Solutions for TSEA26 exam on 2010-01-14

Andreas Ehliar

October 19, 2010

Note: Some RTL code is written in Verilog in these solutions. It is certainly ok to use VHDL on the exam as well if you prefer that. The exact syntax of the RTL code is also not very important as long as the meaning is understandable.

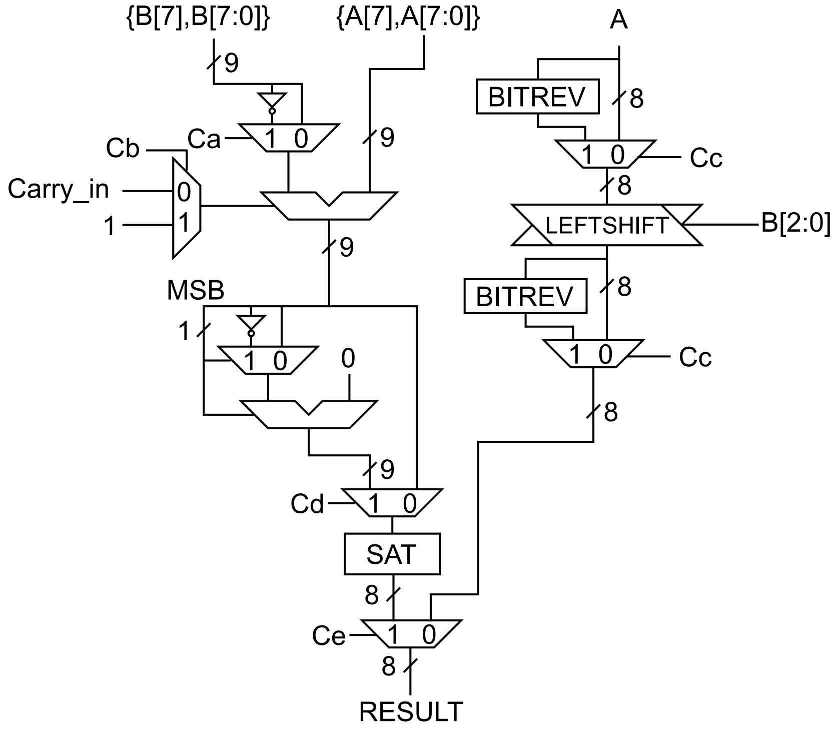
Question 1

a) Maximum possible value: $|0.25| + |-0.75| + |3.0| + |0.5| = 4.5$ This will require 3 guard bits. (Actually, it is not quite 4.5 since we multiply at least one number with $0.11111..._2$.)

b) When decoding a conditional branch it is impossible to know in the decode stage whether the branch should be taken or not since the condition may not be calculated yet. Instead of stalling the pipeline until the condition is calculated a processor may use delay slots instead, that is, fetching and executing one or more instructions after the branch regardless of whether the condition is true or not.

c) In a real time system you can trust the result of static profiling since it is much better to overestimate the amount of time a certain task may take than to underestimate it. Dynamic profiling cannot be trusted to give reliable worst case execution times unless the input data is very carefully constructed.

Question 2



Operation	Ca	Cb	Cc	Cd	Ce
OP0	0	0	-	0	1
OP1	1	1	-	0	1
OP2	1	1	-	1	1
OP3	-	-	0	-	0
OP3	-	-	1	-	0

SAT:
```
if ((in[8:7] == 2'b00) || (in[8:7] == 2'b11)) begin
   out = in[7:0];
end else begin
   out = { in[8], {7 !in[8]}};
end
```

