Examination
Design of Embedded DSP Processors, TSEA26

Date	2015-10-30
Room	TB
Time	8-12
Course code	TSEA26
Exam code	TEN1
Course name	Design of Embedded DSP Processors
Department	ISY
Number of questions	5
Number of pages (including this page)	15
Course responsible	Andreas Ehliar
Teacher visiting the exam room *Phone number during the exam time*	Andreas Ehliar
Visiting the exam room	About 9 and 11
Course administrator	Gunnel Hässler
Permitted equipment	None, besides an English dictionary
Grading	**Points** **Swedish grade** 41-50 5 31-40 4 21-30 3 0-20 U

Important information:

- You can answer in English or Swedish.
- **When designing a hardware unit you should attempt to minimize the amount of hardware.** (Unless otherwise noted in the question.)
- The width of data buses and registers must be specified unless otherwise noted. Likewise, the alignment must be specified in all concatenations of signals or buses. When using a box such as *"SATURATE"* or *"ROUND"* in your schematic, you must (unless otherwise noted) describe the content of this box! (E.g. with RTL code). You can assume that all numbers are in two's complement representation unless otherwise noted in the question.
- In questions where you are supposed to write an assembler program based on pseudo code you are allowed to optimize the assembler program in various ways as long as the output of the assembler program is identical to the output from the pseudo code. You can also (unless otherwise noted in the question) assume that hazards will not occur due to parts of the processor that you are not designing.

Good luck!

Question 1: MAC(10p)

Draw a schematic and a control table for a MAC unit with the following operations:

- OP0: NOP
- OP1: ACRx = 0
- OP2: ACRx = OpA * OpB
- OP3: ACRx = ACRy + OpA * OpB
- OP4: ACRx = ACRy + ACRz
- OP5: ACRx = ACRy - ACRz
- OP6: ACRx = ABS(ACRy)
- OP7: ACRx = SCALE(ACRy, OpB)
- OP8: ACRx = ACRy + SCALE({ OpA, 24'h0 } , OpB)
- OP9: RF = SAT(ROUND(SCALE(ACRy, OpB)))

Constraints:

- Your MAC unit should have 4 accumulator registers. Each accumulator register is 40 bits wide
- Scaling and rounding should be under total user control based on the OpB input
- res = SCALE(foo, OpB) has the following behavior:
  - If OpB is 24: res = foo $/2^0$
  - If OpB is 23: res = foo $/2^1$
  - ...
  - If OpB is 0: res = foo $/2^{24}$

Inputs/outputs:

- OpA, OpB: 16 bit inputs
- x,y,z: 2 bit wide inputs from the instruction decoder that selects the appropriate accumulator register
- To_RF: 16 bit output to the general purpose register file
- And of course, whatever clock signals and control signals you deem necessary.

Question 2: PFC(8p)

Draw a schematic and a control table for a PFC unit with the following features:

- Starts executing at address 0x4000 when reset
- Subroutines with absolute address (return address should be saved in DM0) **Hint:** Make sure the saved return address is correct!
- Conditional branches with PC-relative address
  - The following conditional branches should be supported: (jump if negative, jump if positive, jump if equal, jump if not equal)
- Unconditional branches with absolute address
- Single statement repeat loop (between 5 and 250 iterations)

Constraints:

- The PC should be 16 bits wide
- PC-relative address use a 7 bit offset
- You can use whatever inputs and outputs that you think are necessary for this PFC.

Question 3: AGU(13p)

Create an AGU suitable for the following pseudo code:

```
function sum_structs(ptr1)
    int sum = 0;

    for i=0; i < 16; i = i + 1
        sum = sum + DM0[ptr1++]
        tmp = DM0[ptr1++]
        if tmp != 0
        then
            sum = sum + sum_structs(tmp)
        endif
    endfor

    return sum
endfunction


function postprocess_fft()

    maxval = 0
    maxidx = 0

    for i = 0; i < 256; i = i + 1
        idx2 = 0
        tmp = 128

        for j=1; j < 256; j = j  << 1
            if i & j              // Bitwise and
            then
                idx2 = idx2 | tmp // Bitwise or
            endif
            tmp = tmp >> 1  // Logic right shift
        endfor

        tmp = abs(fftoutput[idx2])
        if tmp > maxval
            maxidx = i
            maxval = tmp
        endif
    endfor

    return maxidx

endfunction
```

Constraints:

- The sum_structs function needs to execute in at most 200 clock cycles in the worst case. (Excluding any additional recursive function calls.)
- The postprocess_fft function needs to execute in at most 2600 clock cycles.
- In addition to whatever address registers you need, you need to have an extra address register which is solely intended as a stack pointer.
- There is an input/output available to the memory that allows you to write special purpose registers directly to the memory and read them from the memory. (Which might be useful if you for example would need to implement support for instructions such as `push ar0`.)

