，异或操作得出输出z的符号位；

固定xm1，移动ym1，每次往高位移动一位，再根据当前xm1的数位决定是否加ym1（本质即为：将乘法看作加权的加法）。移位次数用mulshift标记，当移位25次后，得到了两个乘数小数部分的乘积zm，进入NORMALIZE；

（3）NORMALIZE

通过判断zm的最高位，来确定是否需要底数指数对应移位来保证底数为1开头的小数，为了保证不产生溢出，用shift\_1和shift\_2暂时表示指数部分需要＋和-的位数，并进入OVER；

（4）OVER

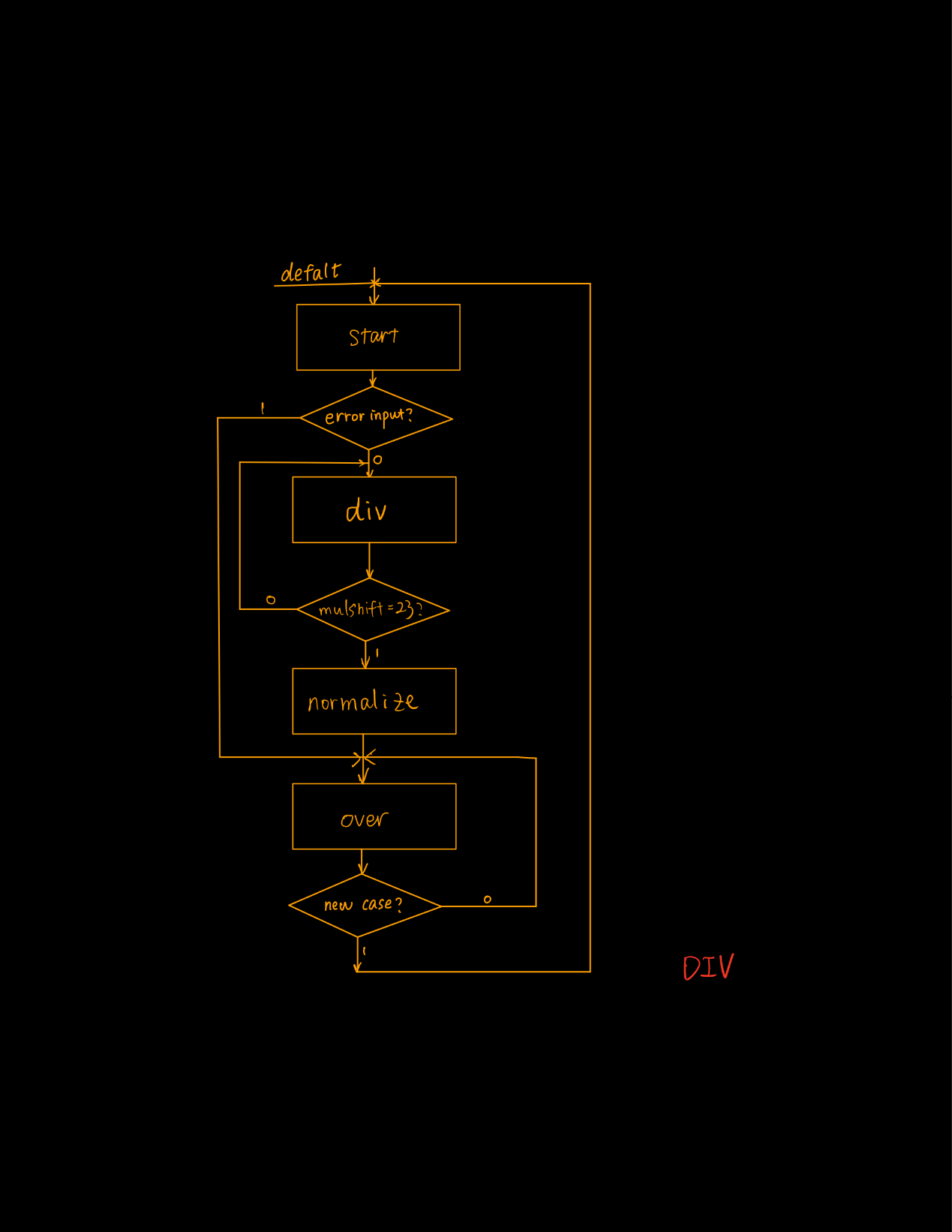
用e\_temp来存储指数部分，考虑到溢出情况，设定其数位为10位；

根据其与8‘d255和shift\_2的关系，检测边界情况，注意，此时底数已经完全符合规范，故指数溢出，即为整个数的溢出。

若没有溢出，则正常规格化：根据其指数是否为8‘d255或者指数为零而底数不为零，分别置数overflow，若一切正常，则置数overflow为0；

判断是否enable为1且至少一个被乘数出现变化，若是，则跳转到START进行新的一轮运算，反之，继续在OVER自动循环。

1.3.4. DIV

1.3.4.1 DIV 框图

1.3.4.2 DIV 运行过程说明

（1）START

将输入的两个运算数x，y拷贝到cmp\_x，cmp\_y，用于之后比对是否要进行新的一轮运算；

将x和y拆分成xe，ye，xm，ym。其中xe，ye为指数（考虑到可能溢出，故先不减去BIAS），xm，ym为带1的小数部分（即在本身的0.xxxxx上变成1.xxxxx）；

创建50位的xm1，ym1，为了后续除法需要的移位准备（考虑了worstcase的移位情况，故选择了50位）；

判断输入是否规范，具体为：指数是否为8‘d255，以及指数为零且底数不为零，若是，则令overflow为2’b11（代表数值不符合规范），并直接跳转到OVER，反之，进入ONE\_ZERO；

（2）DIV

固定ym1，不断移位xm1，判断每一次ym1对应数位的值，来决定是否要相减（本质上是将xm1乘2的23次幂后求商，因为这样得到的商为23位），并用mulshift进行计数；

若未到次数，则进入DIV自我循环，直到mulshift为23时，循环结束，进入NORMALIZE；

（3）NORMALIZE

根据zm的最高有效数位，判断指数需要移动的位数，为防止直接运算溢出，存放在shift\_1和shift\_2中，进入OVER；

（4）OVER

判断e\_temp与8‘d255和shift\_2的大小关系（因为自身数位足够，不必考虑e\_temp的上溢），若出现溢出，则分别置数ze为8‘d255或者8’d0，否则，对输出置数；

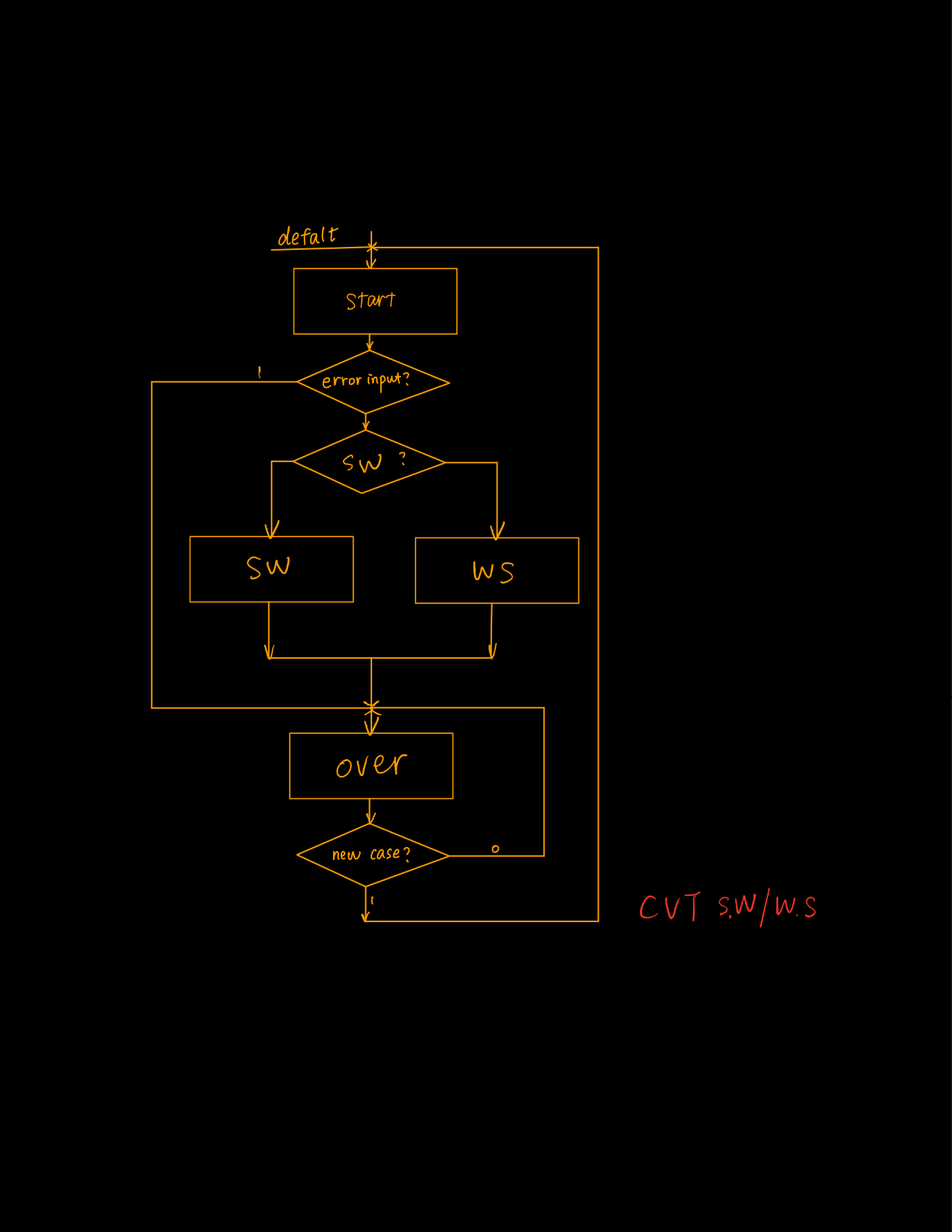
检查输出是否上溢或者趋近于零的非规约，并置数相应overflow；

判断enable是否为1且至少有一个输入改变，若是，则跳转到START开始新的除法运算，反之，跳转到OVER。

1.3.5 CVT

由于其他的CVT操作可以借助之前的基础架构完成（后文会详细讲述），这里只阐述相对复杂的SW/WS部分

1.3.5.1 CVT (SW/WS) 框图



1.3.5.2 CVT (SW/WS) 运行流程说明 （注：切换了判断error的时间，防止将word误认为是float而导致问题）

（1）START

将输入的两个运算数x，y拷贝到cmp\_x，cmp\_y，用于之后比对是否要进行新的一轮运算；

通过判断enable的数值，跳转到SW或者WS；

（2）SW

根据待转化数x的最高有效位，取x的部分位数作为单精度数的小数位，或者取x余下的所有位数并添加末尾的0构成单精度数的小数位，同时根据所需移位的数目（包括左移或者右移）决定指数的大小，最后置数符号位；