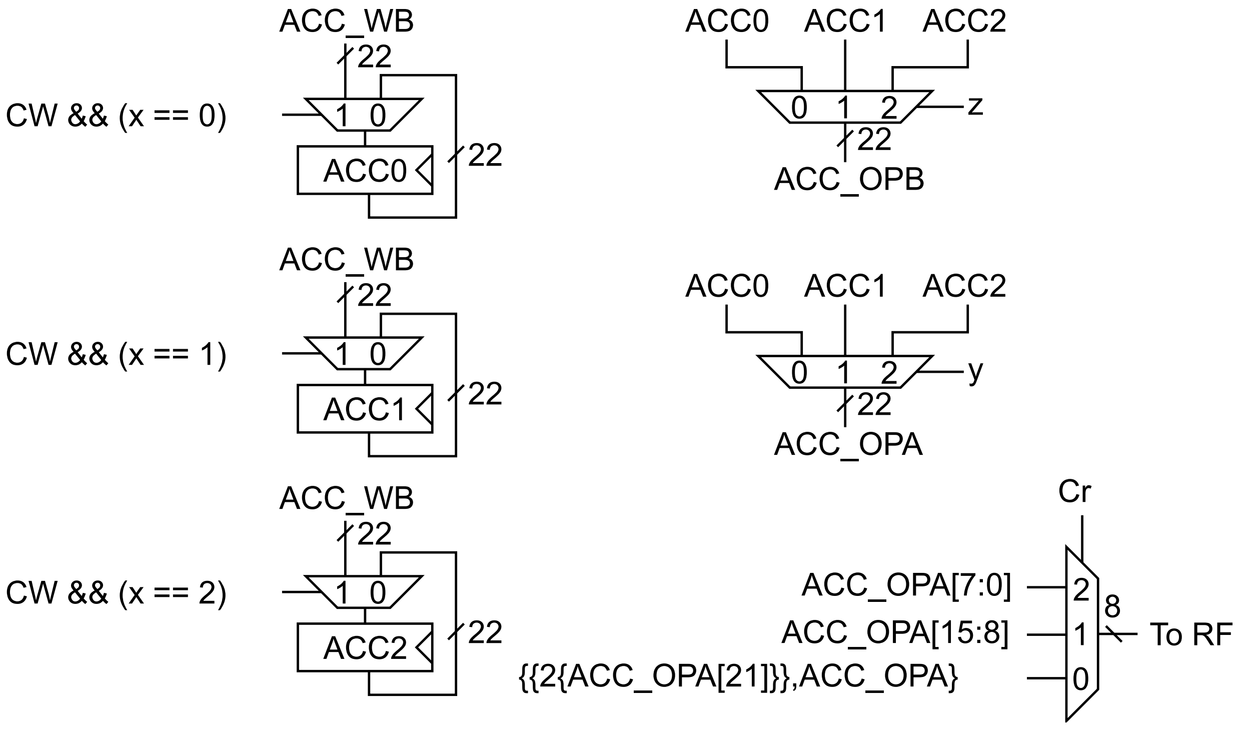
BITREV: `out[7] = in[0]; out[6] = in[1];...`

LEFTSHIFT: `out = in << B[2:0];`

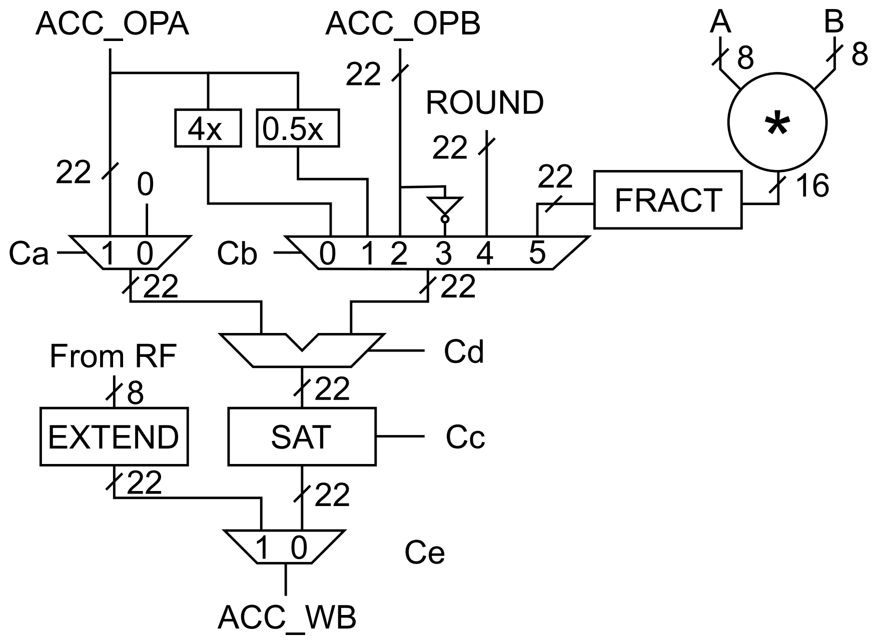
Comments:
Note that this solution utilizes the fact that it is cheaper to use one shifter combined with two bitreverse-operations compared to using two shifters. It also uses the fact that $|A - B| = |B - A|$.

Question 3

Schematic of the accumulators



Schematic of the MAC datapath



Control table

Operation	CW	Cr	Ca	Cb	Cc	Cd	Ce
OP0	0	-	-	-	-	-	-
OP1	1	-	0	5	0	0	0
OP2	1	-	1	5	0	0	0
OP3	1	-	0	0	0	0	0
OP4	1	-	1	1	0	0	0
OP5	1	-	-	-	-	-	1
OP6	1	-	1	4	1	0	0
OP7	1	-	1	2	0	0	0
OP8	1	-	1	3	0	1	0
OP9	0	2	-	-	-	-	-
OP10	0	1	-	-	-	-	-
OP11	0	0	-	-	-	-	-

