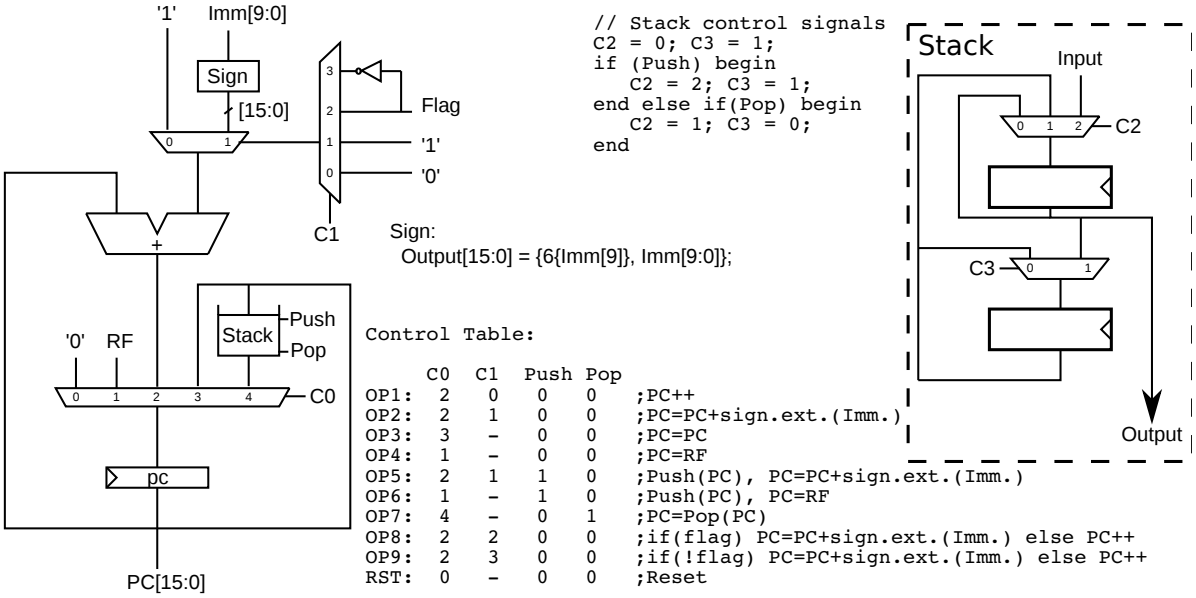
Solution proposal for the 2010-10-23 TSEA26 exam (v1.1)

Question 1




```
// Stack control signals
C2 = 0; C3 = 1;
if (Push) begin
   C2 = 2; C3 = 1;
end else if(Pop) begin
   C2 = 1; C3 = 0;
end
```