（3）WS

将y拆分成ye，ym。其中ye为指数（考虑到可能溢出，故先不减去BIAS），ym为带1的小数部分（即在本身的0.xxxxx上变成1.xxxxx）；

判断输入是否规范，具体为：指数是否为8‘d255，以及指数为零且底数不为零，若是，则令overflow为2’b11（代表数值不符合规范），并直接跳转到OVER；

判断指数是否上/下溢，若是，置数overflow和z，并跳转到OVER；

若没有溢出，将指数部分减去偏置后用于小数部分的移位，若超出则用0补全，不足则舍去多余小数部分；

跳转到OVER；

（4）OVER

判断enable使能信号，若结果为单精度浮点数，检测其是否上溢或非规格趋于零并作出相应overflow的置数，反之不作操作（作没有意义的操作以匹配else）；

判断enable是否为真且至少有一个输入改变，若是，则跳转到START开始新的转换运算，反之，跳转到OVER。

二、行为描述

本部分主要讲述了14条指令的行为描述，相对着重于行为级的执行次序，而非最底层运算。指令运算过程中，绝大部分使用单精度加法器作为核心运算部件，若出现转移数值，则令对应加法器的某个输入为0即可。

2.1 ADD.S

（1）DECODE

创建cmp\_instrction，为之后判断是否要执行新的指令做准备；

将addr\_MUX置为4‘b0001，使得该加法器的输出端口与fpr的端口连接开通（可以写入）；

跳转到ADD\_READ；

（2）ADD\_READ

将read信号置为1，输入指令中ft和fs寄存器的地址，fpu访问地址，将对应寄存器中的数保存至x和y（加法器的两个加数输入端口）；

跳转到ADD\_CAL；

（3）ADD\_CAL

将enable\_adds置为2‘b01，使得对应加法器开始运算；

跳转到ADD\_WRITE；

（4）ADD\_WRITE

将read置为0，停止读书，将write置为2‘b01，并输出fd的地址，将运算得到的z写入到fpu对应地址的目的寄存器中；

（5）OVER

将所有使能信号和输出信号置零，比较cmp\_instruction与当前指令是否一致，若不一致，则跳转到DECODE进行新的运算，反之，在OVER中循环；（之后不再重复）

2.2 ADD.PS

（1）DECODE

基本操作与2.1一致，将addr\_MUX置为4‘b0010；

跳转到ADDPS\_READ1；

（2）ADDPS\_READ1

同ADD\_READ类似；

跳转到ADDPS\_CAL

（3）ADDPS\_CAL

同ADD\_CAL类似；

跳转到ADDPS\_READ2；

（4）ADDPS\_READ2

将ft和fs的地址+1后进行读取；

跳转到ADDPS\_CAL2；

（5）ADDPS\_CAL2

同ADD\_CAL类似；

跳转到ADDPS\_WRITE2；

（6）ADDPS\_WRITE2

写入操作，注意将fd目的寄存器地址+1；

跳转到OVER

（7）OVER

略

2.3 SUB.S

2.4 SUB.PS

以上两条指令与2.1，2.2类似，注意写入使能信号的第二位需要为1，以表示减法运算；

2.5 MUL.S

2.6 MUL.PS

2.7 DIV.S

与2.1，2.2类似，基本思想为

译码（改变输出和overflow数据选择器的地址）——读数——运算——写数——读数（注意地址改变）（如有必要）——运算（如有必要）——写数（注意地址改变）（如有必要）——OVER

