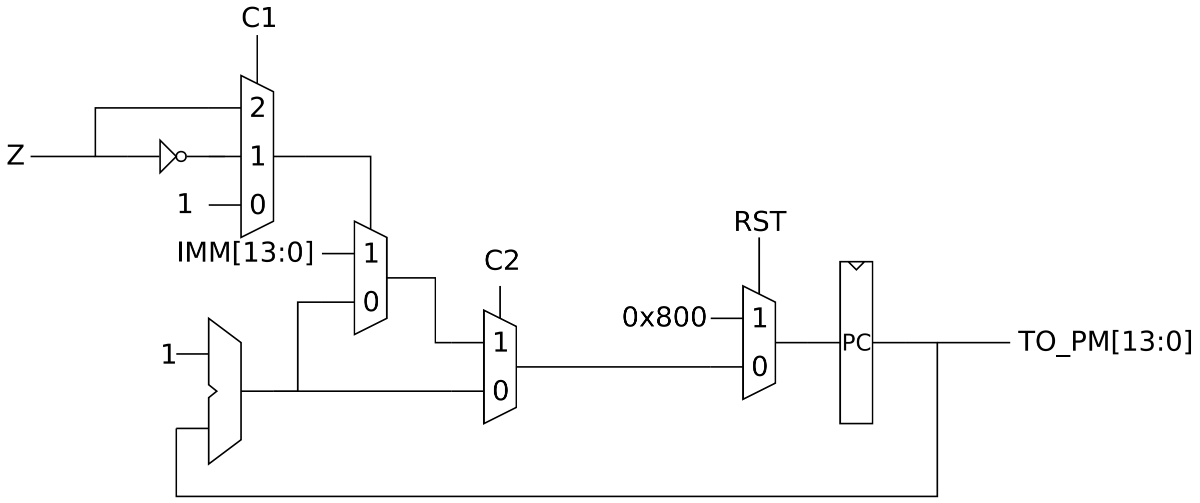
Solution proposal for the 2011-08-23 TSEA26 exam (v1.4)

Question 1



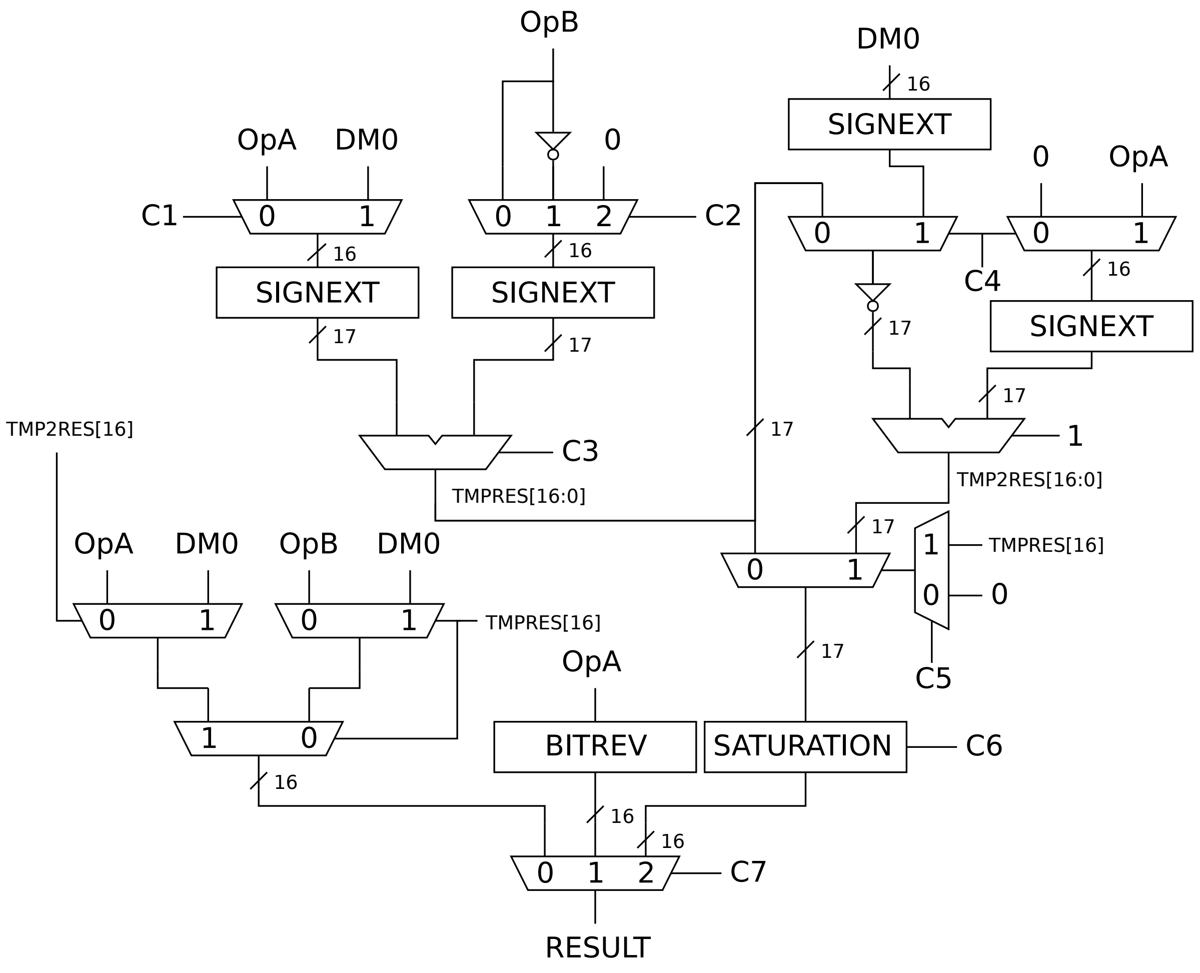
Note: All signals in the schematic above are 14 bits (except for control signals to multiplexers).