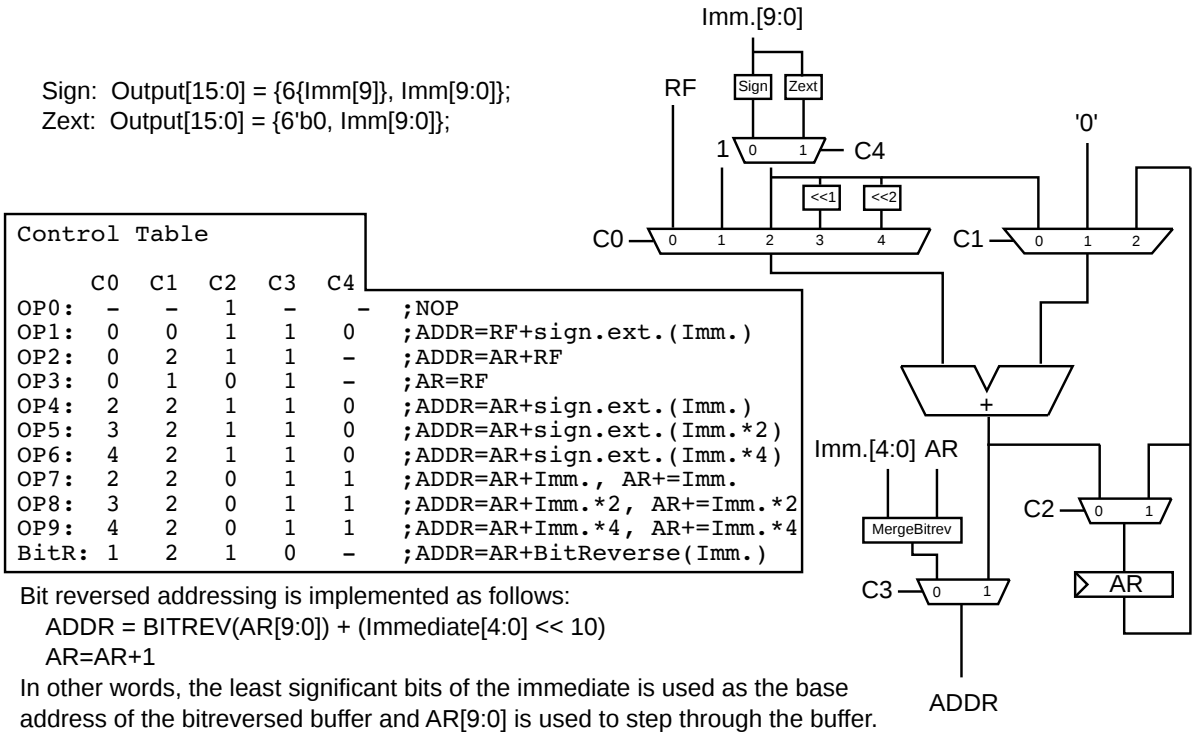
Sign:
Output[15:0] = {6{Imm[9]}, Imm[9:0]};

Control Table:

	C0	C1	Push	Pop	
OP1:	2	0	0	0	;PC++
OP2:	2	1	0	0	;PC=PC+sign.ext.(Imm.)
OP3:	3	-	0	0	;PC=PC
OP4:	1	-	0	0	;PC=RF
OP5:	2	1	1	0	;Push(PC), PC=PC+sign.ext.(Imm.)
OP6:	1	-	1	0	;Push(PC), PC=RF
OP7:	4	-	0	1	;PC=Pop(PC)
OP8:	2	2	0	0	;if(flag) PC=PC+sign.ext.(Imm.) else PC++
OP9:	2	3	0	0	;if(!flag) PC=PC+sign.ext.(Imm.) else PC++
RST:	0	-	0	0	;Reset

Question 2

Sign: Output[15:0] = {6{Imm[9]}, Imm[9:0]};
Zext: Output[15:0] = {6'b0, Imm[9:0]};



Control Table

	C0	C1	C2	C3	C4	
OP0:	-	-	1	-	-	;NOP
OP1:	0	0	1	1	0	;ADDR=RF+sign.ext.(Imm.)
OP2:	0	2	1	1	-	;ADDR=AR+RF
OP3:	0	1	0	1	-	;AR=RF
OP4:	2	2	1	1	0	;ADDR=AR+sign.ext.(Imm.)
OP5:	3	2	1	1	0	;ADDR=AR+sign.ext.(Imm.*2)
OP6:	4	2	1	1	0	;ADDR=AR+sign.ext.(Imm.*4)
OP7:	2	2	0	1	1	;ADDR=AR+Imm., AR+=Imm.
OP8:	3	2	0	1	1	;ADDR=AR+Imm.*2, AR+=Imm.*2
OP9:	4	2	0	1	1	;ADDR=AR+Imm.*4, AR+=Imm.*4
BitR:	1	2	1	0	-	;ADDR=AR+BitReverse(Imm.)