2.8 CVT.PS.S

与2.2类似，基本思想为

译码（改变输出和overflow数据选择器的地址）——读数——类型转换——写数——读数（注意地址改变）——类型转换——写数（注意地址改变）——OVER

2.9 CVT.S.W

2.10 CVT.W.S

2.11 CVT.S.PL

2.12 CVT.S.PU

与2.1类似，基本思想为

译码（改变输出和overflow数据选择器的地址）——读数（如有必要，注意地址变换）——类型转换——写数（如有必要，注意地址变换）——OVER

2.13 LUI

2.14 ORI

2.15 MFC1

2.16 MTC1

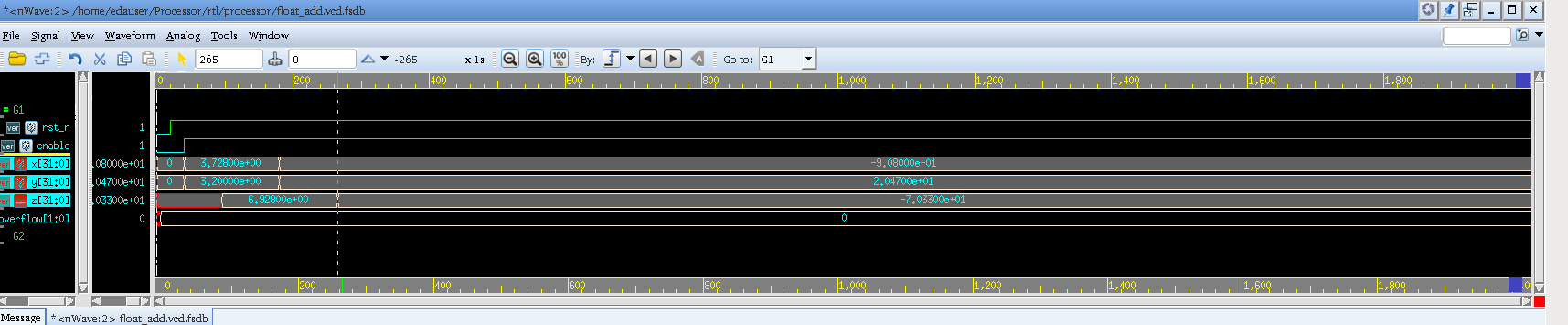