Content of boxes

FRACT: `out[21:0] = { {5{in[15]}}, in[15:0], 1'b0};`
ROUND: `out[21:0] = 22'h000080;`
EXTEND: `out[21:0] = { {6{in[7]}}, in[7:0], 8'b0 };`

SAT:
```
if (Cc == 1) begin
   if ((in[21:15] == 7'h00) || (in[21:15] == 7'h7f)) begin
      out = {in[21:8],8'b0};
   end else begin
      out = { {7 {in[21]}}, {15 {!in[21]}} };
   end
end else begin
   out = in;
end
```

4x: `out[21:0] = {in[19:0], 2'b00};`
0.5x `out[21:0] = {in[21], in[21:1]};`

Comments:
The tricky thing in this question is that you not only need to know how to design a MAC unit, you also need to apply your knowledge on how to design a register file in order to simplify the part that contains the accumulators.

The exam was not clear about where the rounding would start, so on this exam it would be ok to use another round vector as long as it is relatively reasonable.

Question 4

Required addressing modes for DM0:

- Modulo addressing, postincrement, stepsize 1
- Address register + offset

Required addressing modes for DM1:

- Post increment with step size 1

Pseudo assembler code:

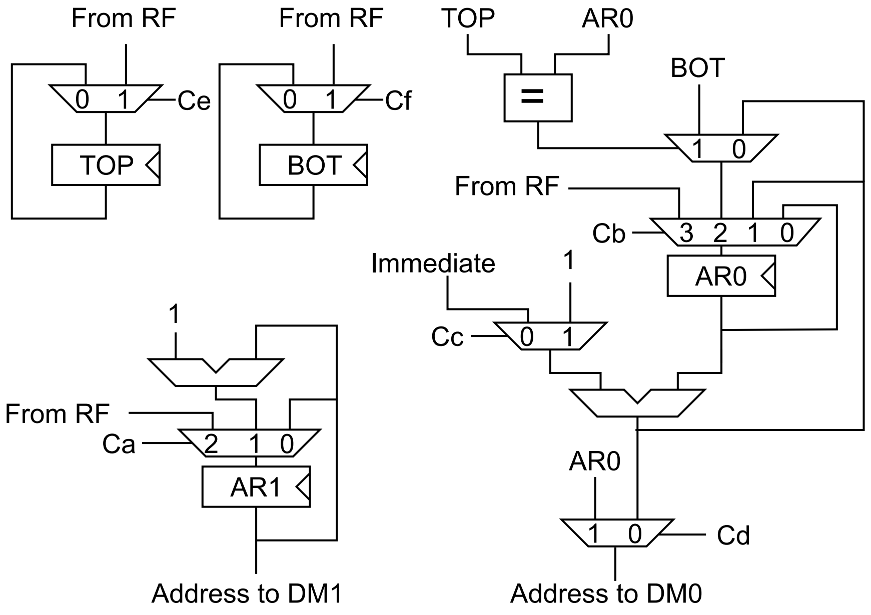
```
FIR_FILTER: add    R2,R2,-1; Fix for quirky hardware (see below)
            move   AR0, R0 ; we assume R0 contains samplesptr
            move   BOT, R1 ; and so on...
            move   TOP, R2
            move   AR1, R3
            repeat 128     ; just the next instruction
            mac    ACC, DM0[AR0%++], DM1[AR1++]
            ret
```

Note that the hardware schematic on the next page is checking for equality before AR0 has been increased by 1 whereas the desired behavior in the given pseudo code is that this check should happen after the samplesptr has been increased. This can be fixed by decreasing the value written into the TOP register by one in the assembler program as seen above. After this fix, the behavior is the same for both the pseudo code and the assembler/hardware implementation in this solution proposal.[1]

```
STORE_VAL:  set   AR0, 59
            ld    R2, dm0[AR0+0]
            move  AR0, R0         ; We assume addr is in R0
            st    DM0[AR0+0], R1  ; We assume value is in R1
            st    DM0[AR0+2], R2  ; Store parameter
            st    DM0[AR0+4], R2
            st    DM0[AR0+8], R2
            st    DM0[AR0+16], R2
            st    DM0[AR0+32], R2
            ret
```

[1]Why do it in this way as opposed to the pseudo code behavior? To reduce the critical path!

AGU Schematics



Control table

Operation	Ca	Cb	Cc	Cd	Ce	Cf
Set AR0	0	3	-	-	0	0
Set AR1	2	0	-	-	0	0
Set TOP	0	0	-	-	1	0
Set BOT	0	0	-	-	0	1
MODULO ADDR	1	2	1	1	0	0
AR0 + Immediate	0	0	0	0	0	0

Note that MODULO ADDR also includes post increment addressing for AR1 to DM1.

