Solution proposal for the TSEA26 exam on 2011-01-12 (v1.2)

Andreas Ehliar

October 25, 2013

Note about this solution proposal: This solution proposal is quite long, but it includes quite a few notes about the included solutions (including alternative solutions for some of the exercises). It is of course not required to come up with various alternative solutions when writing the exam. (In fact, it is likely to be a bad idea to do so since it increases the likelihood of mistakes not to mention the time it takes to do so.)

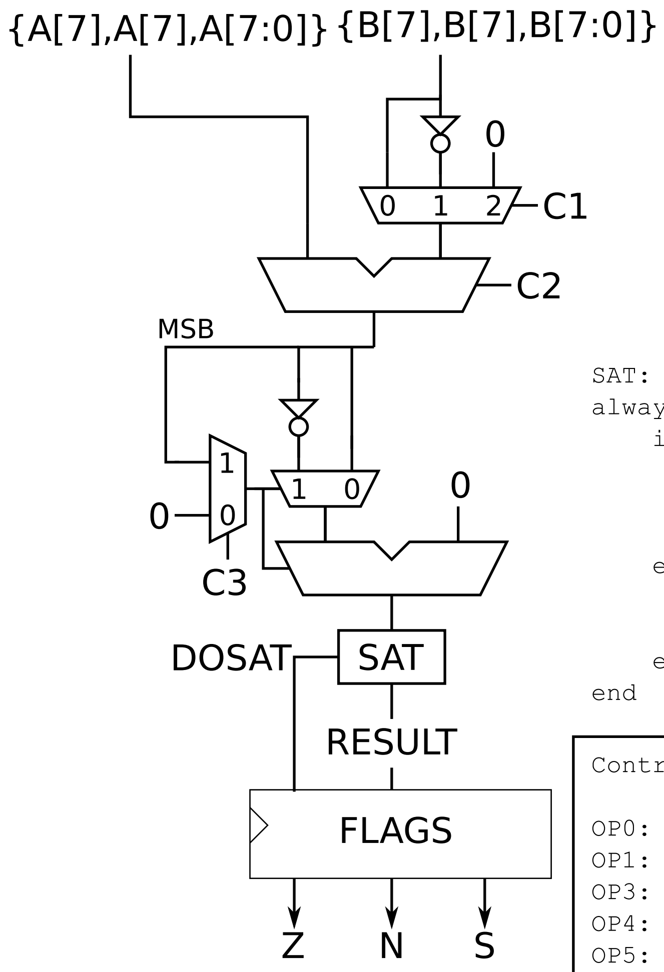
Question 1: General knowledge

a) The unit doesn't contain an accumulator register. (Instead it contains a combinational loop.)

b) The memory is the largest component and the multiplexer is the smallest component. (If you claimed that the multiplier was the largest component and the multiplexer the smallest component you would get 0.5 points on this question.) All other answers would lead to 0 points on this question.

c) The most important case is when saturating the number 10000_2 (-16), as the number 16 will not fit in the same number of bits (assuming two's complement representation is used for the answer as well).

Question 2: ALU



```
Flags:
always @(posedge clk) begin
    if(OP == 7) begin
        S <= 0;
    end else if(OP != 0) begin
        S <= S || DOSAT;
        N <= RESULT[7];
        Z <= RESULT == 0 ? 1 : 0;
    end
end
```

```
SAT:
always @* begin
    if( (in[9:7] != 3'b000) ||
        (in[9:7] != 3'b111)) begin
        RESULT = {in[9], {7{~in[9]}} };
        DOSAT = 1;
    end else begin
        RESULT = in[7:0];
        DOSAT = 0;
    end
end
```

Control table

	C1	C2	C3	
OP0:	-	-	-	No operation
OP1:	0	0	0	SAT(A+B)
OP3:	1	1	0	SAT(A-B)
OP4:	2	0	1	SAT(ABS(A))
OP5:	0	0	1	SAT(ABS(A+B))
OP6:	1	1	1	SAT(ABS(A-B))
OP7:	-	-	-	Set S=0

Note: The operation numbering is odd since OP2 was forgotten in the exam.