以上操作均涉及对gpr的存取，基本思想与2.1类似，不再赘述。

注意使能信号不一定是read&write，可能是read\_gpr，write\_gpr，write\_imm等

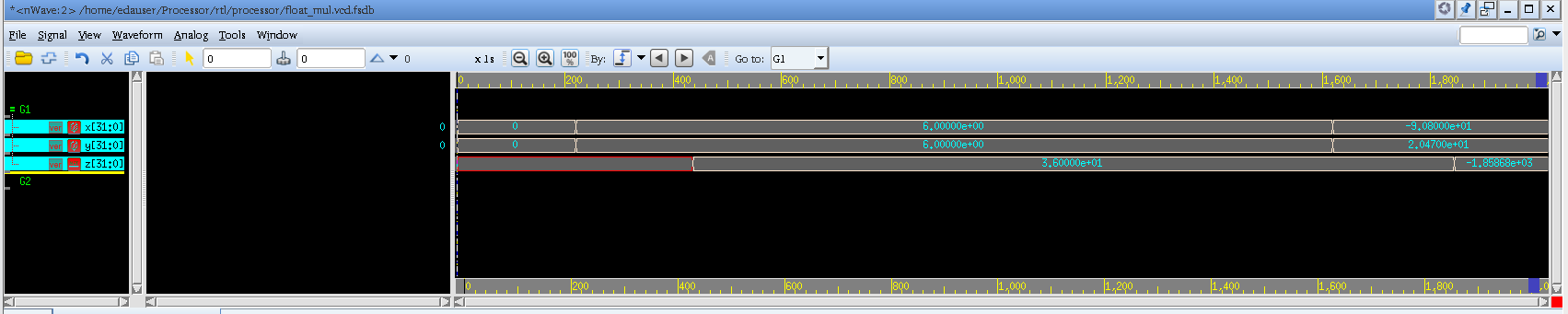
三、测试样例（部分）

3.1 单个模型（单精度）

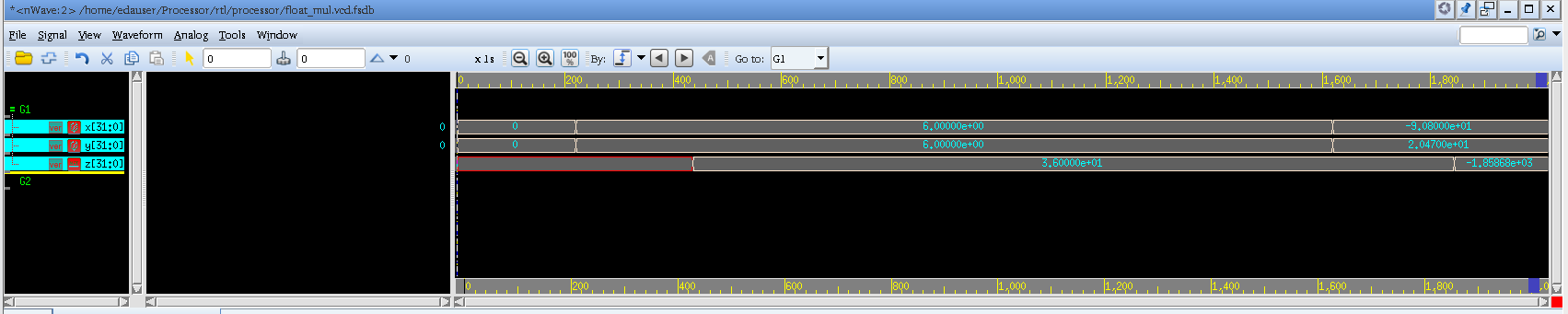
3.1.1 加法器



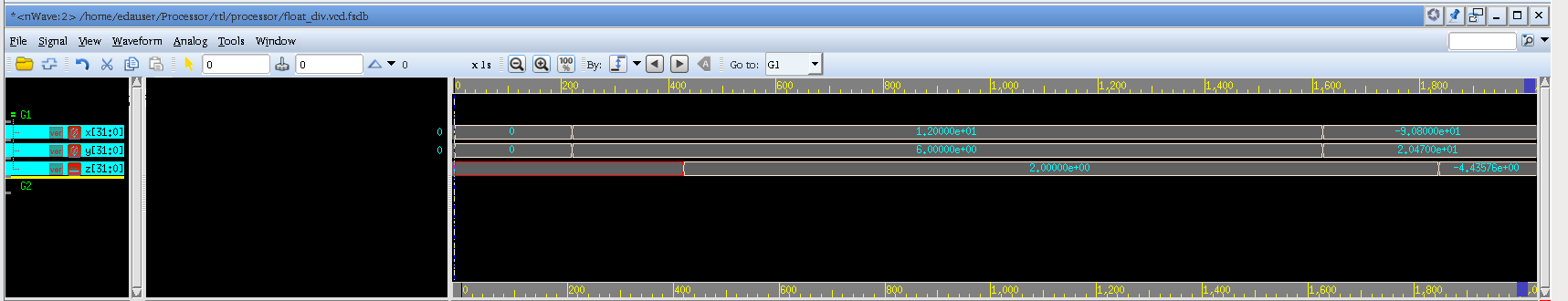
3.1.2 减法器



3.1.3 乘法器

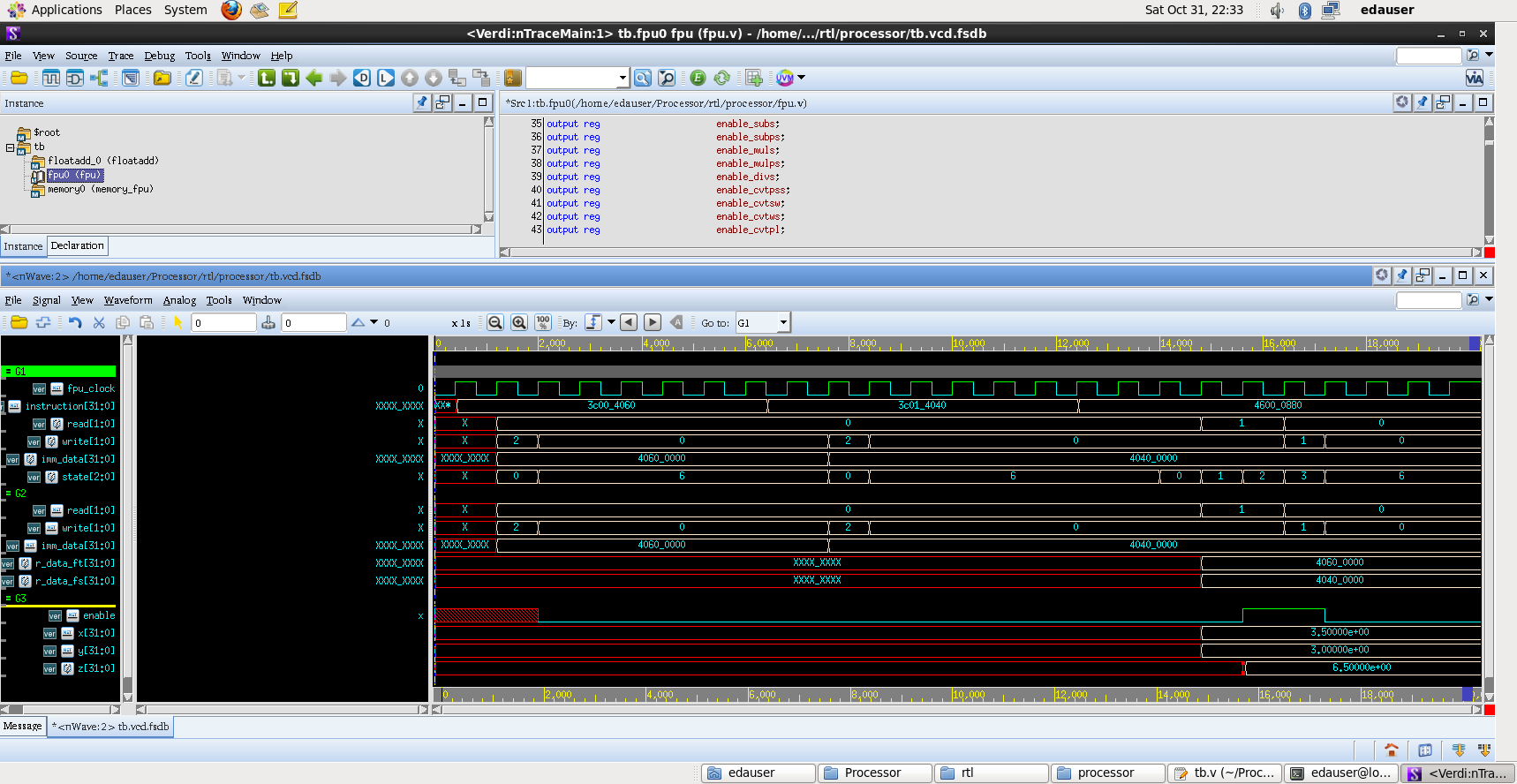