Inputs/outputs:

- OpA: 16 bit input from the register file
- To_RF: 16 bit output to the register file
- DM_Addr: Address to DM0
- From_DM: 16 bit input from DM0 (where the stack is located)
- To_DM: 16 bit output from DM0 (where the stack is located)
- And of course, whatever clock signals and control signals you deem necessary.

(a) (6p) Decide an instruction set for your AGU and translate the pseudo code above into assembler

(b) (7p) Draw a schematic and a control table for your AGU.

Question 4: General knowledge(5p)

(a) (1p) Suppose that you are designing a DSP processor that should have a memory bandwidth to the data memory of 64 bits / second. Discuss the advantage/disadvantage of using either a single 64 bit wide memory or alternatively, two 32 bit wide memories.

(b) (2p) Many DSP processors have an S flag which is set whenever saturation occurs in the ALU (or MAC). Please explain why this flag typically is sticky (i.e. it will remain set until a special clear S flag instruction is run).

(c) (2p) When writing an instruction set simulator it is often important that this simulator accurately handles hazards. Briefly discuss how data hazards can be modelled in an instruction set simulator.

Question 5: ALU(14p)

Your task is to design an ALU suitable for the following pseudo code:

```
dct_transform:
Statement  // Start of transform. You can assume that data_0 to data_7 is in
Number     // registers already. (You can also assume that all constants have
           // been loaded into registers.)

1          tmp0 = data_0 + data_7
2          tmp7 = data_0 - data_7
3          tmp1 = data_1 + data_6
4          tmp6 = data_1 - data_6
5          tmp2 = data_2 + data_5
6          tmp5 = data_2 - data_5
7          tmp3 = data_3 + data_4
8          tmp4 = data_3 - data_4

9          tmp10 = tmp0 + tmp3
10         tmp13 = tmp0 - tmp3
11         tmp11 = tmp1 + tmp2
12         tmp12 = tmp1 - tmp2

13         data_0 = (tmp10 + tmp11)
14         data_4 = (tmp10 - tmp11)

           // Hint: You can assume that there is a sat.rnd instruction in the
           // MAC unit that reads an acr register, adds 4096 and right shifts
           // the result arithmetically. (All right shifts in this code are
           // supposed to be arithmetic.)
15         data_2 =  (tmp12 * CONSTANT0 + tmp13* CONSTANT1  + 4096 )  >> 13
16         data_6 = (-tmp12 * CONSTANT2 + tmp13 * CONSTANT3 + 4096 ) >> 13

17         z1 = tmp4 + tmp7
18         z2 = tmp5 + tmp6
19         z3 = tmp4 + tmp6
20         z4 = tmp5 + tmp7
21         z5 = (z3 + z4)* CONSTANT4

22         z1 = z1* CONSTANT9
23         z2 = z2* CONSTANT10
24         z3 = z3* CONSTANT11
25         z4 = z4* CONSTANT12

26         z3 = z3 + z5
27         z4 = z4 + z5

28         tmp4 = tmp4 * CONSTANT5
29         tmp5 = tmp5 * CONSTANT6
30         tmp6 = tmp6 * CONSTANT7
31         tmp7 = tmp7 * CONSTANT8
32         data_7 =  (tmp4 + z1 + z3 + 4096) >> 13
33         data_3 =  (tmp6 + z2 + z3 + 4096) >> 13
34         data_5 =  (tmp5 + z2 + z4 + 4096) >> 13
35         data_1 =  (tmp7 + z1 + z4 + 4096) >> 13
           // Finished. data_0 to data_7 should be located in registers by this point.
36         return
```

```
function sortvals(ptr1, ptr2)
    repeat 100
        x0 = DM0[ptr1]
        x1 = DM0[ptr1+1]
        x2 = DM0[ptr1+2]
        idx0 = 0
        idx1 = 1
        idx2 = 2
        if x1 < x0
        then                  // Where swap(x0,x1) expands to the following code:
            swap(x0,x1)       //      tmp = x0
            swap(idx0,idx1)   //      x0 = x1
        endif                 //      x1 = tmp
        if x2 < x1
        then
            swap(x2,x1)
            swap(idx2,idx1)
        endif
        if x1 < x0
        then
            swap(x0,x1)
            swap(idx0,idx1)
        endif
        DM0[ptr1++] = x0
        DM0[ptr1++] = x1
        DM0[ptr1++] = x2
        DM0[ptr2++] = idx0
        DM0[ptr2++] = idx1
        DM0[ptr2++] = idx2
    endrepeat
endfunction
```