Operation	C1	C2
OP1	2	1
OP2	1	1
OP3	0	1

Comment: This isn't a very realistic PFC as there is no "NOP" operation which simply increments the program counter. The schematic above supports this operation by setting C1 and C2 to zero, but no points will be deducted if your solution can't handle such an operation. (However, your PFC still needs to be able to increment the program counter in case a conditional branch is not taken!)

Question 2

This is a solution proposal for Q2 for constraint alternative 1 (worth 5 points).



```
SIGNEXT: out[16:0] = {in[15], in[15:0]};

BITREV: out[15:0] = in[0:15];

SATURATION:

if (in[16] != in[15]) begin
   if(in[16]) out = 16'h8000;
   else out = 16'h7fff;
end else begin
   out = in[15:0];
end
```

Operation	C1	C2	C3	C4	C5	C6	C7
OP1: OpA+OpB	0	0	0	-	0	0	2
OP2: OpA-OpB	0	1	1	-	0	0	2
OP3: SAT(OpA-OpB)	0	1	1	-	0	1	2
OP4: SAT(ABS(OpA))	0	2	0	0	1	1	2
OP5: SAT(ABS(OpA-OpB))	0	1	1	0	1	1	2
OP6: BITREV(OpA)	-	-	-	-	-	-	1
OP7: LD.MINMAX	1	1	1	1	-	-	0