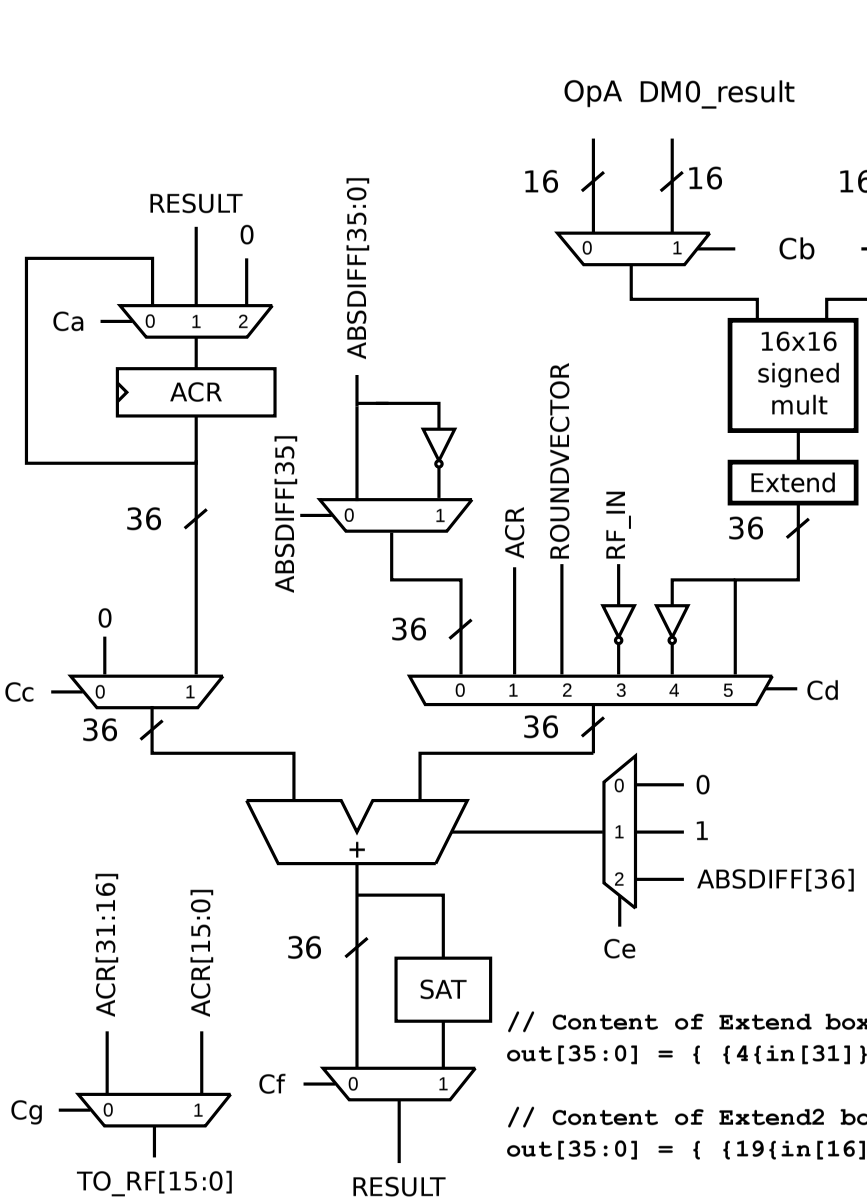
Bit reversed addressing is implemented as follows:
ADDR = BITREV(AR[9:0]) + (Immediate[4:0] << 10)
AR=AR+1
In other words, the least significant bits of the immediate is used as the base address of the bitreversed buffer and AR[9:0] is used to step through the buffer.
Explanation of MergeBitrev: out[15:0] = {Imm[4:0],AR[0:9]};

Question 3

For the first function we use a small trick, we don't have a separate multiplication instruction depending on whether we use fractional or integer multiplication. Instead we scale the result in program 1 by two by adding the ACR to itself after the multiplications are finished. This is ok as the exercise only specifies that the result in r6/r7 should be in fractional, our intermediate calculations can be done in whatever way we want to. Note: For the `sat.rnd` instruction, ACR[15:0] will become undefined due to the solution used in the schematic. However, this doesn't matter in this case as only ACR[31:16] is accessed after this.