List of common mistakes:

- The absolute operation doesn't work correctly
- It is not clear whether the flags are registers, latches, or just combinational logic.
- There is a combinational loop in the design
- The hardware is not able to perform saturation correctly for the case where A and B are -128 and we perform SAT(ABS(A+B)).
- The saturate flag does not work correctly
- The saturate flag is not "sticky". That is, the saturation flag can be reset to 0 without running the OP6 instruction.

Question 3: AGU

In this question we will need the following AGU operations:

- Register + offset for both memories
- Post increment for both memories simultaneously
- Post increment for DM1, modulo-like addressing with variable step-size for DM1
- Special indirect addressing mode where DM0 is addressed using post increment mode and the output of DM0 is sent as address into DM1

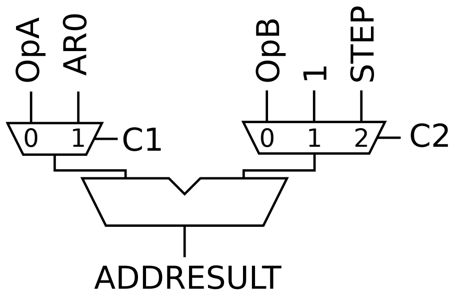
The reason that I call it modulo-like instead of modulo addressing is that the C-code in the original exercise is not a true modulo addressing mode with variable step size. The version from the exam is actually slightly easier to implement as seen below as no subtraction is required:

```
// Code from the exam
for(i=0; i < 128; i = i + 1)
   tmp += DM0[ptr0+=step] * DM1[ptr1++]
   if (ptr0 > end) then
      ptr0 = start
   endif
endfor

// Real modulo addressing with variable step-size
for(i=0; i < 128; i = i + 1)
   tmp += DM0[ptr0+=step] * DM1[ptr1++]
   if (ptr0 > end) then
      ptr0 = ptr0 - (end - start)
   endif
endfor
```

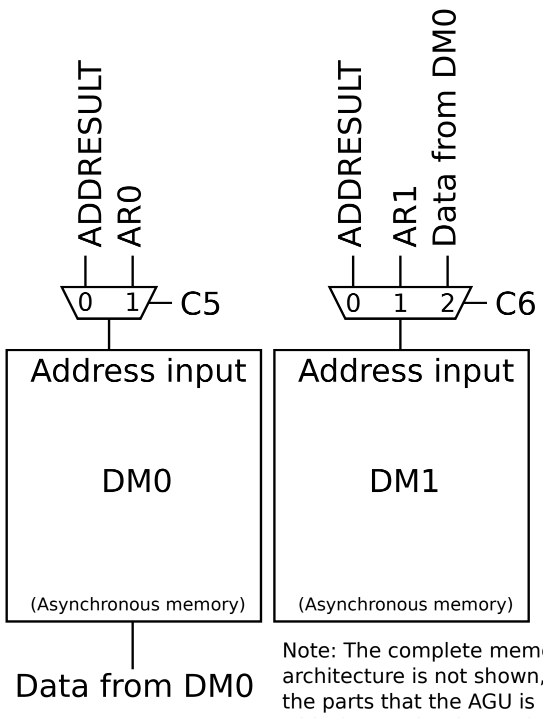
Also, note that the special indirect addressing mode is not very realistic, especially if asynchronous memories are used. However, a more realistic (but more complicated) solution proposal is shown later in this document which doesn't require such an addressing mode.