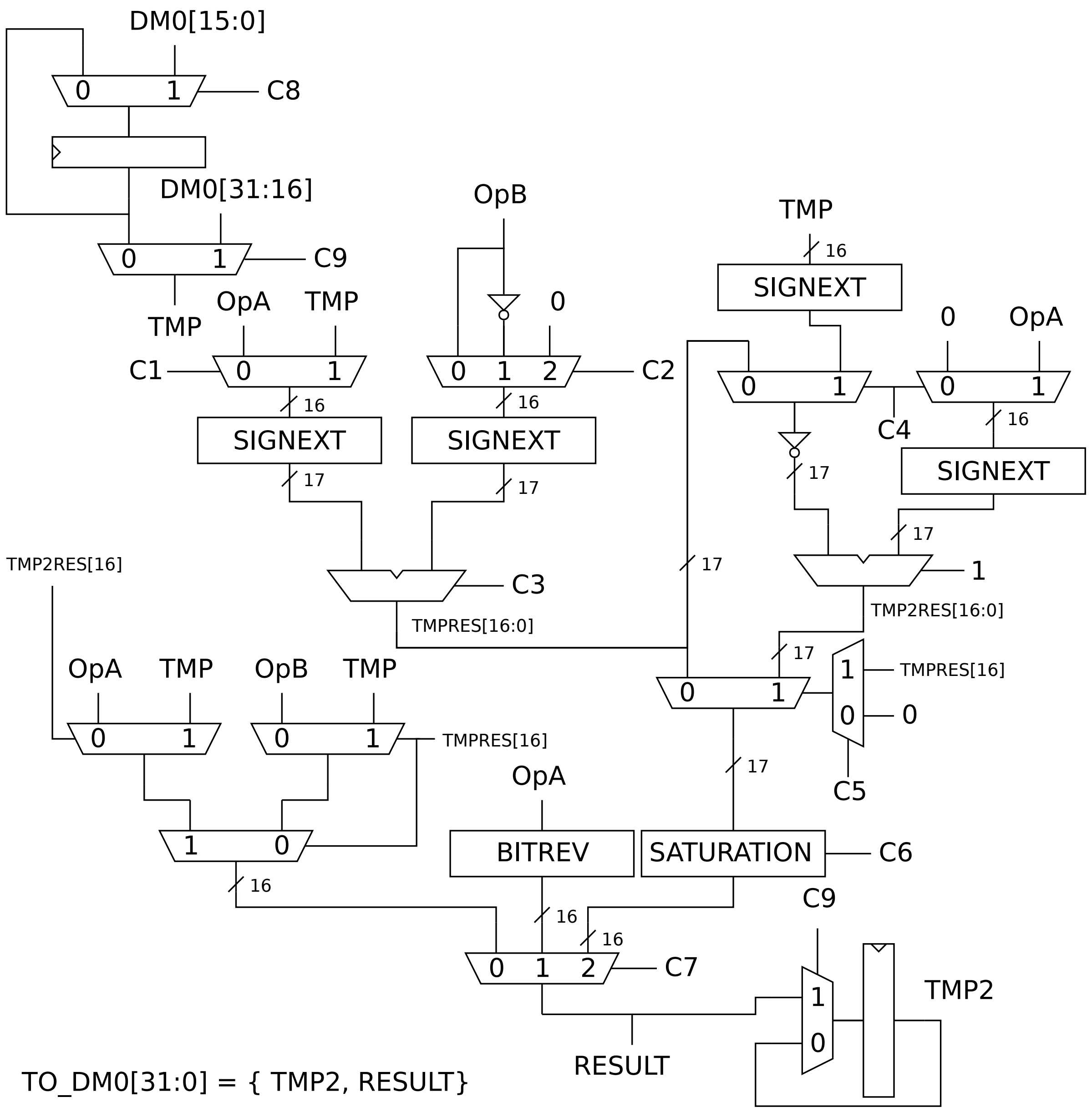
```
saturate_values:
     move ar0,r0

     ; If you don't have any dependency checking and register bypass
     ; logic in your pipeline you may have to unroll this loop.
     repeat 160,endloop
     ld.minmax r3,DM0[ar0],r2,r1  ; r3 = max(min(r1,DM0[]),r2)
     st DM0[ar1++],r3
endloop:

     ret
```

The difficult (9p) version of part b can be solved by doubling the width of the memory so that 32 bits can be read out at the same time. Note that doubling the width of the memory can also be beneficial for other units, such as the MAC unit. However, it is not something to undertake lightly, as a wider memory will consume more power and if you don't have any use for a double wide read most of the time you will waste quite a lot of power.

- ld.double.minmax DM0[ar0],r2,r1 ; TEMPVAL = DM0[15:0]; TMP2 = max(min(r1,DM0[31:16]),r2);
- st.double.minmax DM0[ar0+=2],r2,r1 ; DM0[31:16] = TMP2; DM0[15:0] = max(min(r1,TEMPVAL),r2); ar0+=2;

The assembly program which implements this routine looks like this:

```
saturate_values:
    move ar0, r0

    repeat 80, endloop
    ld.double.minval DM0[ar0],r2,r1
    st.double.minval DM0[ar0+=2],r2,r1
endloop:
    ret
```



Operation	C1	C2	C3	C4	C5	C6	C7	C8	C9
OP1-OP6: Same as before									
LD.DOUBLE.MINMAX	1	1	1	1	-	-	0	1	1
ST.DOUBLE.MINMAX	1	1	1	1	-	-	0	0	0