In program 2 we perform another trick. By reordering the instructions slightly we need one accumulator without violating the performance constraints.

```
// function butterfly_part()
mul ACR, r0, r2     // ACR = r0 * r2
mdm ACR, r1, r3     // ACR -= r1 * r3
add ACR, ACR        // ACR = ACR << 1 (Shift up one step to
                    // compensate for fractional)
sub.shift16 ACR, r4 // ACR = ACR - r4 << 16
sat.rnd ACR, ACR    // ACR = SAT(ROUND(ACR))
move r6, ACR[31:16] // r6 = ACR[31:16]
mul ACR, r0, r3
mac ACR, r1, r2     // ACR += r1 * r3
add ACR, ACR
sub.shift16 ACR, r5
sat.rnd ACR, ACR
move r7, ACR[31:16]
ret
// function filter()
clear ACR           // ACR = 0
set AR0, r0
set AR1, r1
repeat 30                       // Only repeats next instruction
mac ACR,DM0[AR0++],DM1[AR1++] // ACR += DM0[AR0++] * DM1[AR1++]
sat ACR,ACR
move r2,ACR[31:16]
move r3,ACR[15:0]
clear ACR
set AR0,r0
set AR1,r1
repeat 30                       // Only repeats next instruction
diffacc ACR0,DM0[AR0++],DM1[AR1++] // ACR = ACR +
                                   // ABS(DM0[AR0++]-DM1[AR1++])
move r4,ACR[31:16]
move r5,ACR[15:0]
ret
```



Control table		Ca	Cb	Cc	Cd	Ce	Cf	Cg
nop		0	-	-	-	-	-	-
clear	ACR	2	-	-	-	-	-	-
mul	ACR, OpA, OpB:	1	0	0	5	0	0	-
mdm	ACR, OpA, OpB	1	0	1	4	1	0	-
mac	ACR, DM0[], DM1[]	1	1	1	5	0	0	-
diffacc	ACR, DM0[], DM1[]	1	1	1	0	2	0	-
add	ACR, ACR, ACR	1	-	1	1	0	0	-
sub.shift16	ACR, OpA	1	-	1	3	0	0	-
sat	ACR, ACR	1	-	0	1	0	1	-
sat.rnd	ACR, ACR	1	-	1	2	0	1	-
move	RF, ACR[31:16]	0	-	-	-	-	-	1
move	RF, ACR[15:0]	0	-	-	-	-	-	0

```
// Content of Extend box:
out[35:0] = { {4{in[31]}}, in};

// Content of Extend2 box:
out[35:0] = { {19{in[16]}}, in};
```

```
// Explanation of RF_IN and ROUNDVECTOR
RF_IN = { {4{OpA[15]}}, OpA[15:0], 16'b0};'
ROUNDVECTOR=36'h8000;

// Content of SAT box:
if(in[35:31] != {5{in[35]}}) begin
    out = { {5{in[35]}}, ~{31{in[35]}};
end else begin
    out = in;
end
```

Question 4

Version with cmul, cadd, and csub

This exercise is quite straight forward to solve: Just create a complex multiply, complex add, and complex subtract instruction.

```
// I'm assuming the following aliases for OpA/OpB:
REA = OpA[15:0] ; IMA = OpA[31:16]
REB = OpB[15:0] ; IMB = OpB[31:16]

// Instruction set:
cmul: TO_RF[15:0]  = REA*REB - IMA*IMB;
      TO_RF[31:16] = REA*IMB + REB*IMA;
csub: TO_RF[15:0]  = REA-REB;
      TO_RF[31:16] = IMA-IMB;
cadd: TO_RF[15:0]  = REA+REB;
      TO_RF[31:16] = IMA+IMB;

// Assembler listing:
// Assume i0 is in r0, c0 in r8, i1 in r1
//        i2 is in r2, c1 in r9, i3 in rr
//        o0 in r12, o1 in r13, o2 in r14
//        o3 in r15

cmul r12, r0, r8
csub r12, r12, r1
cadd r13, r0, r1

cmul r14, r2, r9
csub r14, r14, r3
cadd r15, r2, r3
```

A more area efficient solution: Merge cmul and csub

Implementing a complex multiplier in a straight forward fashion will use four multipliers. This is very wasteful as we will only use them two times the 7 clock cycles we are allowed to use.