Part B: AGU schematic and control table

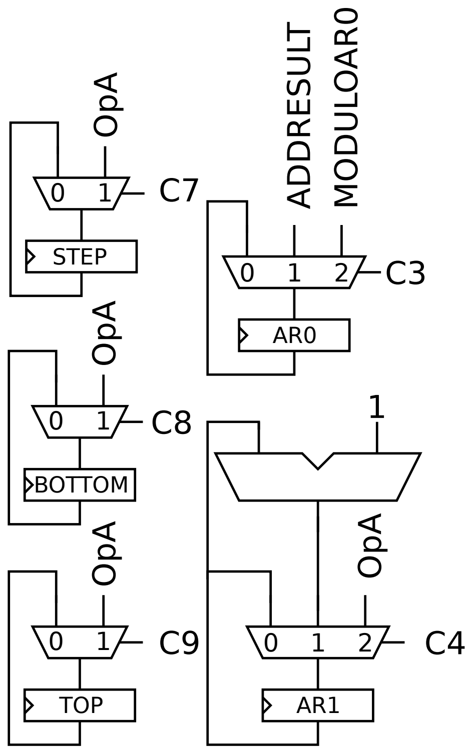
NOTE: All signals are 16 bit wide (except for mux control signals.)



```
// Creating the MODULOAR0 signal
always @* begin
    // (Unsigned comparison)
    if(ADDRESULT > TOP) begin
        MODULOAR0 = BOTTOM;
    end else begin
        MODULOAR0 = ADDRESULT;
    end
end
// Cost: One comparator and one
// multiplexer
```



Note: The complete memory architecture is not shown, only the parts that the AGU is involved with. (E.g. write data/write enable signals are missing.)



Control table:

Operation	C1	C2	C3	C4	C5	C6	C7	C8	C9
Set STEP	-	-	0	0	-	-	1	0	0
Set BOTTOM	-	-	0	0	-	-	0	1	0
Set TOP	-	-	0	0	-	-	0	0	1
Set AR0	0	0	1	0	-	-	0	0	0
(Note: OpB should be set to 0 here)									
Set AR1	-	-	0	1	-	-	0	0	0
DM0: Register + offset	0	0	0	0	0	-	0	0	0
DM1: Register + offset	0	0	0	0	-	0	0	0	0
Post increment for both	1	1	1	1	1	1	0	0	0
DM0: Modulo, DM1: Postincr.	1	2	2	1	1	1	0	0	0
Special indirect mode	1	1	1	1	1	2	0	0	0

Part C: Assembly code

```
filter:
        set AR0, r0
        set AR1, r1
        set STEP, r2  ; Assume stepsize is in r2
        set START, r3 ; Assume start is in r3
        set END, r4   ; Assume end is in r4
        CLR ACR
        repeat 128
        conv ACR,DM0[AR0%++], DM1[AR1++]
        ret

interleaver:
        load    r1, DM0[r0+r31]
        load    r2,DM0[r0+33]
        set     AR0, r1
        set     AR1, r2

        repeat  256, endloop
        load    r0,DM1[DM0[AR0++]] ; Special indirect addr-mode
        store   DM1[AR1++], r0
endloop:
        ret
```

A more realistic solution

This exercise was made quite easy by the fact that asynchronous memories were allowed. This allowed us to easily create the special indirect addressing mode where the output from DM0 was sent as an address into DM1. In a real processor synchronous memories would be used here instead (it is hard to create a large and efficient asynchronous memory).

If this architecture would be created with a synchronous memory, it would be tough to avoid data and structural hazards. However, there is another way to allow the interleaver() function to execute in less than 700 clock cycles. We can make an instruction which stores data into DM1 while at the same time loading data from DM0. This can be done without violating the constraints placed on us in the exercise (e.g. we are only allowed to use single port memories and the register file has two read ports and one write port).

```
// A more realistic interleaver() function.
interleaver:
        load   r1,DM0[r0+31]
        load   r2,DM0[r0+33]
        // Might need a NOP here depending on how the pipeline looks like
        set    AR0, r1
        set    AR1, r2


        // The loop is unrolled four times to ensure that data hazards
        // cannot occur. (Depending on the pipeline it could be possible to
        // unroll it fewer times, but four times is probably enough for
        // most reasonable pipelines.)

        // Note the need for a prologue and epilogue to the loop.
        load   r0,DM0[AR0++]
        load   r1,DM0[AR0++]
        load   r2,DM0[AR0++]
        load   r3,DM0[AR0++]

        repeat 63, endloop
        load   r0,DM1[r0]
        load   r1,DM1[r1]
        load   r2,DM1[r2]
        load   r3,DM1[r3]
        ; The next instruction stores r0 into DM1
        ; and loads r0 from DM0
        loadstore r0,DM0[AR0++], DM1[AR1++], r0
        loadstore r1,DM0[AR0++], DM1[AR1++], r1
        loadstore r2,DM0[AR0++], DM1[AR1++], r2
        loadstore r3,DM0[AR0++], DM1[AR1++], r3
endloop:

        load   r0,DM1[r0]
        load   r1,DM1[r1]
        load   r2,DM1[r2]
        load   r3,DM1[r3]
        store  r0,DM1[AR1++]
        store  r0,DM1[AR1++]
        store  r0,DM1[AR1++]
        store  r0,DM1[AR1++]

        ret
```