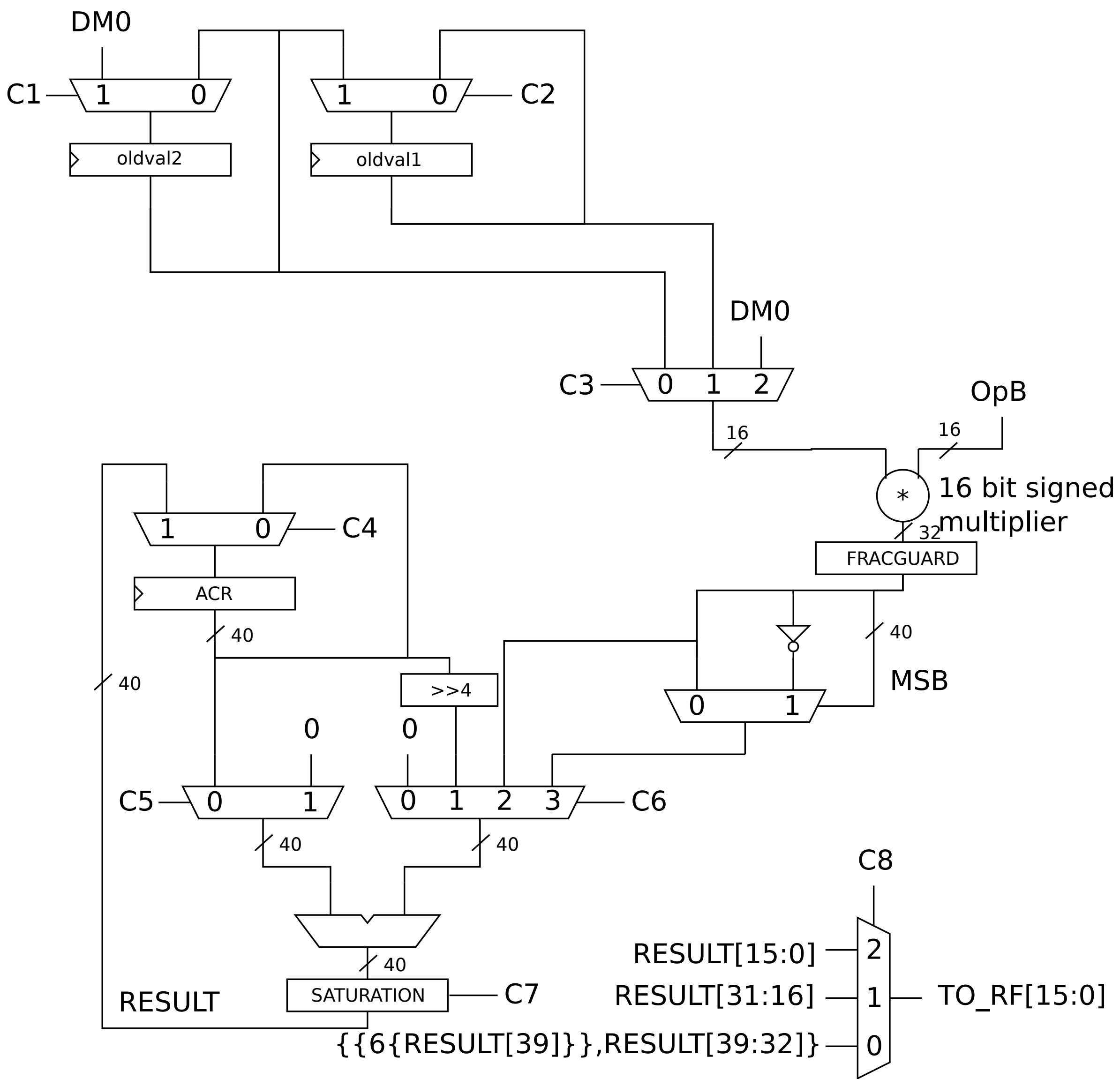
Question 3

This exercise can be answered in many different ways. The following is a minimal assembly language answer which is possible to implement in a reasonable hardware efficient manner. However, it is somewhat unrealistic as it assumes that it is possible to forward a result from the load unit directly to the next instruction (which uses the MAC unit).

```
; The first key to making this exercise easier for a hardware
; designer is to recognize that there is no need to check flag
; in the middle of the loop.
;
; No points would be deducted for checking the flag in the hardware
; however, as long as it is done in a reasonable way. (E.g., you must
; show that the flag comes from OpA or OpB.)
do_filter:
    move    ar0,r0
    move    ar1,r1
    cmp     r3,0
    clr     ACR
    jump.eq flag_is_zero

    ; It is not very realistic to use the result of the load immediately,
    ; in real life you would probably unroll this loop at least one time.
    repeat 100, absloop_end
    ld      r4,DM0[ar0++]
    mac.abs ACR, DM0[ar1++], r4
absloop_end

    jump loop_end

flag_is_zero
    repeat 100, loop_end
    ld      r4,DM0[ar0++]
    mac     ACR, DM0[ar1++], r4
loop_end:

    mac.checkval r0,ACR   ; Sets r0 to 1 if ABS(ACR) >= 0x80000000
    cmp     r0,0
    jump.eq done
    rshift4 ACR           ; ACR = ACR >> 4
    set     r10, 1
    st      DM0[r2],r10

done:
    sat ACR
    move r0, HIGH(ACR)
    ret

do_small_fir:
    add     r0,r0, 2
    move    ar0,r0
    move    ar1,r1
    repeat 99, endloop
    mul ACR, DM0[ar0-2], r2     ; ACR = oldval1 * tap1
    mac ACR, DM0[ar0-1], r3     ; ACR += oldval2 * tap2
    mac.sat r0, DM0[ar0++], r4  ; r0 = SATURATE(val * tap3 + ACR)
    st DM0[ar1++], r0
endloop
    ret
```

The above solution would be perfectly ok on the exam. However, the mac.checkval instruction looks like a fairly ugly special case even though there is plenty of clock cycles available for prologue/epilogue. Additionally, the do_small_fir function performs many redundant reads from memory, increasing the power consumption significantly. Therefore I propose the following solution:

```
do_filter:
    move    ar0,r0
    move    ar1,r1
    cmp     r3,0
    jump.ne flag_is_zero
    clr     ACR                ; Delay slot

    ; Unrolled four times to avoid possible data hazards
    repeat 25, absloop_end
    ld      r4,DM0[ar0++]
    ld      r5,DM0[ar0++]
    ld      r6,DM0[ar0++]
    ld      r7,DM0[ar0++]
    mac.abs ACR, DM0[ar1++], r4