Luckily, it is quite easy to solve this exercise using only two multipliers by combining the complex multiply and complex subtract into two instructions where the first instruction performs two multiplications and the second instruction performs another two instructions:

```
// Instruction set
```

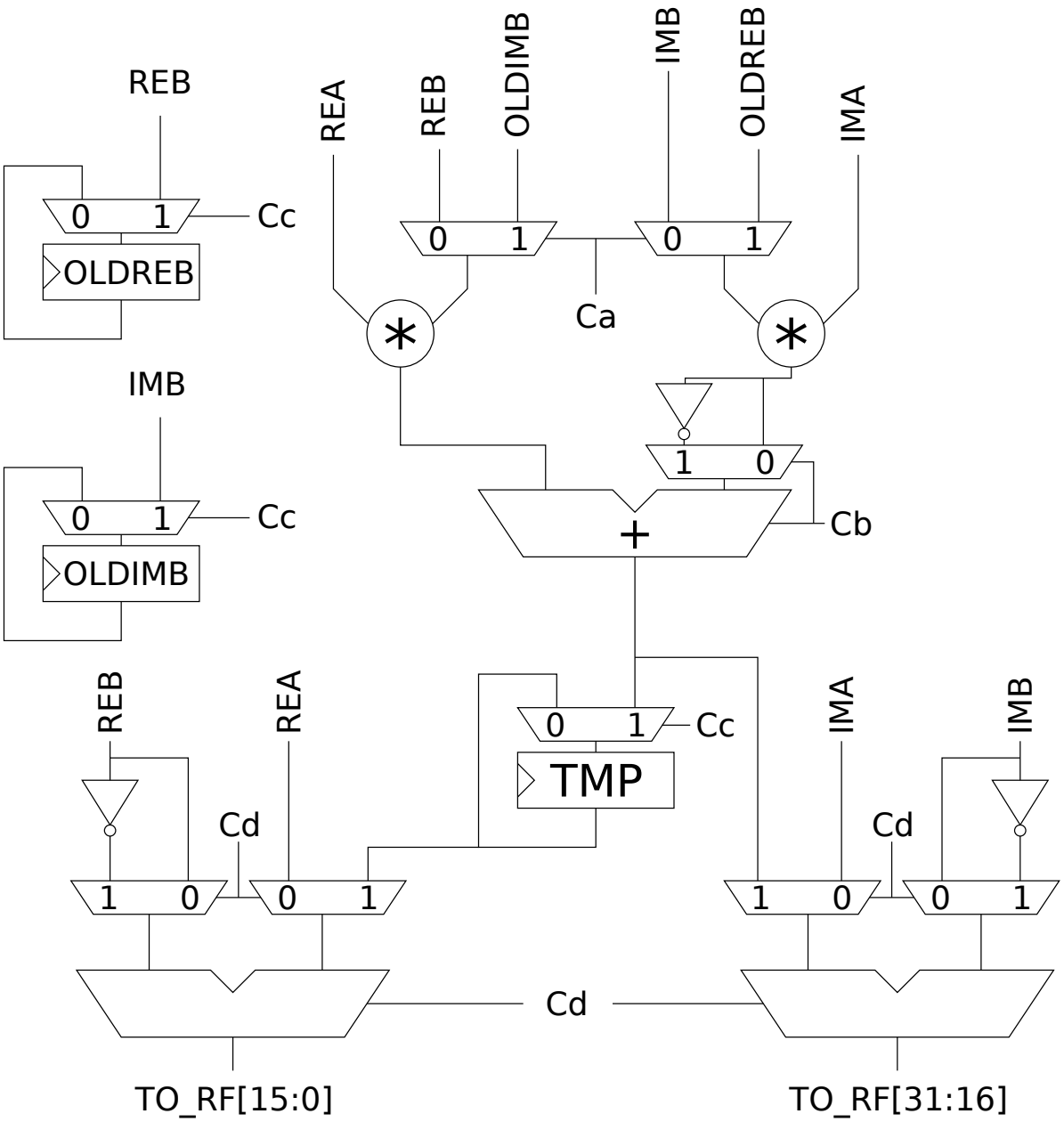
```
csubmul1 : TMP = REA * REB - IMA * IMB;
           OLDREB = REB;
           OLDIMB = IMB;
csubmul2 : TO_RF[31:16] = REA * OLDIMB + OLDREB * IMA - IMB;
           TO_RF[15:0]  = TMP - REB;
cadd     : TO_RF[31:16] = IMA + IMB;
           TO_RF[15:0] = REA + REB;

// Assembler listing

csubmul1 r0, r8 // No destination reg
csubmul2 r12, r0, r1
cadd     r13, r0, r1

csubmul1 r2, r9 // No destination reg
csubmul2 r14, r2, r9
cadd     r15, r2, r9
```