Constraints:

- The dct_transform should be finished in at most 42 clock cycles (excluding the return instruction).
- You should not include a multiplier in your ALU, your assembler code should instead use the mul, mac, and sat/round instructions that are available in the MAC unit when this is appropriate. You can assume that the MAC unit has 8 accumulator registers.
- **You do not need to translate the entire dct_transform function into assembler code. However, you need to translate statement number 9 to 15 completely into assembler language.** You also need to write a motivation how you are using your instruction set to reach a clock cycle count of less than 42 clock cycles. (E.g., I'm using the xyzzy instruction for statement X to Y with a total cost of 2 clock cycles, etc.)
- The sortvals function should be finished in at most 2400 clock cycles.
- All operations that could cause overflows in the programs above should use saturation.

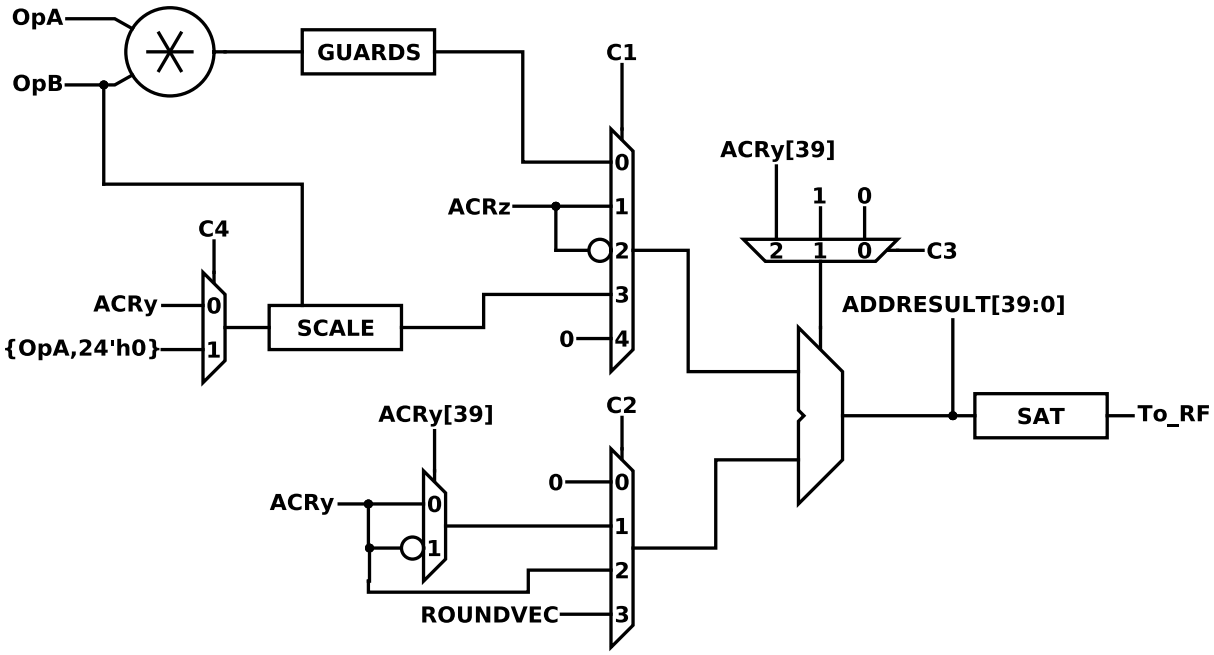
Inputs/outputs:

- OpA, OpB: 16 bit inputs from the general purpose register file/immediates.
- RESULTA, RESULTB: 16 bit outputs that are written to the register file (i.e., the register file has **two** write ports. Please indicate which ports you use in your control table.)
- Flags: Whatever flag outputs the PC FSM requires when running your assembler programs.
- And of course, whatever clock signals and control signals you deem necessary.