List of common errors

- It is not possible to execute the interleaver() function in less than 700 clock cycles.
- Forgetting about the step, top, and/or bottom register (including forgetting to set these registers from the assembly code)
- The answer contains no hint as to how DM0 and DM1 can be accessed simultaneously
- The hardware implements pre-increment addressing while the assembly code is written using post increment addressing

Question 4: PFC

There are many different ways to solve this exercise. It is for example possible to trade off hardware complexity against software complexity. This is also the reason that no proposed distribution of points is given to the a and b part of this exercise. (That is, a solution which depends mostly on software and contains few instructions would get points mostly for part b whereas a solution which contains many different branch instructions and very straight forward software would get points mostly for part a.)

Required PFC instructions

Assumptions: It is actually not stated in the exam how wide PC should be, so we assume a 16 bit PC in this exercise.

- **jump immediate** Unconditional branch. `PC = PC + 1;`. On next clock cycle: `PC = immediate`
- **jump.lte immediate** Branch on less than or equal. `PC = PC + 1`. On next clock cycle: `PC = PC + 1`. On next clock cycle: `if(N || Z) then PC = immediate else PC = PC + 1;`
- **jump.eq immediate** Branch on equal. `PC = PC + 1`. On next clock cycle: `PC = PC + 1`. On next clock cycle: `if(Z) then PC = immediate else PC = PC + 1;`
- **repeat immediate** Repeat next instruction: `loopcounter = immediate; PC = PC + 1;`
- **jal reg, immediate** Jump and link. `PC = PC + 1`. On next clock cycle: `reg = PC + 1; PC = immediate`
- **jump OpB** Indirect jump. `PC = PC + 1`. On next clock cycle: `PC = PC + 1`.On next clock cycle: `PC = PC + 1`. On next clock cycle: `PC = OpB`
- **Normal instruction** `if (loopcounter != 0) then loopcounter-- else PC = PC + 1;`

Note 2: `jal reg, immediate` and `jump OpB` are used in order to handle call to subroutine and return from subroutine. (This allows us to do some tricky things involving conditional branches to subroutines where the return register (r31) is set in the delay slot.) (Actually, we could probably manage to write these programs without having a `jal` instruction at all, but at a slight performance loss.)