Solution with only one multiplier

It should be noted that it is actually possible to solve this exercise using only one multiplier. This is based on the fact that it is possible to perform a complex multiplication using only three multiplications (at a cost of a few extra additions)[1]. Using this algorithm it is possible to perform the two complex multiplications required by the algorithm in only 6 clock cycles at a cost of only one multiplier. (However, since I never mentioned this possibility during the lectures, I will leave the rest of this solution as an exercise for the reader.)

[1]For details, see for example `http://en.wikipedia.org/wiki/Multiplication_algorithm#Gauss.27s_complex_multiplication_algorithm`.

REA = OpA[15:0]; IMA = OpA[31:16]
REB = OpB[15:0]; IMB = OpB[31:16]

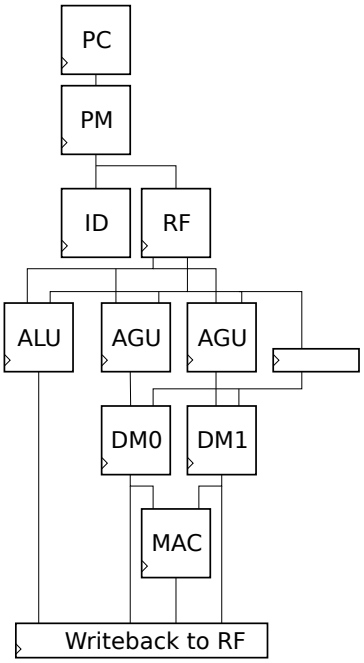


Control table	Ca	Cb	Cc	Cd
csubmul1	0	1	1	-
csubmul2	1	0	0	1
cadd	-	-	0	0

Question 5

a) Program memory

b)



In case you didn't answer correctly on the previous question, I would still give you points for it if you included a program memory in this answer (assuming the answer was not completely wrong for other reasons).

c) The maximum value of the FIR filter is approximately $(0.4 + 1.4 + 1.1 + 0.1) \cdot 2 = 6$. This means that 3 guard bits are required.

d) In this exercise I want you to mention branch instructions. I also want you to mention that whether the branch condition is true or not is not known until the instruction has gotten pretty far in the pipeline when a control hazard occurs.

Common problems

As a service to the reader I have included the most common mistakes for each exercise. (The most common mistake is the first one for each exercise.)

Question 1

- Minor error in the control table
- Stack cannot work.
- PC output is wrong.
- Implementation of two-entry stack is not given in detail.
- The PC register is missing.
- Sign extension function is wrong.
- Stack input is wrong.
- Sign extension function is not described.
- Using two adders in the design is not optimal.
- This design cannot perform OP8 and OP9.
- The schematic is not reasonable
- Reset is wrong.
- Implementation of Flag is wrong.
- Stack is not implemented.

Question 2

- Bit reverse function should perform bit reverse on the LSB 10-bits.
- Scaling block with equal input/output width will lose MSB.
- Bit reverse is not implemented.
- Scaling function is wrong.
- AR cannot be updated by new ADDR.
- No solution is given.
- Bit reverse is not reasonable
- Scaling function is not described.

