Solution proposal for the 2011-10-22 TSEA26 exam (v1.3)

Problem 1

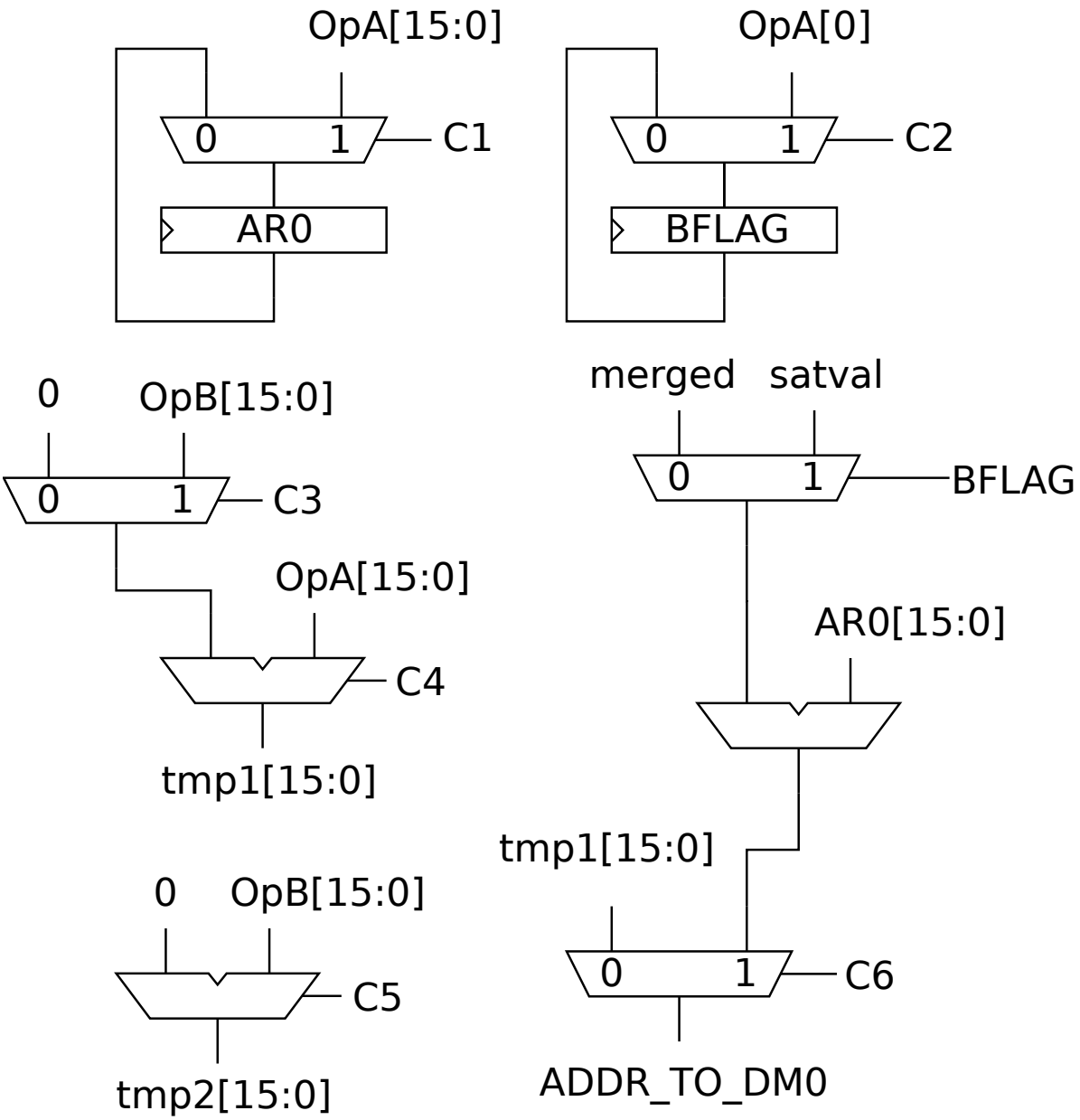
This exercise can be solved in quite a few different ways by trading off performance vs area. There are two tricky things in this problem:

- Realize that `blit()` can be rewritten so that only two multiplications with 160 is needed in the prologue of the kernel. (This multiplication can be done using regular MAC instructions.) Alternatively, if no hardware multiplier is available in the MAC unit (unlikely for a DSP processor), the multiplication with 160 can be done using two shifts and one add.
- Realize that the multiplication with 64 in `getsingle()` can be done using a shift.