Assembly code

```
// In this code I assume the calling convention that r0-r15, all
// address registers, all accumulator registers, and the status
// register can be modified by a called function. However, r16-r30
// must not be modified by a subroutine. (If it needs to be modified
// it needs to be stored first so that it can be restored later on.)
// Finally, r31 contains the return address.

// Finally: Keep track of the delay slots when reading this code!
// (Unconditional branches have 1 delay slot, conditional branches
// have 2 delay slots, and register indirect branches have 3 delay
// slots!)

// (main doesn't need to store r31 as it will never return)

main:
    jal r31, get_packet
    nop

    cmp r0,0           ; Set flags as if we used r0 - 0
    jump.lte anerror
    nop
    nop

    cmp r0,9
    jump.lte worker
    set r31, before_update_outputs
    nop

    cmp r0, 59
    jump.lte anerror
    nop
    nop

    jal r31, guiworker
    nop

    jump before_update_outputs
    nop
```

```
anerror:
    jal r31, logerror
    nop

before_update_outputs
    jal r31, update_outputs
    nop
    jump main
    nop


worker:
    // Store r31 on the stack
    push r31

    jal r31, get_packet_operations
    nop
    cmp r0, 0

    jump.eq  dosum
    nop
    nop

    cmp r0, 1
    jump.eq dodiff
    nop
    nop

    // Tail call optimization
    pop r31
    jump logerror
    nop

dosum:
    jal r31, sum
    nop

    pop r31

    jump r31
    store DM0[95], r0
    nop
    nop

dodiff:
    jal r31, diff
    nop
    pop r31
    jump r31
    store DM0[96], r0
    nop
    nop

sum:
    push r31
    jal r31, get_current_buffer
    nop
    pop r31
    move AR0, r0

    clr ACR0

    repeat 100
    add ACR0, DM0[AR0++]

    jump r31
    sat ACR0
    move r0, ACR0
    nop



diff:
    push r17
    push r16
    push r31


    jal r31, get_current_length
    nop

    jal r31, get_current_buffer
    move r16, r0

    jal r31, get_previous_buffer
    move r17, r0

    ; r16 now contains the current length
    ; r17 contains ptr1
    ; r0 contains ptr2

    move AR0, r17
    move AR1, r0

    clr ACR0
    add r0,r0,-1

diffloop:
    jump.neq diffloop
    add ACR0, ABS(DM0[AR0++]-DM1[AR1++])
    add r0,r0,-1

    pop r31

    jump r31   ; And return from subroutine
    move r0, ACR
    pop r16
    pop r17
```

Common errors

Not that many people attempted to solve this exercise, but the most common problem for those who tried to do so is that no detail description of each instruction was present. While it doesn't have to be written in exactly the same way as written above, it should contain the same kind of information.

Besides that, it was common with mistakes in the assembler code, but there wasn't any mistake there that really stood out.

Alternative solutions

I've tried to keep the solution above fairly "mainstream" with few instructions that wouldn't be found on a normal processor. However, it is possible to solve it in a few different ways. For example, the solution above contains a "jump and link"-instruction to handle call to subroutines. This means that the solution outlined above requires quite a bit more support from the software, but requires easier hardware. A more traditional solution would use a call/ret instruction pair which would automatically store and load the return address from a stack (either in memory or a separate hardware stack).

Another way that could be used to simplify the assembly code a lot is to use a conditional call instruction. This is not commonly found in real processors, but it could have simplified main() and worker() quite a lot.

Finally, a repeat instruction with support for more than one instruction could have been used to simplify the diff() instruction. However, such a repeat instruction would need to read the number of iterations from a register (which would make it troublesome to implement in actual hardware when very few iterations must be supported). (But there is nothing in the constraints listed in the exercise that would prohibit such a repeat instruction.)