- Minor error in the control table
- Why do you use saturation in the address generator unit?
- The schematic is not reasonable
- Sign extension function is wrong.
- Sign extension function is not described.
- Bit reverse does not support iteration.

Question 3

- No control table included
- The accumulator is larger than necessary. (The largest value that sumofproducts can be set to is 0x0780000000 which can be represented in two's complement integer format by using 36 bits.
- The MAC schematic is not reasonable
- You don't describe how your saturation works.
- You are performing part (or all) of the butterfly_part() function using regular registers instead of accumulators. There are two problems with this: First, you are losing some precision. Second, you may not be able to detect an overflow in all situations. (Or, if you are using saturation for these operations, you may detect an overflow that wouldn't be an overflow if all calculations (except SAT/ROUND) was performed using high precision.
- Your accumulator is smaller than necessary. (The largest value that sumofproducts can be set to is 0x0780000000 which can be represented in two's complement integer format by using 36 bits.
- You cannot execute the function filter() in 80 clock cycles.
- Your sum of absolute difference-instruction(s) does not work correctly.
- You don't describe how your round vector block works.
- You are using more than two read ports and one write port on the register file.
- Your multiplier is larger than necessary. (Only a 16x16-bit signed multiplier is required.)
- You are not following the input specification. The exercise said that the inputs to the module are: DM0_result, DM1_result, OpA, and OpB. You are not using all of these inputs.
- There is nothing in the specification of this exercise that allows you to send a flag output to the PC module. This means that you can't branch based on the contents of an accumulator register.

- You have used more than two accumulators[2].
- You don't describe how you add guard bits (or how you sign extend a value)
- You have a a signal of a certain width which is converted into another signal with another width without specifying how this is done. (E.g., sign extension, left shift, zero extension, etc...)
- You are performing part of the filter() function using regular registers instead of accumulators. This mean that you will not be able to detect an overflow in all situations.
- You have not included assembly code for the butterfly_part() function
- You have a minor error in your control table
- Your control table does not contain all instructions that you created in the first part of the exercise.
- You don't describe how you add guard bits (or how you sign extend a value)
- While I like the idea of avoiding the need for a subtract instruction by setting a register to 1 and then using the multiply and diminish instruction, it will not work correctly. (It is not possible to represent 1 when using a fractional number.) (-1 in conjunction with a regular MAC instruction would have worked however...)
- You are supposed to use signed integer multiplication in the filter() function, not fractional multiplication
- You don't describe how your integer/fractional conversion block works.
- You need guard bits for the absdiff operation
- The Verilog syntax that you are using is not correct. This is not normally a big problem. However, in your saturation unit I'm not sure what you actually intend to do due to this. For example, 5'b1 is the same as 5'b00001. And {5{1'b1}} is the same as 5'b11111. (But 5'b11111 is not the same as 5'b1!)

Question 4

- No control table was included
- Four multipliers were used. It is actually possible to solve this exercise using only two multipliers.
- The proposed solution is not reasonable.
- The complex multiplication does not work correctly for some reason
- The complex multiplication does not perform any subtract operation for the real part.

[2]I don't deduct any points for using two accumulators however, even though the exercise can be solved using only one accumulator

- More than two read ports from the register file and one write port to the register file is used.

Question 5

- The answer to part C is not correct.
- The answer to part D is not reasonable.
- Your answer to question A is not correct. Luckily you included a program memory in part B, so I will not deduct any points
- You need to elaborate your answer to part D.
- I'm sorry, but if you don't include a program memory in your schematic I can't give any points for part B.
- The answer to part A is not reasonable
- In 5d you write that you don't know whether a jump should be taken or not. Why is that?
- Your RF module is not located in such a way that operands can be read out from it in a sane manner.
- Where is your register file in part B?

Point distribution

- No grade: 35
- Grade 3: 24
- Grade 4: 11
- Grade 5: 4

Comments: A little more than half of the students passed the exam. While I'm not sure, I believe that the most likely reason for this is that quite a few students seemed to lack the prerequisites for this course (basic computer architecture and digital design).