In addition, if you realize that you can unroll the inner loop in the `blit()` function you only need one address register in the AGU (assuming that you want to avoid an excessive amount of read ports in your register file).

```
blit:
    mul  acr0,r1,160       ; r1 = yoffs1*160
    move r1,LOW(acr0)
    mul  acr0, r3, 160     ; r3 = yoffs2*160
    move r3, LOW(acr0)

    add r1,r1,r0           ; r1 = xoffs1+yoffs1*160
    add r3,r3,r2           ; r3 = xoffs2+yoffs2*160

    add r1,r1,img          ; r1 = &img[xoffs1+yoffs1*160]
    add r3,r3,ref          ; r3 = &ref[xoffs1+yoffs1*160]

    ; If we unroll the loop fully we only need register + offset
    ; addressing.
    repeat 8,endloop
    ld r4,DM0[r3+0]        ; AGUOP1
    st DM0[r1+0],r4

    ld r4,DM0[r3+1]
    st DM0[r1+1],r4

    ld r4,DM0[r3+2]
    st DM0[r1+2],r4

    ld r4,DM0[r3+3]
    st DM0[r1+3],r4
```

```
    ld r4,DM0[r3+4]
    st DM0[r1+4],r4

    ld r4,DM0[r3+5]
    st DM0[r1+5],r4

    ld r4,DM0[r3+6]
    st DM0[r1+6],r4

    ld r4,DM0[r3+7]
    st DM0[r1+7],r4

    add r1,r1,160
    add r3,r3,160
endloop:
    ret     ; 9+144 clock cycles (excluding RET instruction)

getsingle:
    move boundaryflag,r2   ; AGUOP2
    set ar0,tmem        ar0=&tmem[0]  ; AGUOP3
    clr acr
    add acr,DM0[(r0+   r1<<6)     + ar0] ; AGUOP4
    add acr,DM0[(r0+   (r1+1)<<6) + ar0] ; AGUOP5
    add acr,DM0[((r0+1)+r1<<6)    + ar0] ; AGUOP6
    add acr,DM0[((r0+1)+(r1+1)<<6)+ ar0] ; AGUOP7
    ret


////////////////////////////////////////////////////////////////////////
// Beginning of hardware design
////////////////////////////////////////////////////////////////////////

always @* begin
    merged = {4'b0,tmp2[5:0],tmp1[5:0]};
    satval = 0;

    if (tmp1[15]) satval[5:0] = 0;
    else if(|tmp1[15:6]) satval[5:0] = 63;
    else satval[5:0] = tmp1[5:0];

    if (tmp2[15]) satval[11:6] = 0;
    else if(|tmp2[15:6]) satval[11:6] = 63;
    else satval[11:6] = tmp2[5:0];
end
```



Assumptions made in this solution: There are two inputs to the AGU from the general purpose register file: OpA and OpB, each 16 bits wide. One of these inputs (OpB) can carry an immediate as well.

Operation	C1	C2	C3	C4	C5	C6
AGUOP1 (OpA+OpB)	0	0	1	0	-	0
AGUOP2 (set boundaryflag)	0	1	-	-	-	-
AGUOP3 (set AR0)	1	0	-	-	-	-
AGUOP4 (OpA + (OpB <<6) + ar0	0	0	0	0	0	1
AGUOP5 (OpA + (OpB + 1) <<6) + ar0	0	0	0	0	1	1
AGUOP6 (OpA + 1 + (OpB) <<6) + ar0	0	0	0	1	0	1
AGUOP7 (OpA + 1 + (OpB + 1) <<6) + ar0	0	0	0	1	1	1