3.1.4 除法器

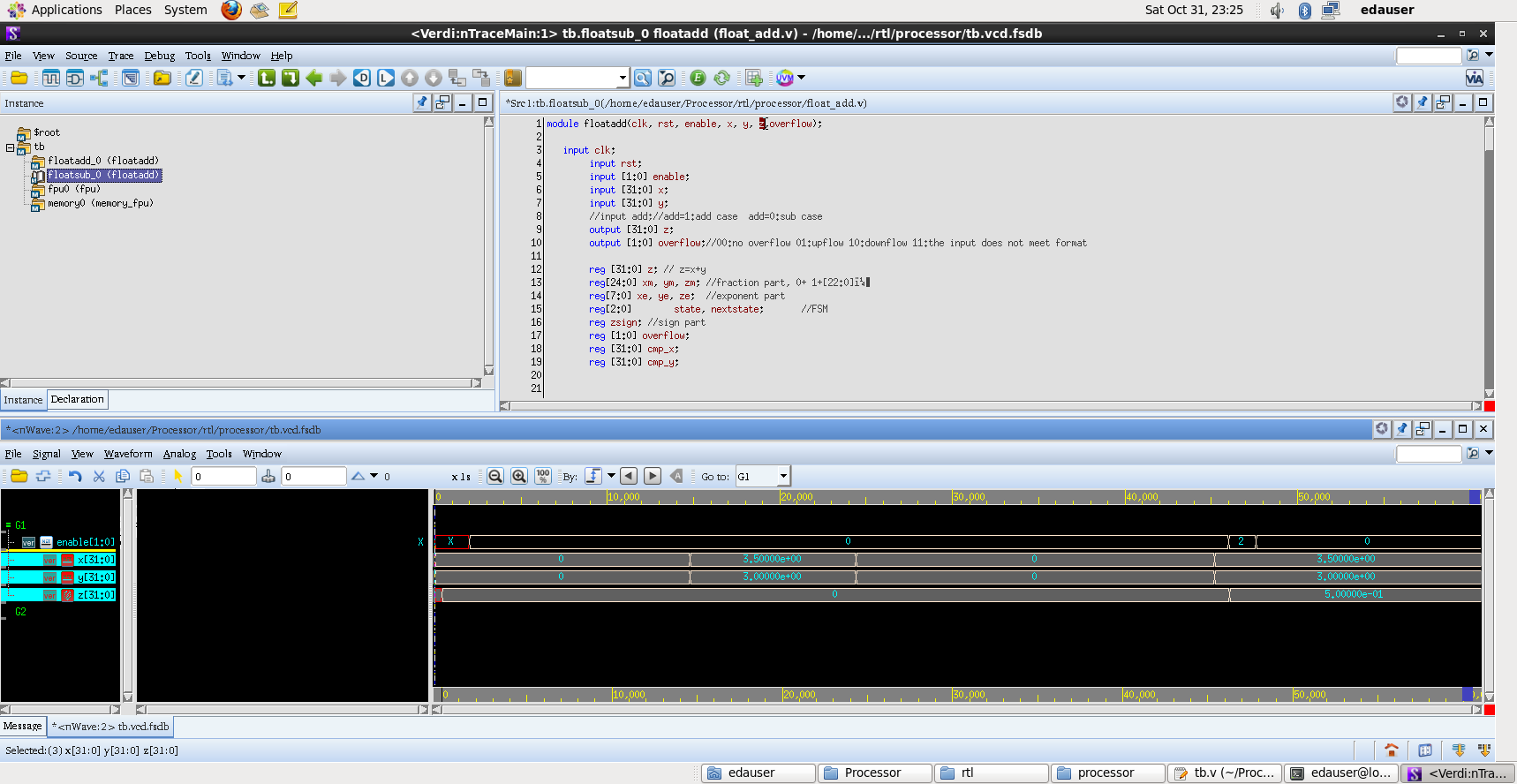


3.2 指令

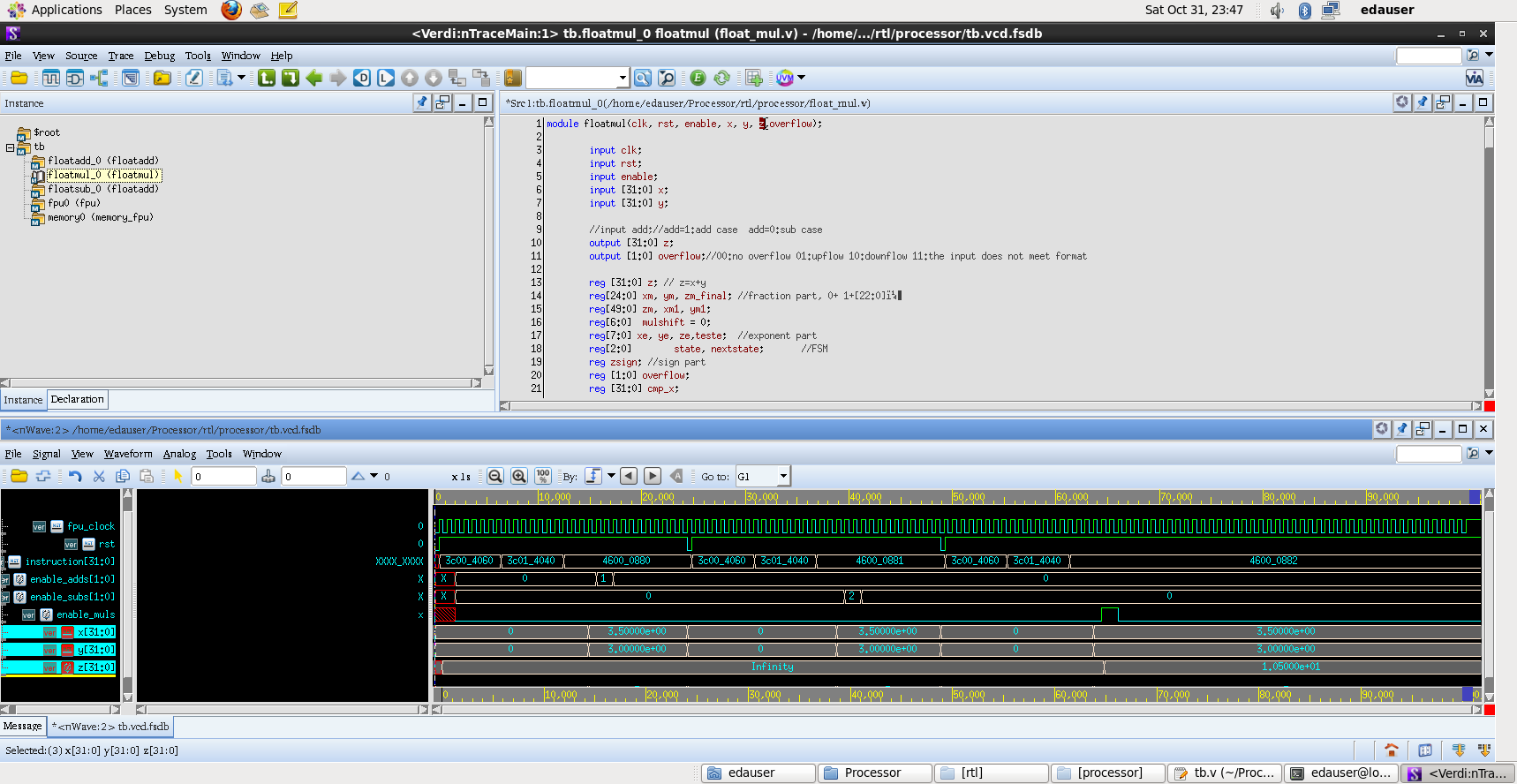
3.2.1 ADD.S （3.5+3=6.5）



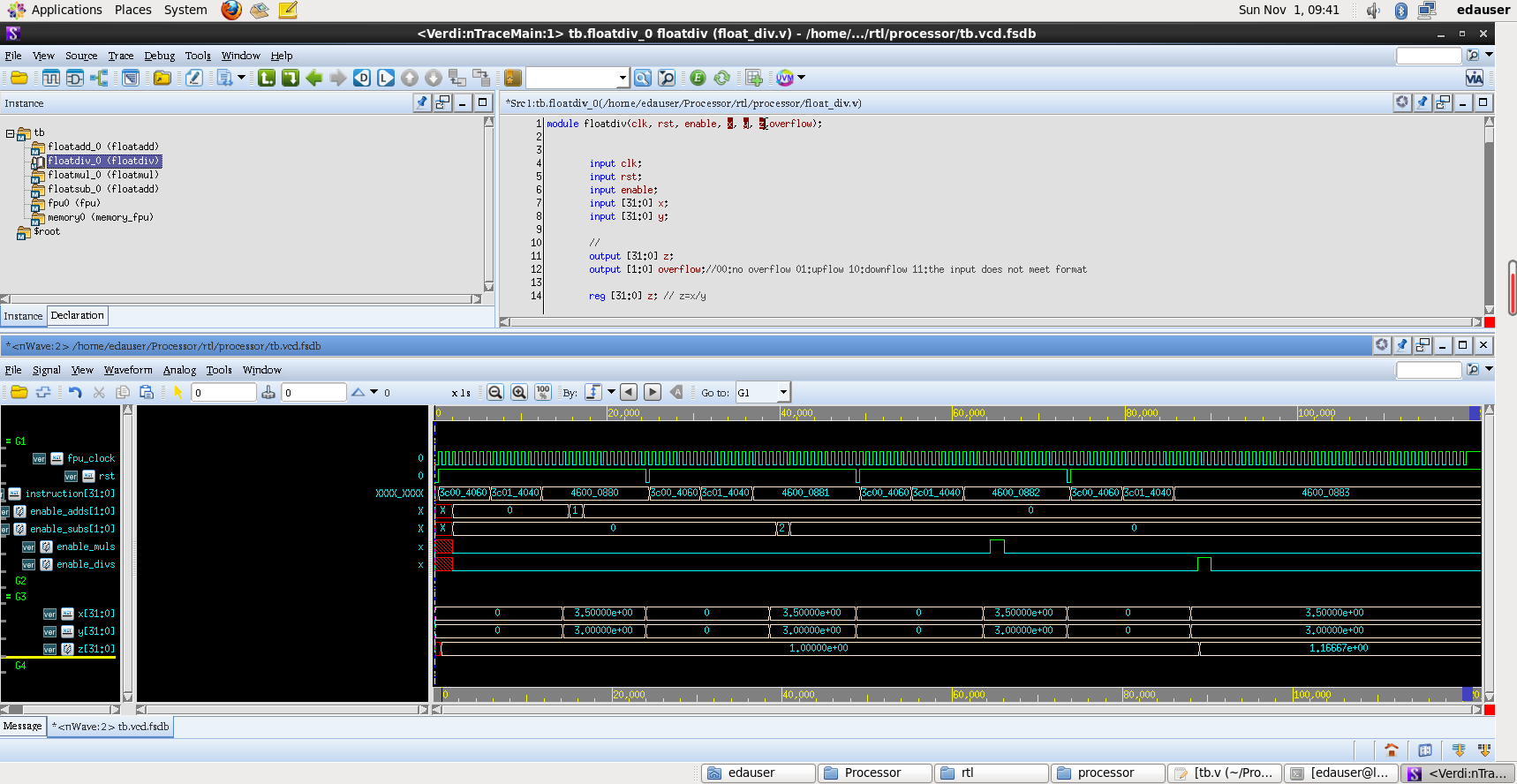
3.2.2 SUB.S （3.5-3=0.5）



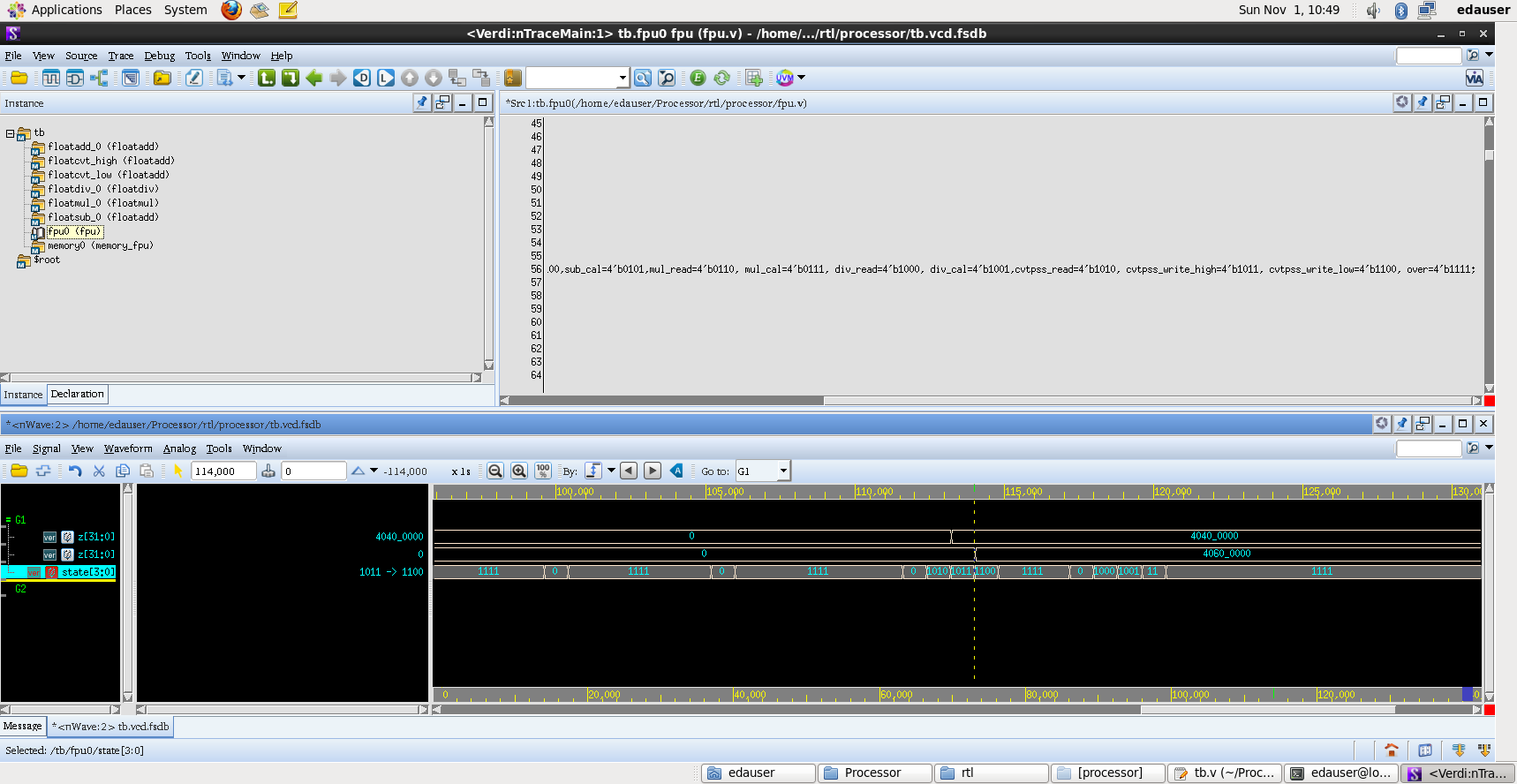
3.2.3 MUL.S （3.5\*3=10.5）



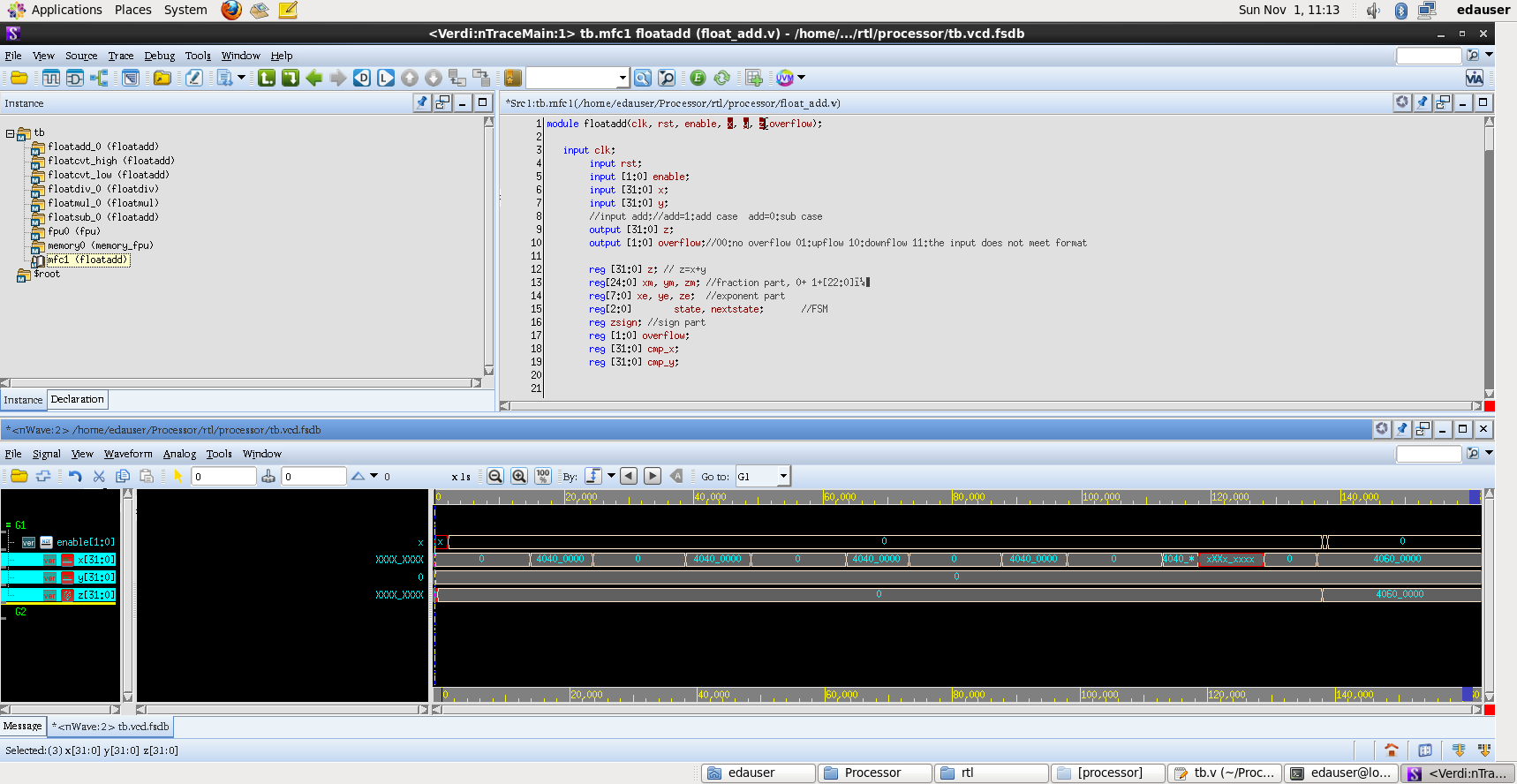