    mac.abs ACR, DM0[ar1++], r5
    mac.abs ACR, DM0[ar1++], r6
    mac.abs ACR, DM0[ar1++], r7
absloop_end

    jump loop_end
    nop                        ; Delay slot

flag_is_zero
    repeat 25, loop_end
    ld      r4,DM0[ar0++]
    ld      r5,DM0[ar0++]
    ld      r6,DM0[ar0++]
    ld      r7,DM0[ar0++]
    mac     ACR, DM0[ar1++], r4
    mac     ACR, DM0[ar1++], r5
    mac     ACR, DM0[ar1++], r6
    mac     ACR, DM0[ar1++], r7
loop_end:

    ; The comparison is very similar to the one done when saturating a
    ; number. Unfortunately it is not quite as easy as we also need to
    ; check whether ACR == 0xff80000000.
    move    r4, GUARD(ACR)

    cmp     r4, 0      ; Check whether guard bits are all 0
    jump.eq checkhigh
    move    r5, HIGH(ACR) ; Delay slot

    cmp     r4, -1
    jump.ne doshift     ; If guard bits are not all 1 here we
    move    r6, LSB(ACR); need to shift!

    cmp      r5, 0x8000 ; Check whether ACR[31:16] is larger
    jump.ugt done       ; than 0x8000. If so we are done.
    nop

    cmp      r6,0       ; If r6 is zero we know that ACR is
    jump.eq  doshift    ; equal to 0xff80000000 here.
    nop
    jump     done


checkhigh:
    cmp     r5, 0
    jump.pl done  ; Jump if positive (i.e., less than 0x8000)
    nop

doshift:
    ; Ok, at this point it is clear that ABS(ACR) > 0x80000000.
    ; Scale ACR down four steps and set a flag indicating that
    ; this was done. (This could allow the main program to for
    ; example scale down the input data from this point in order
    ; to implement some sort of block floating point.)
    set     r0,1
    rshift4  ACR   ; ACR = ACR >> 4
    st      DM0[r2],r0

done:
    sat     ACR
    nop
    move    r0,HIGH(ACR)
    ret
    nop


do_small_fir:
    ; First we load the shift registers
    move ar0,r0
    move ar1,r1
    mac.shifttaps.sat ACR,DM0[ar0++],r2
    mac.shifttaps.sat ACR,DM0[ar0++],r2
    clr     acr

    ; This one is also unrolled a few times to
    ; avoid data hazards in the pipeline.
    repeat 33, endloop
    mul               ACR, oldval1,r2
    mac               ACR, oldval2,r3
    mac.shifttaps.sat r5, ACR, DM0[ar0++],r4
    mul               ACR, oldval1,r2
    mac               ACR, oldval2,r3
    mac.shifttaps.sat r6, ACR, DM0[ar0++],r4
    mul               ACR, oldval1,r2
    mac               ACR, oldval2,r3
    mac.shifttaps.sat r7, ACR, DM0[ar0++],r4
    st  DM1[ar1++], r5
    st  DM1[ar1++], r6
    st  DM1[ar1++], r7
endloop

    ret
```