Problem 1 - Alternative solution

An alternative solution to problem 1 is to create address registers with post-increment and pre-decrement mode. The DM0[ar0,ar1,ar2] addressing mode should take the boundaryflag into account in the same way that the previous solution does. Advantage of this solution: Small code size for blit() and no need to use OpA and OpB for AGUOP4-7 (allowing them to potentially be used to carry other data when these addressing modes are used). Disadvantage: More address registers are required (although in a real processor you would probably want to have more than one address register anyway).

```
blit:
    mul acc, r1,160          getit:
    move r1,acc                  move boundaryflag, r2
    mul acc, r3,160              move ar2,4000  // Address of tmem in DM0
    move r3,acc                  move ar0,ar1,r0,r1
                                 clr acr0
    set r6,0                     add acr0,DM0[ar0,ar1++,ar2]
    repeat 8, outerloopend       add acr0,DM0[ar0++,ar1--,ar2]
    add r4,r0,img                add acr0,DM0[ar0,ar1++,ar2]
    add r4,r4,r1                 add acr0,DM0[ar0,ar1,ar2]
    move ar0,r4                  ret

    add r5,r1,ref
    add r5,r5,r3
    move ar1,r5

    repeat 8, innerloopend
    ld r6,DM0[ar1++]
    st DM0[ar0++],r6
innerloopend:

    add r1,r1,160
    add r3,r3,160
outerloopend

    ret
// Note: blit can be optimized somewhat, but it is not needed to meet the
// performance constraints.
```

Why do you need these functions?

The blit() function is basically used for motion compensation (basically the inverse of what you did in lab 4 where you optimized a DSP processor to handle motion estimation in an efficient manner).

The getit() function, is based on the access patterns for bilinear interpolation of 2D

data. A more realistic getit() function could look like this in C when using floating point math (I felt that a more realistic example would be too cumbersome to handle on the exam so I simplified it):

```
static int getit_bilinear(float x, float y, int channel,int boundaryflag )
{
    int xc=x;
    int yc=y;

    double u_ratio = x - xc; // Get fractional parts
    double v_ratio = y - yc;

    u_ratio = 1;
    v_ratio = 0.5;
    double u_opposite = 1 - u_ratio;
    double v_opposite = 1 - v_ratio;
    double result;

    result =
      ( (float) getsingle(x,y,boundaryflag)     * u_opposite
    +   (float) getsingle(x+1,y,boundaryflag)   * u_ratio) * v_opposite
    + ( (float) getsingle(x,y+1,boundaryflag)   * u_opposite
    +   (float) getsingle(x+1,y+1,boundaryflag) * u_ratio) * v_ratio;


    return result;
}
```

Note that the memory access pattern is the same as the simplified version used on the exam. Essentially, `getit()` is a simplified version where `getit_bilinear()` is called with x and y set to $0.5, 1.5, 2.5, \ldots$). (A use case for this would be half-pixel motion compensation, assuming the return value from getit() is divided by 4.)

The boundaryflag is used to indicate what happens when we get to the image border. When the boundaryflag is zero we wraparound to the other side of the image (assuming the image is 64x64 large). This is quite useful for repeating signals. (Archetypal example: Texture lookup.)

When boundaryflag is set to one, the processing of non-repeating signal is simplified. This would have been useful in lab 4 for example. If an AGU similar to the AGU above was present in Senior, it would be much easier to search for similar looking blocks close to the image boundary. (You may recall that lab 4 was written in a simplified way so that the pixels close to the borders was ignored due to the difficulty of keeping track of the image border in the performance critical block match kernel.)

Comments

This problem had many different solution proposals. It was fairly common that answers included an adder as follows:

```
result = (x << 5) + (x << 7); // Multiply with 160 without using multiplier
```