Question 5

Adding 5 guard bits

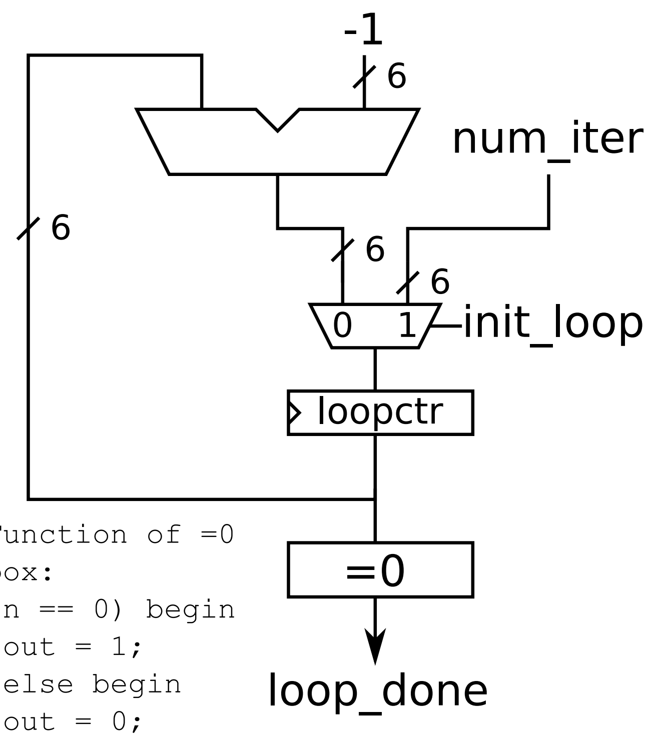
```
out = { {5{in[22]}}, in[22:0]};
```

Register file

I've included a Verilog version here. To see what it would look like in a schematic, please see the book.

```
module regfile(input wire clk, we
               input wire [1:0] opa_sel, opb_sel, wb_sel
               input wire [7:0] wb_data
               output reg [7:0] opa, opb);


    reg [7:0] rf[2:0];

    always @* begin
        opa = rf[opa_sel];
        opb = rf[opb_sel];
    end

    always @(posedge clk) begin
        if(we) begin
            rf[wb_sel] <= wb_data;
        end
    end

endmodule
```

Loop controller



```
// Function of =0
// box:
if(in == 0) begin
    out = 1;
end else begin
    out = 0;
end
```