3.2.4 DIV.S （3.5/3=1.6667）



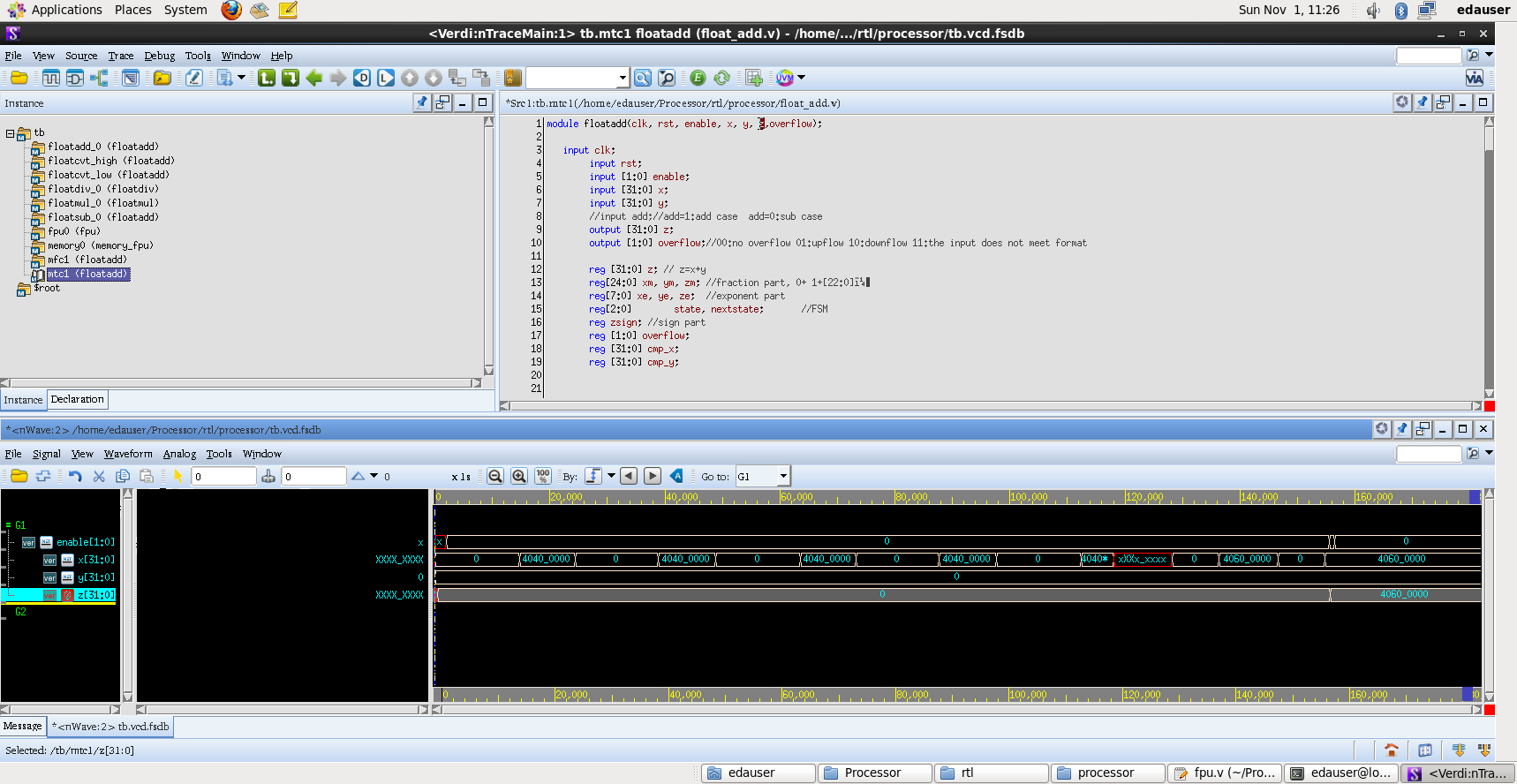
3.2.5 CVT.PSS（两个z代表两个转换后的数3、3.5）



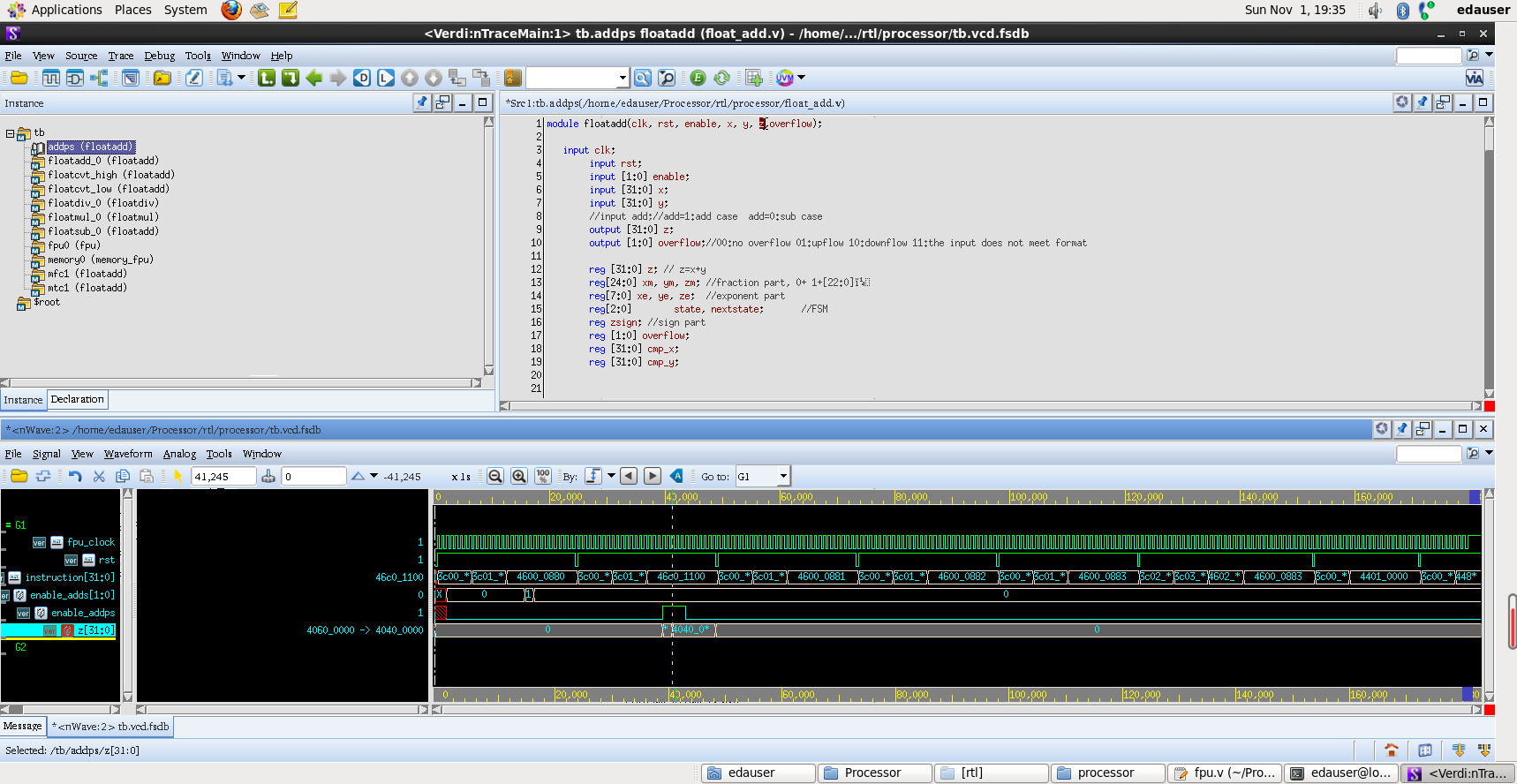
3.2.6 MFC1



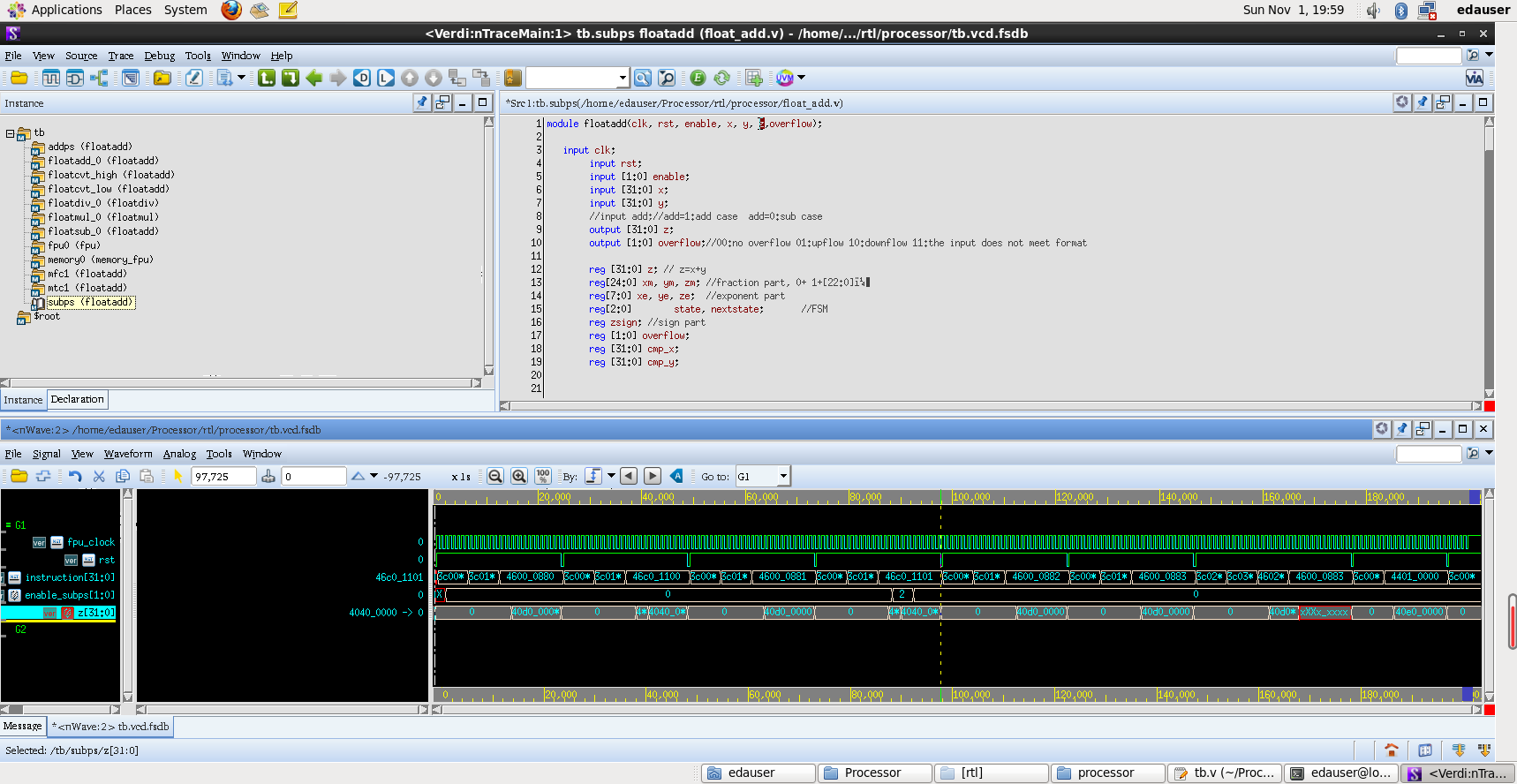
3.2.7 MTC1



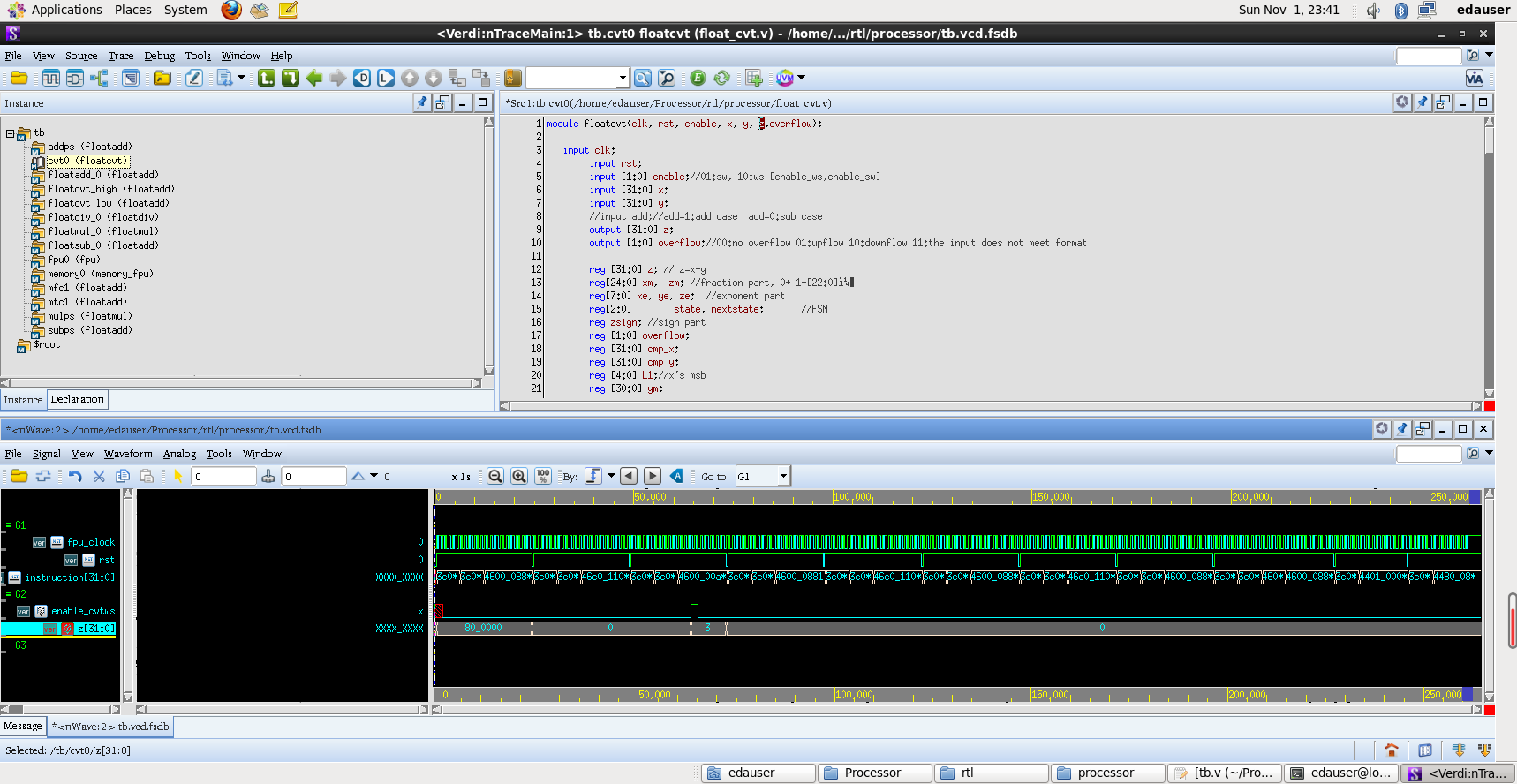
3.2.8 ADD.PS (Z的时间点显示从第一对的加法输出到第二对的加法输出)