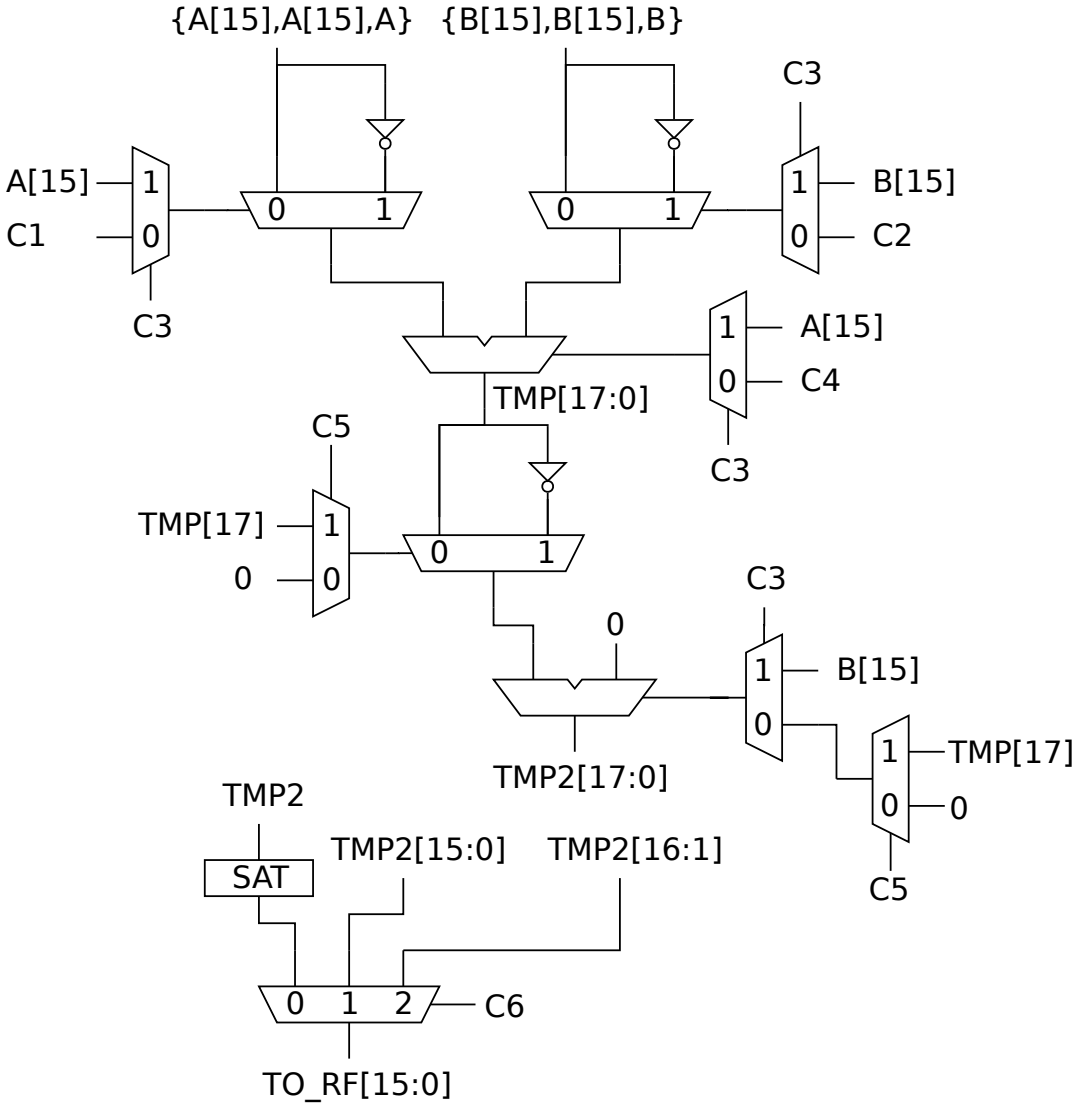
I considered this an acceptable solution, although I much preferred answers where the multiplication with 160 was done outside the loop using the regular MAC unit.

Another common issue is that many answers forgot to include the saturation check for every access. That is, they did something similar to the following pseudo code:

```
if (boundaryflag == 1) {
    t = min(max(t,0),63)
    s = min(max(t,0),63)
}else{
    t = t & 63;
    s = s & 63;
}
tmp = tmem[s+t*64]
tmp += tmem[s+1+t*64]
tmp += tmem[s+(t+1)*64]
return tmp + tmem[s+1+(t+1)*64]
```

Finally, a very common mistake was to forget about the fact that the exercise didn't specify where `ref`, `img`, and `tmem` arrays were located in memory. (Many solutions would only work if these arrays were located at address 0.)