(a) (7p) Create an instruction set for your ALU and translate the code above into assembler.

(b) (7p) Draw a schematic and a control table for your ALU.

Solution proposal, question 1



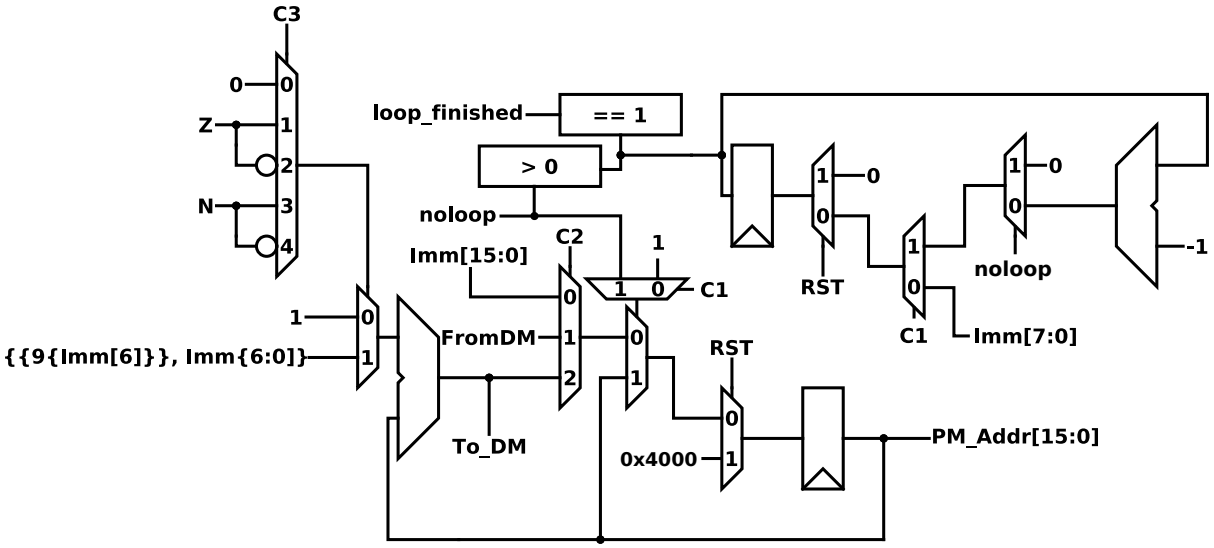

```
always @* begin
    scaleout = scalein >>> OpB; // SCALE module
    roundvec = {39'b0, ACRy[OpB-1]};

end

// Note: In the original exam version, SCALE was defined as follows:
// If OpB = 24: res = foo[15:0]
// If OpB = 23: res = foo[16:1]
// ...
// If OpB = 0: res = foo[39:24]
//
// I later realized during correction that while this could made sense
// for SAT(ROUND(SCALE(ACRy, OpB))) it doesn't really make sense for
// the other operations where SCALE is used.
// Therefore this part of the question was corrected liberally. For example,
// the following solution would be ok for scaleout:

always @* begin
    case(OpB)
        24: scaleout = scalein[15:0]
        23: scaleout = scalein[16:1]
        // And so on...
        0: scaleout = scalein[39:24]
    endcase
end

// Or this solution:
always @* begin
    scaleout = scalein >> OpB;
    scaleout = scaleout[15:0];
end
// (Some other interpretations were also allowed.)
```

```
// Personally I prefer to describe register files using RTL code
// rather than by drawing a schematic:
reg [39:0] ACR[3:0];
always @(posedge clk) begin : ACR_RF
    if(Cw) begin
        ACR[x] <= ADDRESULT;
    end
end

always @* begin : SAT
    To_RF = ADDRESULT[15:0];
    if(ADDRESULT[39]) begin
        if(ADDRESULT[38:15] != 24'hffffff) begin
            To_RF = 16'h8000;
        end
    end else if(ADDRESULT[38:15] != 24'h000000) begin
        To_RF = 16'h7fff;
    end
end
```