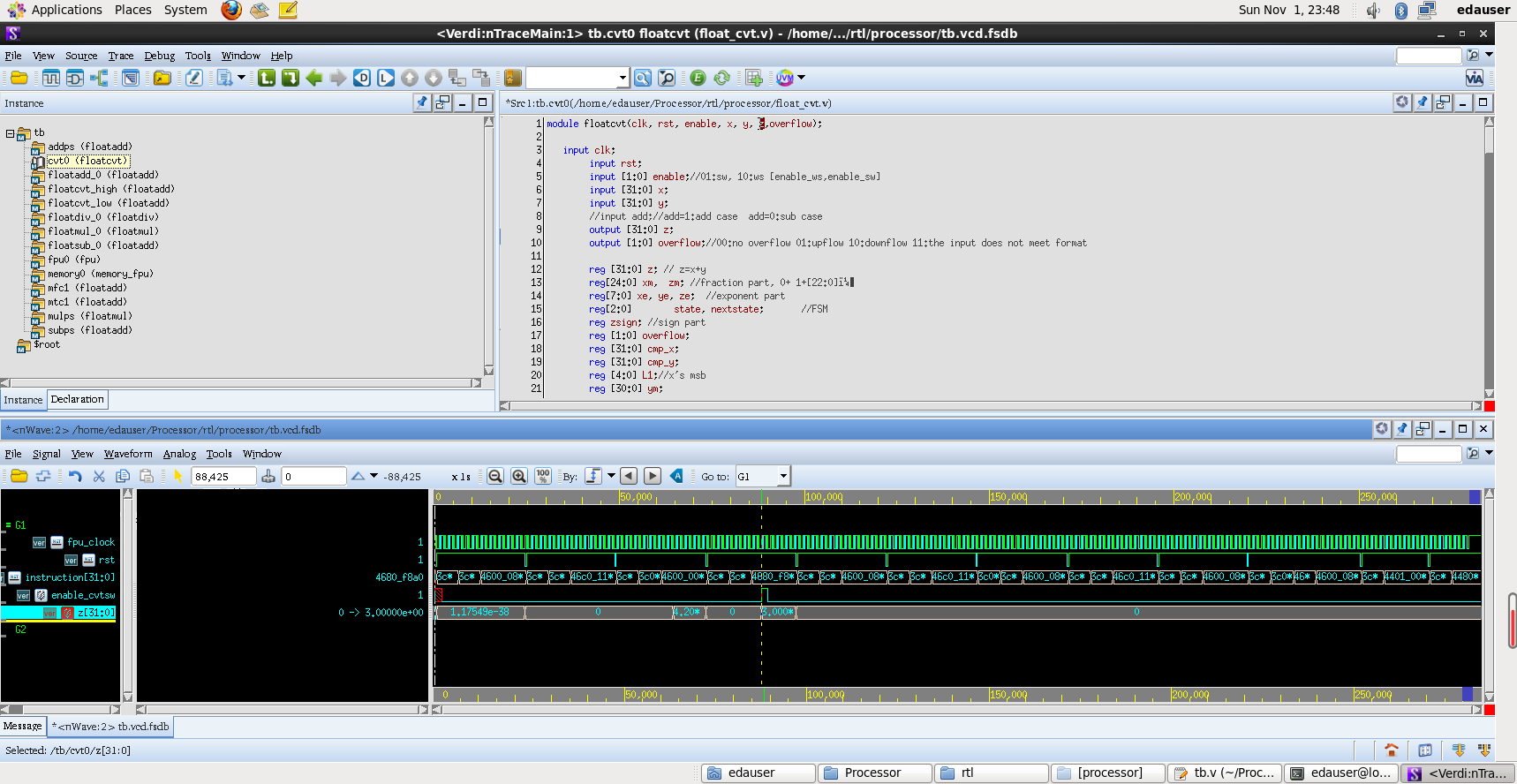
3.2.9 SUB.PS（同样是z的变化）



3.2.10 CVT.W.S（3的转换，下同）

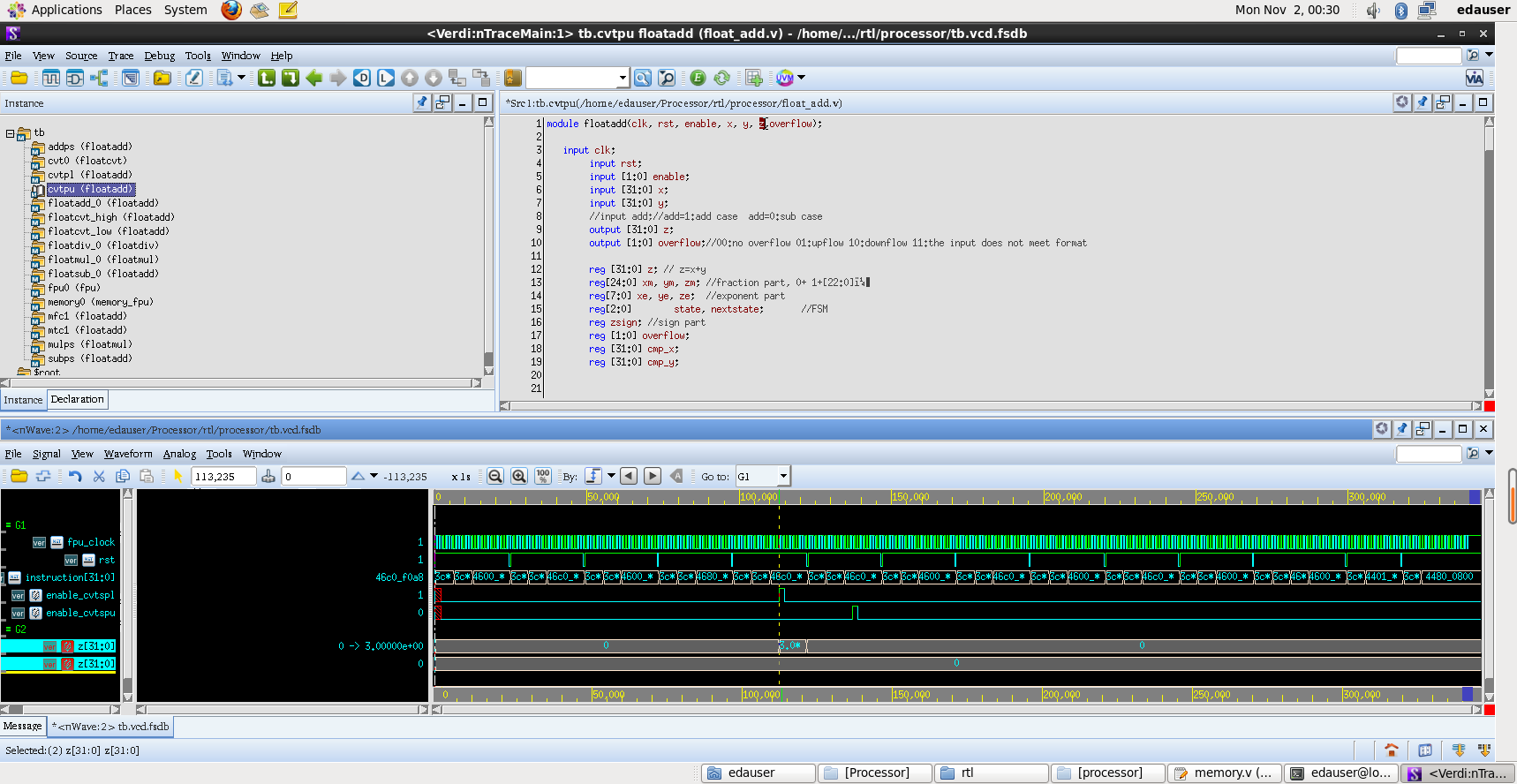


3.2.11 CVT.S.W（3的转换，同上）



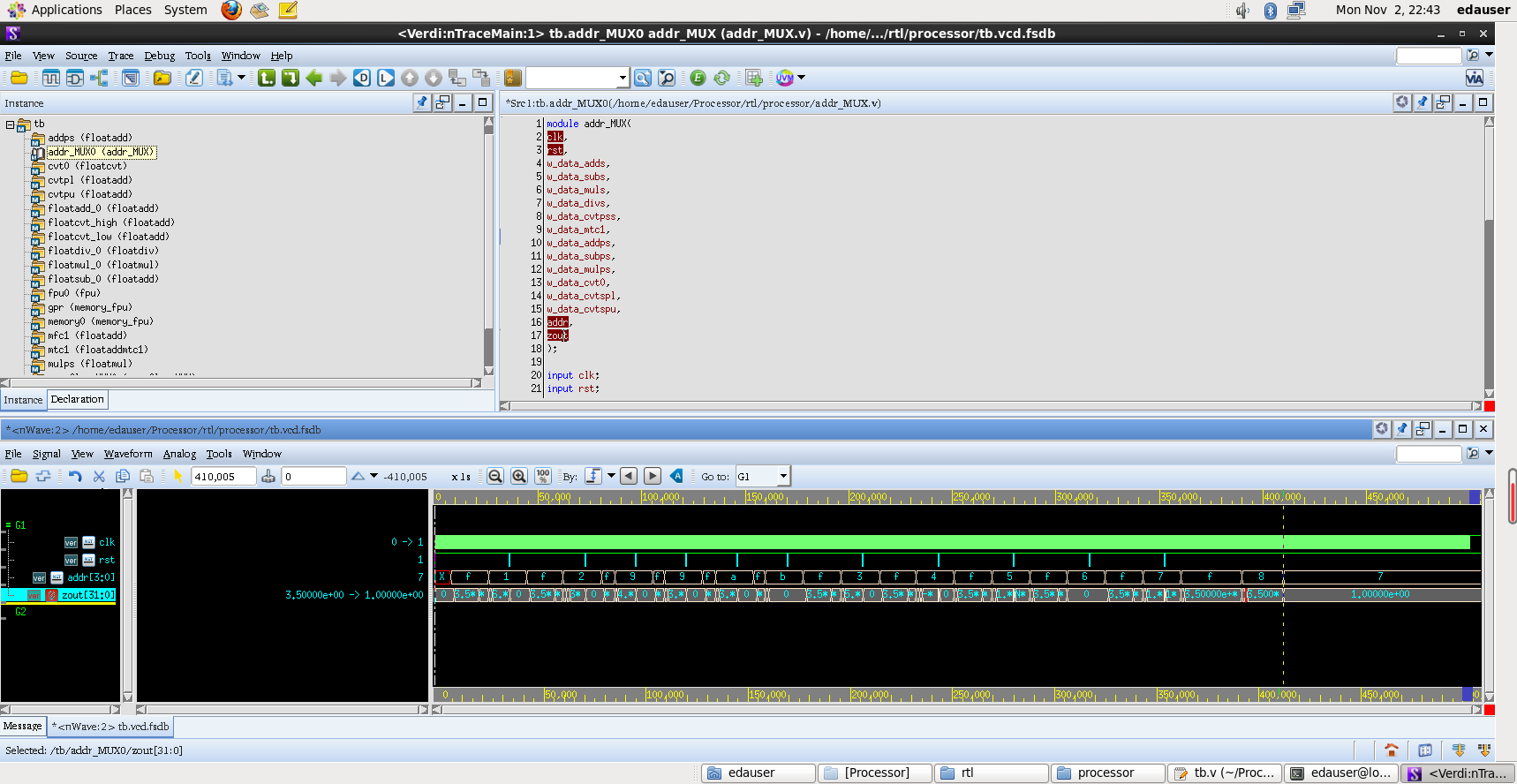
3.2.12 CVT.S.PL

3.2.13 CVT.S.PU



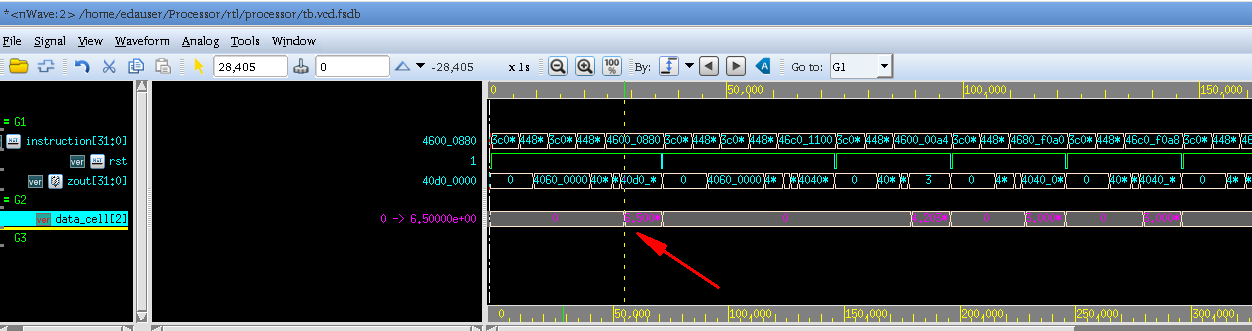
3.3 综合样例

在tb.v中，按顺序输入了12个周期的指令，遍历了所有操作（前12个周期除了如下所示外，使用了lui和mtc1，最后一个周期多使用了ori和mfc1，且经过寄存器波形查看，均正确）