Problem 2



Operation	C1	C2	C3	C4	C5	C6
OP1	0	0	0	0	0	1
OP2	0	1	0	1	0	1
OP3	0	0	0	0	0	0
OP4	0	1	0	1	0	0
OP5	0	0	0	0	0	2
OP6	-	-	1	-	0	0
OP7	0	1	0	1	1	0

```
// Content of SAT:
casez(nisse[17:15])
  3'b000: out = in[15:0];
  3'b111: out = in[15:0];
  3'b0??: out = 16'h7fff;
  3'b1??: out = 16'h8000;
endcase
```

Alternative solution

A pretty neat alternative solution used by a few students was based on the fact that `SAT(ABS(A)+ABS(B))` can be divided into the following cases:

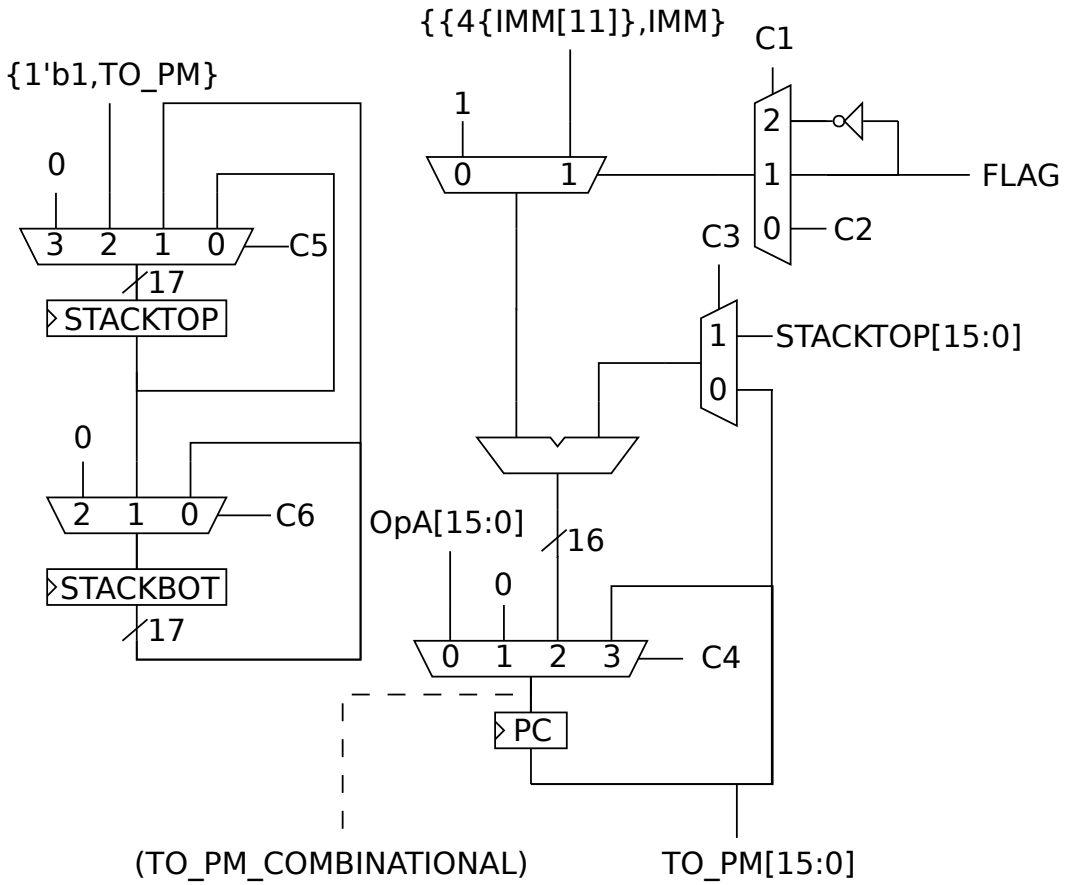
- A positive, B positive: `SAT(ABS(A)+ABS(B)) = SAT(ABS(A+B))`
- A positive, B negative: `SAT(ABS(A)+ABS(B)) = SAT(ABS(A-B))`
- A negative, B positive: `SAT(ABS(A)+ABS(B)) = SAT(ABS(A-B))`
- A negative, B negative: `SAT(ABS(A)+ABS(B)) = SAT(ABS(A+B))`

Comments

A very common issue found in the answers to this problem was that too few guard bits were used. (You require two guard bits to handle a corner cases for OP6 when A and B are set to the smallest possible value (-0x8000). Another common issue was that too many adders were used.

It was also very often unclear whether arithmetic or logic right shift was used for OP5. (However, many people lucked out here since the MSB bit was commonly ignored at the output.)