Control table:

Instruction	C1	C2	C3	C4	C5	C6	C7
nop	0	-	0	-	-	-	-
clr ACR	0	-	1	1	0	-	-
mac.abs	0	2	1	0	3	0	-
mac	0	2	1	0	2	0	-
move rX,GUARD(ACR)	0	-	0	0	0	0	0
move rX,HIGH(ACR)	0	-	0	0	0	0	1
move rX,LOW(ACR)	0	-	0	0	0	0	2
rshift4 ACR	0	-	1	1	1	0	-
sat ACR	0	-	1	0	0	1	-
mul ACR,oldval1,rX	0	1	1	1	2	0	-
mac ACR,oldval2,rX	0	0	1	0	2	0	-
mac.shifttaps.sat ...	1	2	1	0	2	1	1

```
FRACGUARD: out[39:0] = { {7{in[31]}}, in[31:0], 1'b0};
```

```
SATURATION:
always @* begin
    case(in[39:31])
    9'h0:   out = in;
    9'h1ff: out = in;
    default: begin
        if(in[39]) out = 40'hff80000000;
        else out = 40'h007fffffff;
    endcase
end
```

There are many ways to solve this exercise of course. For example, the shift register design in the schematic above is not really necessary. The solution below does not require a shift register but solves the problem anyway, under the assumption that indexed addressing can be used at the same time as the mac instruction is running. (However, this solution will consume more power than the previous solution, as DM0 will be read more often.)

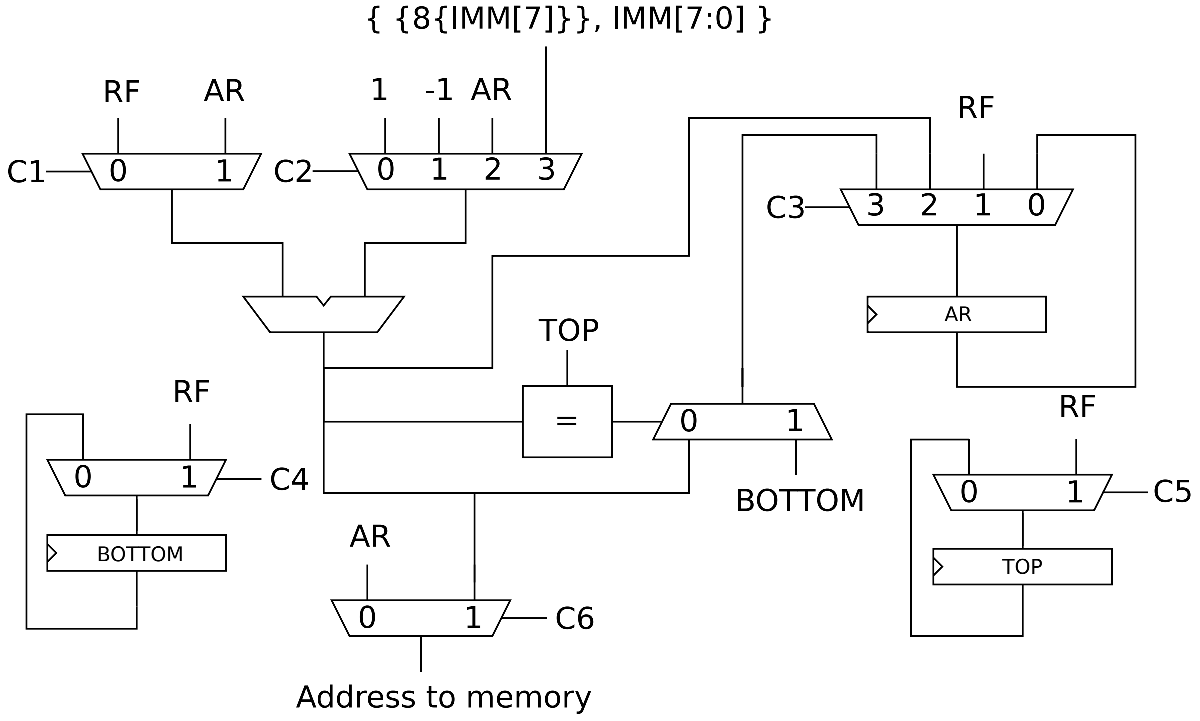
```
alternative_do_small_fir:
    add     r0,r0,2
    set     ar0,r0
    set     ar1,r1
    clr     acr

    ; Loop unrolling may be necessary in a real processor but
    ; not included here for readability.
    repeat 99, endloop
    mul     ACR, DM0[ar0-2],r2
    mac     ACR, DM0[ar0-1],r3
    mac.sat r5, DM0[ar0++],r3
    st      DM1[ar1++]
endloop:
    ret
```