Control table:

Operation	C1	C2	C3	C4	Cw
OP0 (NOP)	-	-	-	-	0
OP1 (CLR)	4	0	-	-	1
OP2 (MUL)	0	0	0	-	1
OP3 (MAC)	0	2	0	-	1
OP4 (ADDL)	1	2	0	-	1
OP5 (SUBL)	2	2	1	-	1
OP6 (ABSL)	2	2	2	-	1
OP7 (SCALE)	3	0	0	0	1
OP8 (ACR+SCALE)	3	2	0	1	1
OP9 (SATRND)	0	2	3	0	0

Solution proposal, question 2



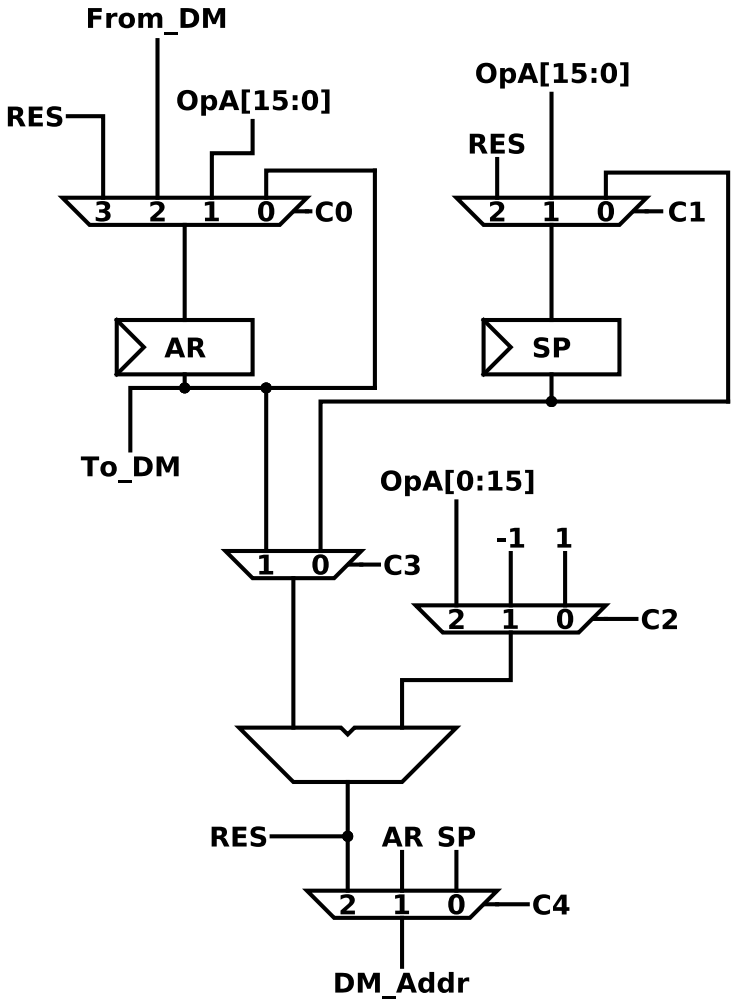
Operation	C1	C2	C3
PC++	1	2	0
JSR	1	0	0
RET	1	1	-
BMI	1	2	3
BPL	1	2	4
BEQ	1	2	1
BNE	1	2	2
JUMP	1	0	-
REPEAT	0	0	-

Note: A single cycle repeat instruction will also require some support from the instruction decoder (since it may have to stall fetches into the instruction register before the PFC has had a chance to notice that a repeat instruction has occurred, depending on the length of the pipeline). In the schematic given above, it is intended that loop_finished is a signal that is sent back to the instruction decoder as a notification that it is soon time to start fetching instructions into the instruction register again.