Problem 3



Note: In a real processor it may be a pretty good idea to take the output to the program memory from the `TO_PM_COMBINATIONAL` signal since the PM most likely has registers at the address inputs. (Either alternative is ok on the exam though.)

Operation	C1	C2	C3	C4	C5	C6
OP1	-	-	-	1	3	2
OP2	-	-	-	3	0	0
OP3	0	0	0	2	0	0
OP4	0	1	0	2	0	0
OP5	1	-	0	2	0	0
OP6	2	-	0	2	0	0
OP7	-	-	-	0	0	0
OP8	0	1	0	2	2	1
OP9	-	-	-	0	2	1
OP10	0	0	1	2	1	2

```
case(operation)
    OP8:  Stack_Error = STACKBOT[16];
    OP9:  Stack_Error = STACKBOT[16];
    OP10: Stack_Error = !STACKTOP[16];
    default: Stack_Error = 0;
endcase
```

Comments

The biggest difficulty with this task was the stack, particularly the Stack_Error signal. Otherwise this task presented no particular difficulties.

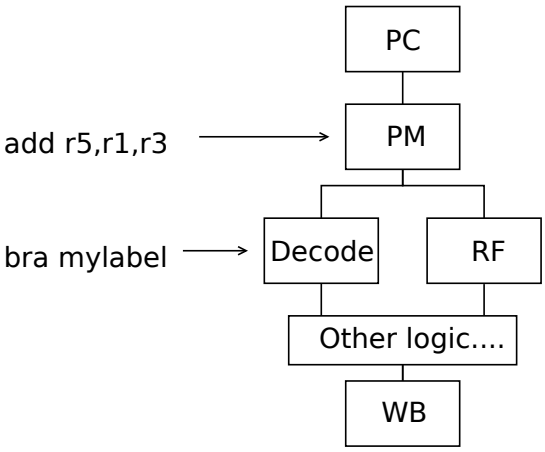
Problem 4

a) If the load instruction has a high latency the loop unrolling performed in alternative 2 will reduce (or eliminate) the impact of the data hazards present in alternative 1.

b) Hardware multiplexing is when you reuse a certain hardware component in different operations (as long as the operations doesn't have to be performed simultaneously). For example, in an ALU the (relatively expensive) adder is reused for the subtraction operation by inverting one of the input operands and setting the carry in to one (using fairly cheap hardware).

c) The fast Fourier transform

d)



Consider the following program:
1: bra mylabel
2: add r5,r1,r3 ; In the branch delay slot

When the decoder has determined that instruction 1 is a branch instruction the processor is already fetching the next instruction. If instruction 2 is executed (irregardless of the branch condition) the instruction is said to be in the branch delay slot.

Comments

The most common issue in these problems was that the answer was not precise enough or unclear.

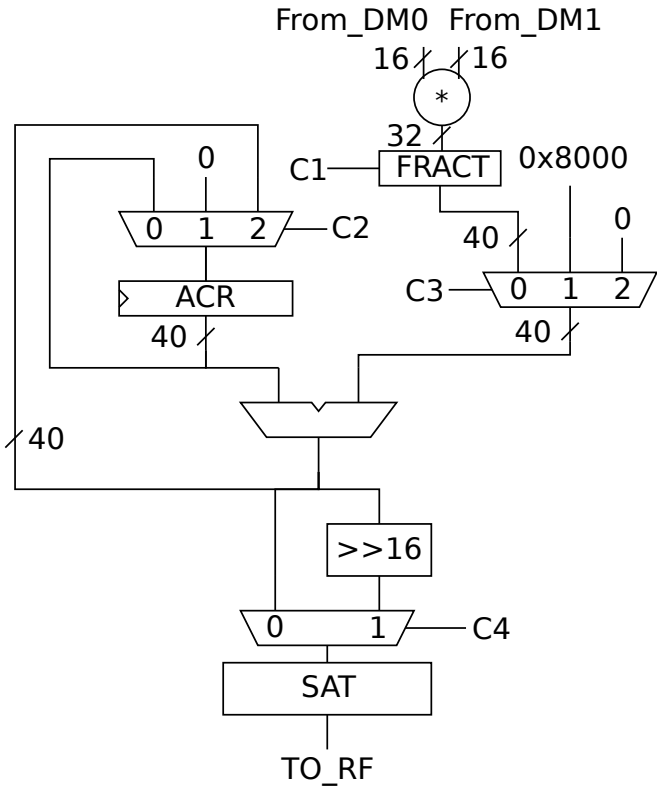
It was also surprising that many answers to part d didn't refer to the figure at all when explaining delay slots. (I thought that a pipeline figure would enable short and concise answers to the question.)