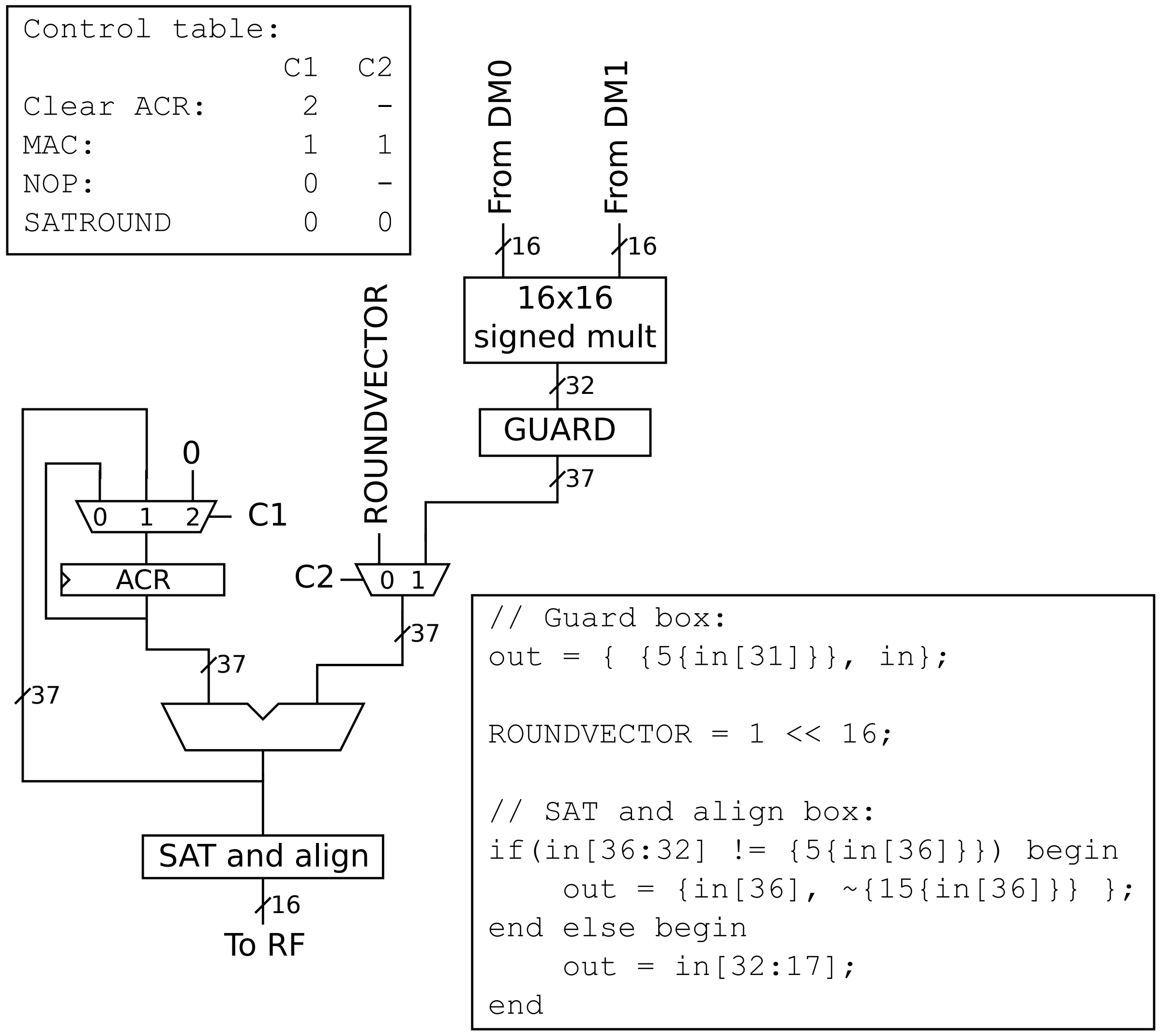
Reality check

The schematic for this part is very simple. The exercise was also intentionally designed in such a way that some subtleties that would be present in a real processor were lost.

Most importantly, it will take a couple of clock cycles before a repeat instruction is decoded, and one or more clock cycles to notify the PFC unit about this fact. Supporting a repeat loop with say 2 iterations could therefore be fairly difficult (and wouldn't make sense anyway as it would take the same amount of space to unroll it). This is the reason that the exercise specifies that the repeat Instruction should be able to handle between four and sixty iterations.

Due to these effects, it may also be necessary to specify a non-obvious value to the `num_iter` input when starting a loop. For example, say that you want to repeat an instruction six times. However, due to pipeline effects, the `num_iter` value sent to the PFC unit might be three instead of six. (As the instruction has already started to be repeated by the instruction decoder at this point.)

MAC unit



```
Control table:
                 C1  C2
Clear ACR:        2   -
MAC:              1   1
NOP:              0   -
SATROUND          0   0
```

```
// Guard box:
out = { {5{in[31]}}, in};

ROUNDVECTOR = 1 << 16;

// SAT and align box:
if(in[36:32] != {5{in[36]}}) begin
    out = {in[36], ~{15{in[36]}} };
end else begin
    out = in[32:17];
end
```

The most important part of this exercise was to see if you could handle the scaling necessary to output a number in Q2.13 format based on inputs in Q0.15 format. Besides this part, it shouldn't be hard to design a MAC unit this simple.

Note: You could make a good case for a 36 bit accumulator here as well, based on the opinion that the MSB bit of the signed multiplication could be called a guard bit. So I would accept such a solution as well here.

Note 2: There is some widespread confusion whether a number in the QX.Y format is X+Y bits wide (sign bit is included) or X+Y+1 bits wide (sign bit is not included). However, as I state in the exam that the data is in 16 bit Q0.15 data, there is only one reasonable interpretation, that is, one sign bit, followed by a radix point, followed by fifteen more bits.

Common errors for question 5

- The most common bug was that there was some mistake in handling the scaling of the accumulator when reading out the result. Many students didn't seem to notice that the output shouldn't be in the same kind of fixed point format as the inputs. Similarly, even for the students who actually noticed this, it was quite common that the round-vector didn't take this into account.
- An extra adder was used for the rounding instead of reusing the main adder and using a roundvector
- Drawing a box (e.g. a box named FRACTION) and not explaining the contents of the box.
- (There was no really common errors for part A-D.)

Revision history

- V1.0: Initial version
- V1.1: Fixed control table and ABS operation for Q2. Thanks to Niclas Sjökvist for pointing this out. Also clarified opcodes in control table for Q2.
- V1.2: Saturate to Q2.13 instead of Q1.14 in 5e. Thanks to Jonas Henriksson for pointing this out.