Comments:
This question not only examines whether you know how to design an AGU, it also examines whether you realize which addressing modes that are used in a more or less realistic piece of source code. Also, in case of the `FIR_FILTER` code, I want to make sure that you realize that it is necessary to set the address registers.

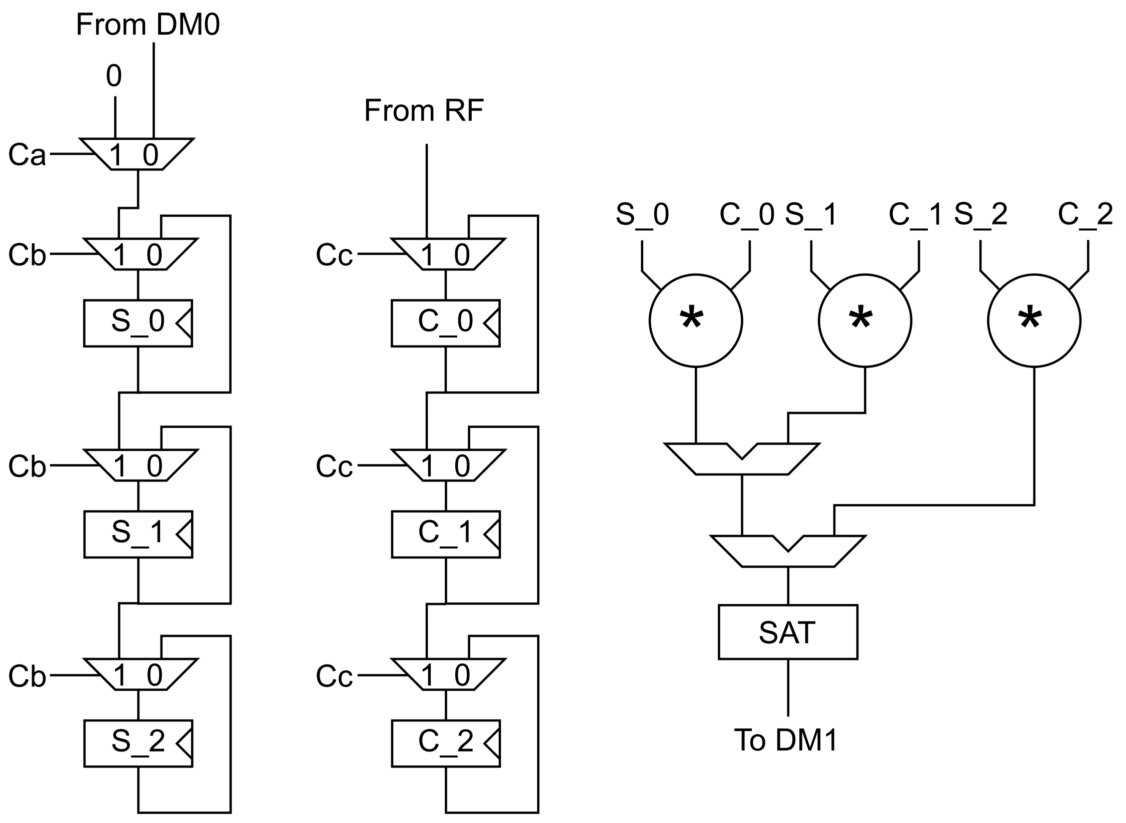
The STORE_VAL function is not modeled on any particularly important computational kernel. It is intended to showcase a few different addressing modes, but in the end, all that is required is one addressing mode such as base register plus offset. For example, the absolute addressing of `dm[59]` can easily be handled by setting the address register to 59 and then using the offset of 0. On the other hand, using just register indirect addressing would not be possible within the alloted number of clock cycles. (Other solutions with only one addressing mode is also possible!)

Question 5

How to modify the pipeline:

You will need to modify the MAC unit and make sure that it can send the result back to DM1. (There is no need to modify the AGU unit as the addressing used in this code snippet are standard post increment addressing modes that any normal DSP processor would have.)

Schematics



Control table

Operation	Ca	Cb	Cc
NOP	-	0	0
CLEARSAMPLES	1	1	0
MOVE COEFF0, REG	-	0	1
FIR_3	0	1	0

```
INIT_FIR:
    CLEARSAMPLES
    CLEARSAMPLES
    CLEARSAMPLES
    move COEFF0, R5    ; val5
    move COEFF0, R4    ; val4
    move COEFF0, R3    ; val3
    move AR0, R1       ; In the AGU
    move AR1, R2       ; In the AGU
    ret
```

Comments:
This question can obviously be answered in many different ways. The main reason for this question is to see whether you can reason about how to add non-standard instructions (ASIP instructions if you will) to a normal DSP processor. I am particularly interested in whether you can reason about modifications to the pipeline, like for example using the memories in non-standard ways.