Solution proposal, question 3

```
// Note: We save a few clock cycles by not resetting the sum value for
// each time we call sum_structs.
sum_structs:                // Assumes ptr1 in AR
        set  r1,#0          // Reset sum.
sum_structs_begin:
        set  r3, #16        // Loop counter
loop:
        add  r3,r3,#-1      //
        beq  loopdone
        load r2,dm0[AR++]  // Delay slot

        load r0,dm0[AR++]
        cmp  r0,#0

        bne  loop           //
        add  r1,r1,r2       // Delay slot

        push AR

        call sum_structs_begin
        move AR,r0

        bra  loop
        pop  AR            // Delay slot

loopdone:
        ret                 // Returns the sum in register r1
        nop                 // Delay slot
// Alternative implementation ideas:
// * Use MAC instructions for the sum. Since the MAC unit typically has
//   a direct connection to the memory we can convert one load/add into
//   a single mac instruction.
// * Use a repeat instruction rather than add r3,r3,#-1; beq loopdone
//   The drawback is that it is probably necessary to save the state of
//   the repeat loop status registers. This might take more time than
//   we save by using repeat in the first place.
// * Use loop unrolling to reduce cost of loop logic.
// * Allow loads to write directly to the address register (similar to
//   how push/pop must work).
```

```
postprocess_fft:
        set r2, #fftoutput
        move AR, r2
        set r0, #0   // Best match so far (maxval)
        set r1, #0   // Index of best match so far (maxidx)
        set r2, #0   // Contains current index

        repeat 256, endloop

        load r3, dm0[AR + bitreverse(r2)]
        nop            // Avoid data hazard for load
        abs  r3, r3

        cmp  r3, r0    // val > maxval
        blt  skip
        nop            // Delay slot.

        move r0,r3     // Update maxval
        move r1,r2     // Update maxidx (actually maxidx * 0x100)

skip:
        add r2, #0x100 // (0x100 is the appropriate increment of a 16
                       // bit number when bitreversing 256 entries)
endloop:

        ret
        lsr r1,r1, #8  // r1 = r1 >> 8 (Delay slot)


// Alternative implementation ideas:
// * The nops can be removed by rewriting the code above. However, they
//   are left for clarity since the code will be harder to read if they
//   are removed.
// * Use conditional execution to avoid blt skip
// * If the performance constraints are higher, put r2 into a second
//   address register and use the addressing mode dm0[AR0 + bitreverse(AR1+=RF)]
```



Control table:

Operation	C0	C1	C2	C3	C4	Comment
NOP	0	0	-	-	-	
AR = OpA	1	0	-	-	-	
SP = OpA	0	1	-	-	-	
ADDR = AR; AR++	3	0	0	1	1	
To_DM = AR; SP−−; ADDR = SP	0	2	1	0	2	Push AR
AR = From_DM; ADDR = SP; SP++	2	2	0	0	0	Pop AR
SP−−; ADDR = SP	0	2	1	0	2	Push X
ADDR = SP; SP++	0	2	0	0	0	Pop X
ADDR = AR + bitreverse(OpA)	3	0	2	1	2	

Note: Pop AR is probably a bit more complicated in a real processor than depicted here. The problem is that ADDR = SP++ probably needs to be executed a couple of clock cycles before AR is set to To_DM since the AGU is at least one pipeline stage earlier than the address output from the data memory. Push X, Pop X is needed for general stack operations (e.g. push/pop return address for subroutines). Pop AR would instead be divided into two operations (i.e. increment AR and AR = From_DM).