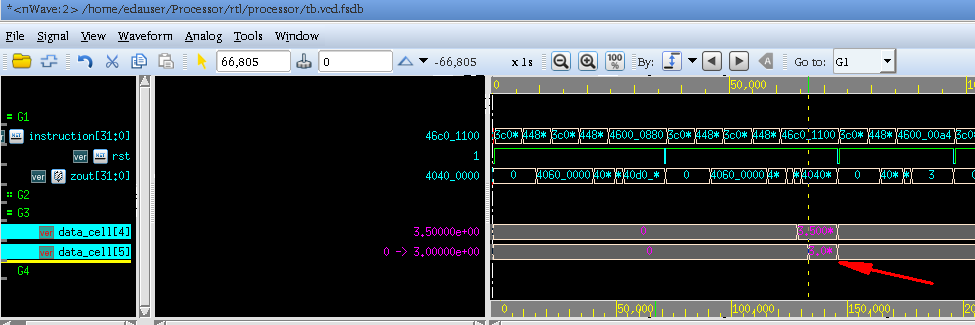


以下分别详细说明

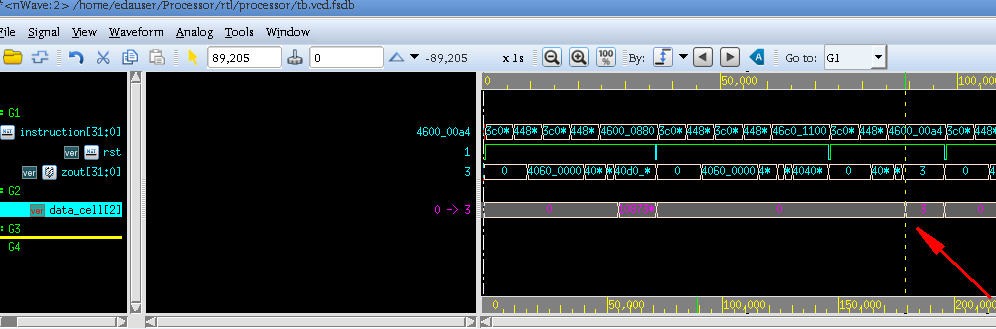
3.3.1 ADD.S



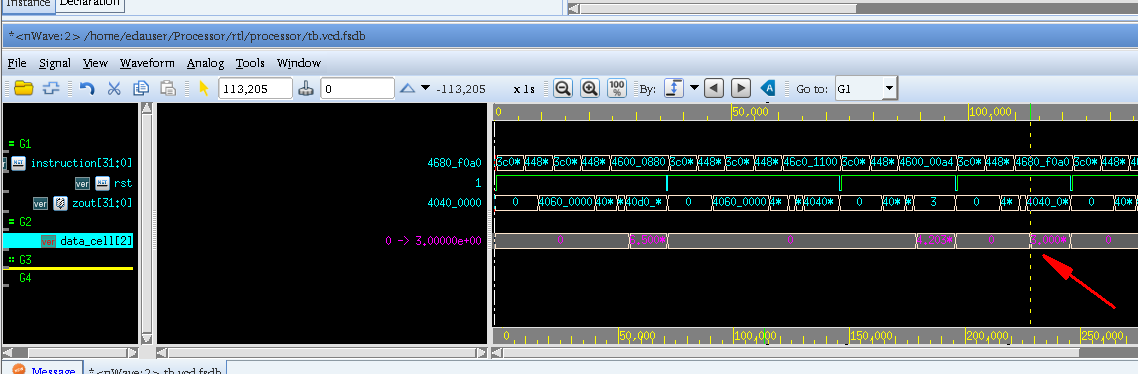
3.3.2 ADD.PS



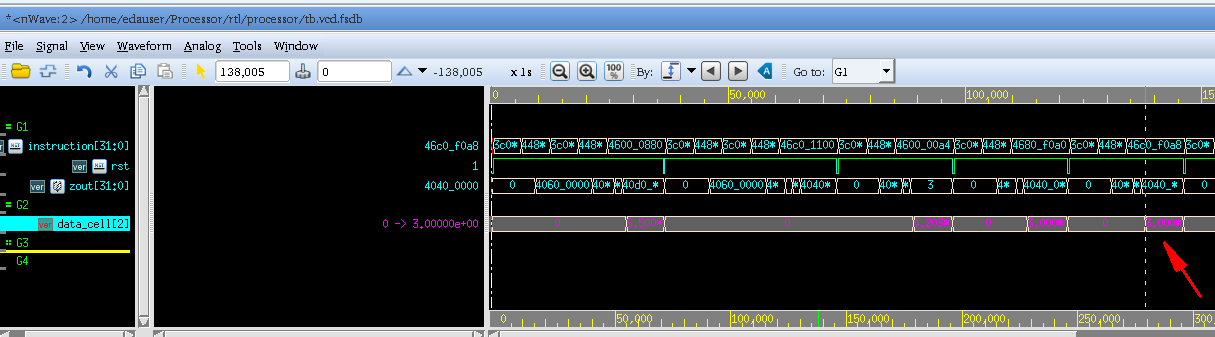
3.3.3 CVT.W.S



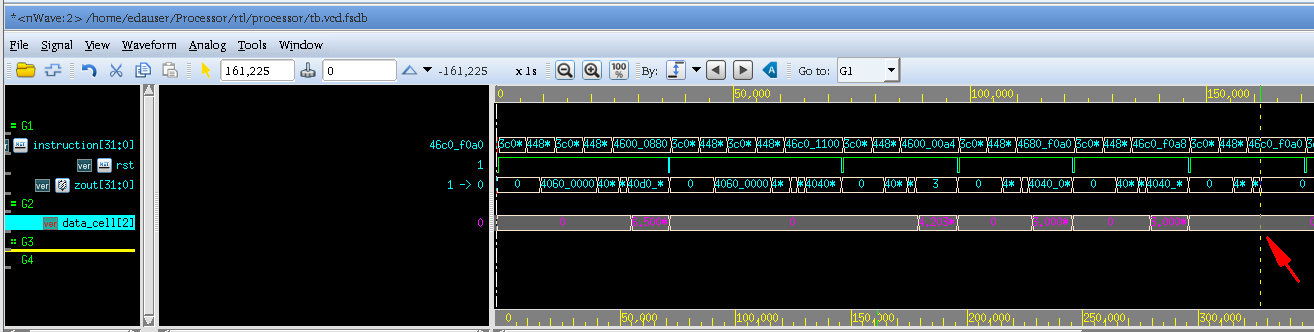
3.3.4 CVT.S.W



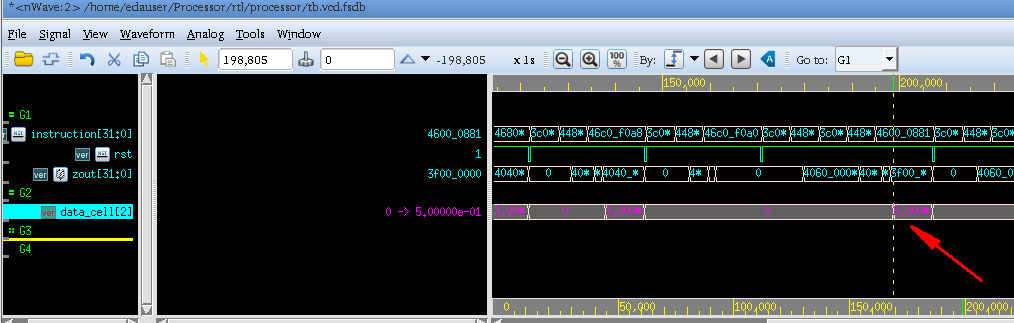
3.3.5 CVT.S.PL



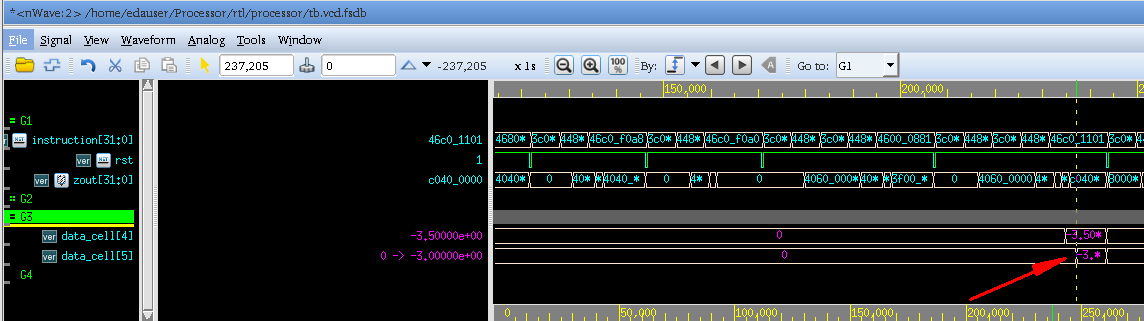
3.3.6 CVT.S.PU



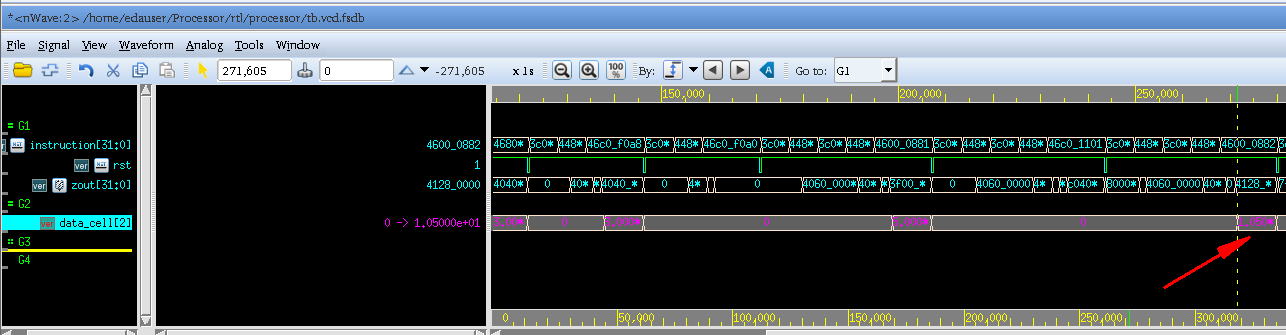
3.3.7 SUB.S



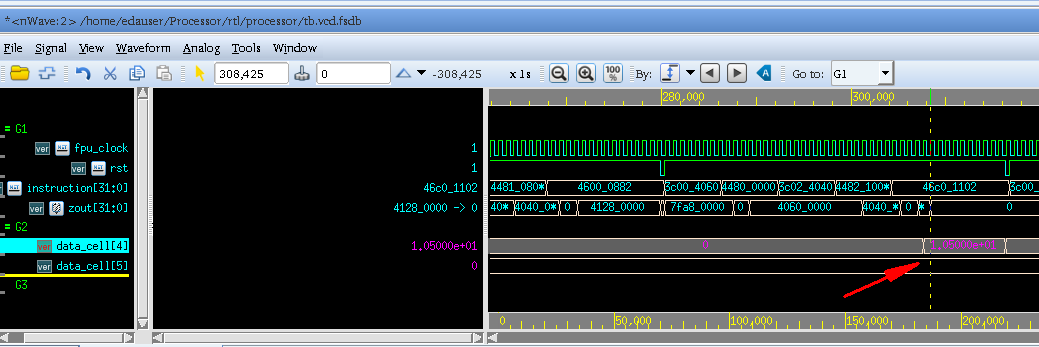
3.3.8 SUB.PS



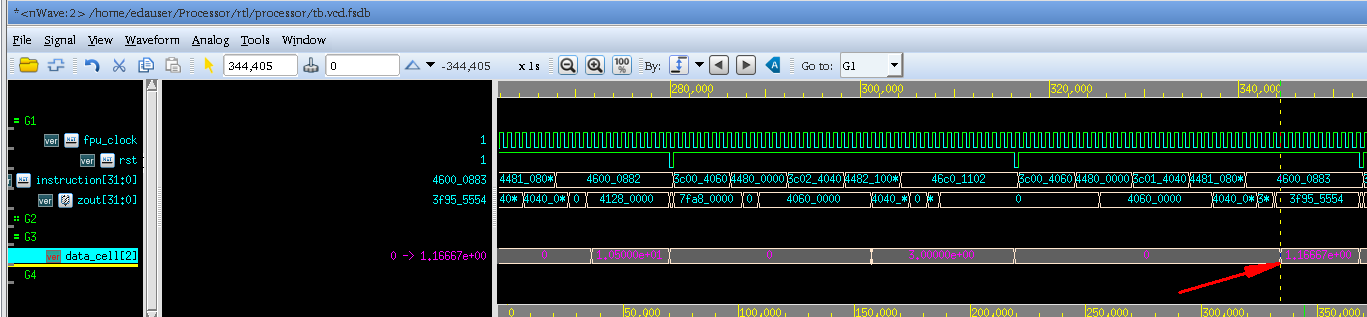
3.3.9 MUL.S



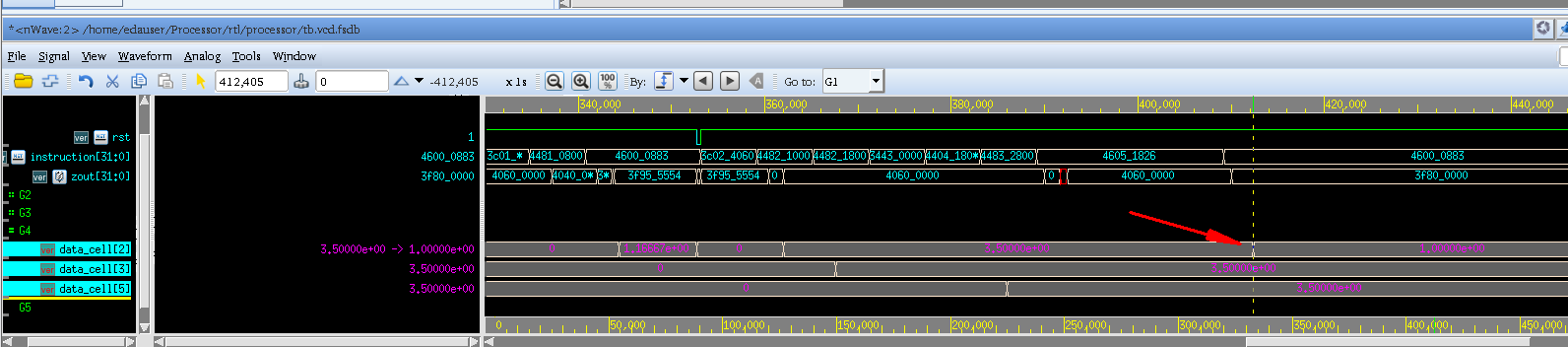
3.3.10 MUL.PS



3.3.11 DIV.S



3.3.12 CVT.PS.S



四、调试

4.1 调试技巧

（1）在命令行中可以用方向键快速切换之前的命令，不用重新输入；

（2）可以用file.f集成想要的所有文件名，以便统一操作；

（3）想要重新跑一次时，可以直接在命令行中ctrl+c来终止目前的前台进程（verdi），再用方向键重新操作一次；

（4）在waveform窗口中可以通过修改radix来切换进制；

（5）在verdi代码窗口中通过双击或者右键，可以跳转到其被引用的地方；

（6）可以通过ok—continue查看内存中的波形。

4.2 调试过程与结果

（1）调试过程见日志，调试结果前文已经给出；

（2）代码覆盖率大约占到70%~80%，缺失的部分：

add的NORMALIZE中，自我循环时判断的是zm，而没有判断指数，可能出现中途溢出；

div为OVER状态时，指数运算xe - ye可能溢出；