Problem 5

This is a fairly straightforward problem. It is made even easier if you realize that `filt2()` is essentially a fractional MAC unit which is very similar to the first MAC unit that was introduced in the course. You can also trade software complexity for hardware complexity in this exercise by realizing that only one accumulator register is needed.

Assembler source code for `filt1()`

```
filt1:
    clr acr

    repeat 32,endloop
    mac acr,DM0[ar0++],DM1[ar1++]
endloop:

    sat16 r0,acr
    ret     ; 35 clock cycles (excluding ret)


filt2:
    clr acr
    move ar0,r0
    move ar1,r1
    set step0,2

    repeat 20,endloop2
    mac.f acr,DM0[ar0+=step0],DM1[ar1++]
endloop2:
    sat.rnd r3,acr

    add r0,r0,1
    move ar1,r1
    move ar0,r0


    repeat 20, endloop3
    mac.f acr,DM0[ar0+=step0],DM1[ar1++]
endloop3:

    sat.rnd r4, acr

    st DM0[r2],r3
    st DM0[r2+1],r4
    ret                     ; 53 clock cycles (excluding ret)
```



Operation	C1	C2	C3	C4
(nop)	-	0	-	-
clr acr	-	1	-	-
mac acr,DM0,DM1	0	2	0	-
mac.f acr,DM0,DM1	1	2	0	-
sat16 rX,acr	-	0	2	0
sat.rnd rX,acr	-	0	1	1

```
// Content of FRACT:
if(C1) begin
    out[39:0] = {{7{in[31]}},in,1'b0};
end else begin
    out[39:0] = {{8{in[31]}},in};
end
// Content of >> 16:
out[39:0] = { {16{in[39]}},in[39:16] };
// Content of SAT:
if(&~in[39:15] || &in[39:15]) begin
    out[15:0] = in[15:0];
end else begin
    out = { in[39], {15{~in[39]}} };
end
```

Comments

A very common mistake in this exercise was to saturate to the wrong number of bits. If you base your solution completely on the behavior of the pseudo code you would need to saturate from 40 bits to 16 bits. However, many students realized that the shift and saturation in `filt2()` could be combined and therefore created a saturation unit that saturated from 40 to 32 bits. Unfortunately, `filt1()` still requires a 40 bit to 16 bit saturation unit.

Another common issue was that people used inputs and outputs that were not allowed by the constraints. For example, many people used immediates or register file operands in this task. Another common issue was to create a saturation instruction which could store the result directly in memory, even though no memory output was available in the constraints.

Some students also used a 17 bit wide multiplier instead of a 16 bit wide multiplier, even though no instruction actually needed these extra bits (or the ability to change between signed and unsigned multiplication).

Finally, many students figured out that only one accumulator was strictly needed and could thus reduce the amount of hardware. However, not so many students realized that the addressing needs to be adjusted for this case. (Using a step size of two and adding an offset of one for the second iteration.) A similar addressing error was to use normal registers instead of address registers in the mac/convolution instruction.

Results

The grades were distributed as follows:

- Grade 5: 7 students (the best score was 49 points, a truly outstanding result)
- Grade 4: 9 students
- Grade 3: 20 students
- U: 18 students

Points distribution:

- Average score, problem 1: 2.8
- Average score, problem 2: 4.8
- Average score, problem 3: 6.1
- Average score, problem 4: 3.7
- Average score, problem 5: 6.5

Revision history

- v1.0: Initial version
- v1.1: Corrected << 16 to << 6 in control table for Q1. Added clarification that the return instruction was not included in cycle count for Q1. Fixed typo where tmp1 was tmp2 and vice versa in Q1. Fixed typo for AGUOP7 in Q1 in control table. Thanks to Erik Alnervik for noticing these discrepancies.
- v1.2: Fixed typo in saturation for Q5
- v1.3: Removed TRUNC box for Q5 and used SAT directly. Thanks to Rakesh Praveen for pointing out this clarification.