Note that the question allowed us to send data directly to DM1 from the MAC unit. None of the solutions above used this fact, but it could allow us to solve the exercise in different ways such as the following solution which allows us to avoid extra memory loads without the need for a shift register in hardware.

```
yet_another_do_small_fir:
    set     ar0,r0
    set     ar1,r1
    clr     acr
    ld      r5,DM0[ar0++]
    ld      r6,DM0[ar0++]
    repeat  33,endloop
    ld      r7,DM0[ar0++]       ; r5 is oldval1,r6 is oldval2, r7 is val
    mul     ACR,r5,r2
    mac     ACR,r6,r3
    mac.sat DM1[ar1++],r7, r4

    ld      r5,DM0[ar0++]       ; r6 is oldval1, r7 is oldval2, r5 is val
    mul     ACR,r6,r2
    mac     ACR,r7,r3
    mac.sat DM1[ar1++],r5, r4

    ld      r6,DM0[ar0++]       ; r7 is oldval1, r5 is oldval2, r6 is val
    mul     ACR,r7,r2
    mac     ACR,r5,r3
    mac.sat DM1[ar1++],r6,r4
endloop
    ret
```

Finally, by combining the `ld` and the `mul` instruction into one instruction it would be possible to run this code in slightly more than 300 clock cycles instead of slightly more than 400 clock cycles without any big changes to the hardware.

Question 4



Control table:

Operation	C1	C2	C3	C4	C5	C6
OP1	-	-	0	0	0	0
OP2	0	2	0	0	0	1
OP3	1	0	2	0	0	1
OP4	1	0	2	0	0	0
OP5	1	1	2	0	0	1
OP6	1	1	2	0	0	0
OP7	1	3	0	0	0	1
OP8	0	3	0	0	0	1
OP9 (Modulo)	1	0	3	0	0	1
OP10 (Load AR)	-	-	1	0	0	-
OP11 (Load TOP)	-	-	0	0	1	-
OP12 (Load BOTTOM)	-	-	0	1	0	-

Comments: The original operation list does not include a way to load the address register. However as such an operation is necessary to actually use the modulo addressing mode, you need to include it anyway...

Question 5

a) No guard bits are required ($|0.25| + |0.125| + |-0.25| < 1$)

b) See the text book.

c)

```
out[15:0] = in[19:5] + in[4];
```

Unfortunately there was a typo in the original question. It was intended that question c should say that a 20 bit two's complement number in Q5.15 format should be rounded to Q5.10 format. (Due to this typo, part c was corrected fairly generously.)

Revision history

- V1.1: Fix repeat bug in Q3 (We should repeat 100 times, not 25) Noticed by Niklas Lundgren.
- V1.2: Added forgotten control table in exercise 2.
- V1.2: Fixed ABS bug in exercise 2. (INV(X)+1 should be used instead of INV(X)+1+1.) Noticed by Daniel Björklund.
- V1.3: Forgot the C6 mux in Q4. Noticed by Jeremia Nyman.
- v1.4: Fixed typo for jump.eq in Q3. Noticed by Rakesh Praveen.