Solution proposal, question 4

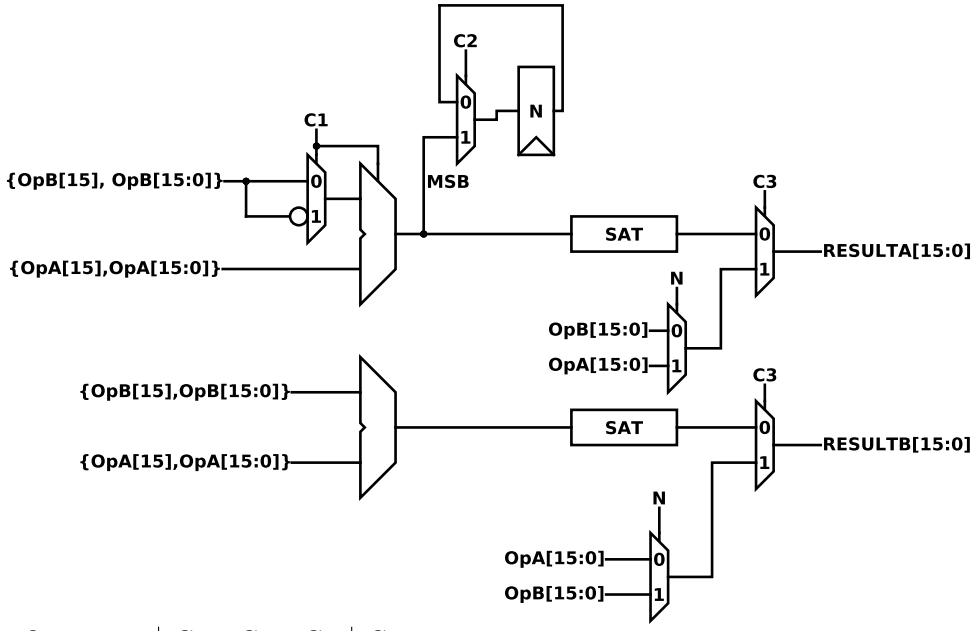
a) The main advantage of using two memory banks as compared to a single, wider, memory bank that arbitrary locations in these memories can be accessed simultaneously. The main disadvantage is that you will get an area overhead when using two memory banks rather than one. Similarly, you will also have a higher power consumption when reading from two separate memories instead of one, wider, memory.

b) One usecase for this feature is that it is possible to run a large number of calculations (for example a transform of some sort) and see if saturation ever occurred during the calculation without having to check the S flag after each computation. (If saturation has occurred the program can for example scale down the input data before the next calculation, or possibly rescale the current data and rerun the transform, although this will complicate the calculation of worst case execution time for real-time systems.)

c) See lab 4 (how the register file is protected via rf_busy). (Alternative explanations could include a description of how to accurately model the pipeline in an instruction set simulator.)

Solution proposal, question 5

```
// Proposed assembler code, dct_transform, statement 9-15:

Statement 1-8: Use addsub instructions, 4 clock cycles

Statement 9-14:

sataddsub r10,r13,r0,r3   // r10 = tmp0+tmp3, r13 = tmp0-tmp3
sataddsub r11,r12,r1,r2   //
sataddsub r20,r24,r10,r11 // r20 (data_0) = tmp10+tmp11, r24 (data_4) = tmp10-tmp11

Statement 15:
mul acr0,r12,CONSTANT0  // (where CONSTANT0 is the register which contains this constant.
mac acr0,r13,CONSTANT1
satrnd r22,acr0


Statement 16, same instructions as stmt 15 (except mdm rather than mac is used), 3 cycles
Stmt 17-20: Use sataddadd instruction (2 cycles) (same as sataddsub but uses two additions
Stmt 21: Use mul and mac  (2 cycles)
Stmt 22-25, 28-31: Use mul, 8 cycles
Stmt 26-27: Use addl (add long, using acr:s) (2 cycles)
Stmt 32-35: Use addl, 8 cycles.

// Total cost: 35 cycles

sortvals:
    repeat 100,endloop
    load r0, DM0[ar0+0]
    load r1, DM0[ar0+1]
    load r2, DM0[ar0+2]
    set r10,#0
    set r11,#1
    set r12,#2

    cmp r1,r0
    swap.lt r0,r1
    swap.lt r10,r11

    cmp r2,r1
    swap.lt r2,r1
    swap.lt r12,r11

    cmp r1,r0
    swap.lt r1,r0
    swap.lt r11,r10

    store DM0[ar0++], r0
    store DM0[ar0++], r1
    store DM0[ar0++], r2
    store DM0[ar1++], r10
    store DM0[ar1++], r11
    store DM0[ar1++], r12
endloop:
```



Operation	C1	C2	C3	Comment
nop	-	0	-	
sataddadd	0	1	0	Write both results
sataddsub	1	1	0	Write both results
cmp	1	1	-	Write no results
swap.lt	-	0	1	Write both resluts

Changes from exam version

- Missing braces added to SCALE operation in MAC task
- Clarified that you could interpret SCALE as a division by 2^{OpB}.
- Renumbered MAC task to avoid duplicate OP2:s
- Added missing + before 4096