Examination
Design of Embedded DSP Processors, TSEA26

Date	2014-01-10
Room	U1
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)	14
Course responsible	Andreas Ehliar
Teacher visiting the exam room *Phone number during the exam time*	Andreas Ehliar
Visiting the exam room	About 9 och 11
Course administrator	Anita Kilander
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: Misc knowledge(6p)

(a) (2p) Explain one reason why it doesn't make sense to add too many pipeline stages to a DSP processor

(b) (1p) Discuss one reason why you wouldn't want a cache in your DSP processor

(c) (3p) Draw a block level schematic of a simple DSP processor. This is supposed to be a block level schematic, using blocks with names such as ALU, MAC, and AGU (and all additional blocks you consider part of a DSP processor. Your task is to show the examiner that you have a good grasp of the components present in a DSP processor and that you know how they are connected.

Question 2: ALU(8p)

Draw a schematic and a control table for an ALU with the following specifications:

Required operations:

- OP1: OpA + OpB + SignExtend(Imm[9:0])
- OP2: OpA - OpB + SignExtend(Imm[9:0])
- OP3: ABS(OpA - OpB)
- OP4: ABS(OpA + OpB)
- OP5: ABS(OpB + SignExtend(Imm[9:0]))

Inputs:

- OpA: 16 bit input from the register file
- OpB: 16 bit input from the register file
- Imm: 10 bit input from the register file
- S: Saturation mode: When 0, no saturation should be performed. When 1, saturation should be performed.

Outputs:

- Result: The 16 bit result of the operation

Question 3: PFC(10p)

Create a program flow controller capable of supporting the following operations:

- OP1: Stall PC (that is, PC=PC)
- OP2: PC = PC + 1
- OP3: PC = Imm[15:0]
- OP4: PC = PC + SignExtend(Imm[7:0])
- OP5: PC = RF
- OP6: Branch if equal
- OP7: Branch if not equal

For OP6 and OP7 you should use PC relative addressing like OP4 does. When starting the system, the PC should be set to 0xfff0.

Inputs:

- Z,N,V,C: Flags from the ALU
- RF[15:0]: A 16-bit operand from the register file
- Imm[15:0]: A 16 bit constant from the instruction word
- RST: Reset signal, set to one when active.
- And of course, a clock signal and the control signals you have created.

Outputs:

- PM_Addr[15:0]: Address to the program memory
- To_RF[15:0]: A value which should be written to the register file

(a) (6p) Draw a schematic and control table for your PFC unit.

(b) (4p) In addition to these operations, your PFC unit should also support one 32 bit wide performance counter, something which is useful when profiling a program.

It should be possible to read and write the value of this counter from a program. It should also be possible to change the mode of this counter to one of the following 5 modes:

- MODE0: Never increment the counter
- MODE1: Increment the counter when any kind of branch is taken
- MODE2: Increment the counter when a conditional branch is executed and the condition is true
- MODE3: Increment the counter when a conditional branch is executed but the condition is false
- MODE4: Increment the counter when the program counter is stalled

Add the operations you consider necessary to support this functionality to your PFC schematic and control table.

Question 4: MAC(13p)

You should a MAC unit which can be used in a DSP processor with the following requirements:

- The system should be able to handle arbitrary length FIR filters at a cost of at most one clock cycle / tap (excluding prologue and epilogue)
- In addition, the system should be able to handle an FIR filter with the following specifications, when running at 100 MHz:
  - The system should support an FIR filter with 16 taps.
  - The sample rate of the FIR filter is 100 kHz.
  - Saturation and rounding should be done (as shown below)
  - The A/D converter the filter uses as an input can be read using the ADINPUT instruction
  - The D/A converter the filter uses as an output can be written to using the ADOUTPUT instruction
  - The maximum latency allowed between the time when the first instruction in the interrupt handler is run and the ADINPUT instruction is 100 ns.
  - The maximum latency allowed between the time when the first instruction in the interrupt handler is run and the filtered output being sent via the ADOUTPUT instruction is 400 ns.
  - The first 100 outputs values from the filter are allowed to be undefined (e.g., you don't need to worry about initializing the sample buffer, etc)
  - DM0, DM1, and ADINPUT/ADOUTPUT are all 16 bits wide.

One possible pseudo code implementation of this interrupt routine is seen below, although there are other ways to solve this as well, which will lead to different trade-offs.

```
interrupt_handler:
   save_registers_used_by_the_next_line()  // Hint: Use the push instruction here
   input_value = ADINPUT()  // Max latency from first instruction: 100 ns
   save_all_other_registers_modified_by_interrupt_handler()

   coeffptr = coeff_start
   bufferptr = saved_bufferptr // saved_bufferptr is stored in DM0
   DM1[bufferptr] = input_value
   tmp = 0;
   repeat 15
      tmp = tmp + DM0[coeffptr++] * DM1[bufferptr++]
      if bufferptr == bufferend then
         bufferptr = start
      endif
   endrepeat

   tmp = tmp + DM0[coeffptr] * DM1[bufferptr]
   saved_bufferptr = bufferptr

   tmp = tmp + 0x2000
   if tmp > 0x1fffffff then
      tmp = 0x7fff
   else if tmp < -0x20000000
      tmp = 0x8000
   else
      tmp = tmp >> 14
   endif

   ADOUTPUT(tmp)                 // <-- Max latency from first instruction: 400 ns
   restore_all_saved_registers() // <-- Not performance critical, but should be
   return_from_interrupt()       // reasonably fast.
```

(a) (6p) Select an instruction set suitable for your MAC unit and write commented assembler code for your interrupt handler.

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

Question 5: AGU(13p)

Create an AGU capable of supporting the following functions under the given constraints:

```
// This program must be runnable in less than 30 clock cycles.
function fir(coeffptr, sampleptr, startptr, endptr)
   tmp = 0
   repeat 20
       tmp = tmp + DM0[coeffptr++] * DM1[sampleptr]
       if sampleptr == endptr then
          sampleptr = startptr
       else
          sampleptr++
       end if
   endrepeat
   return tmp
endfunction

// This function must be able to run in at most 12 clock cycles,
// excluding the return instruction.
// The parameters to this function are placed in r0-r3
// You should assume that baseaddr and pitch are always evenly divisible by 64.
function sum_4x4_block(baseaddr0, baseaddr1, pitch, x, y)
   tmp = 0
   repeat 4
      tmp = tmp + sumpixels_2x(baseaddr0, baseaddr1, pitch, x, y)
      tmp = tmp + sumpixels_2x(baseaddr0, baseaddr1, pitch, x+2, y)
      y = y + 1
   endrepeat
   return tmp
endfunction

// This function will only be called by the sum_4x4_block function, so you
// are strongly encouraged to inline this function...
function sumpixels_2x(baseaddr0, baseaddr1, pitch, x, y)
   addr0 = baseaddr0 + x/2 + y*pitch
   addr1 = baseaddr1 + x/2 + y*pitch
   xremainder = x & 1    // Get the LSB bits of x and y
   yremainder = y & 1    //

   if (xremainder == 0 and yremainder == 0)  return DM0[addr0]   + DM1[addr1];
   if (xremainder == 0 and yremainder == 1)  return DM0[addr0]   + DM1[addr1];
   if (xremainder == 1 and yremainder == 0)  return DM0[addr0+1] + DM1[addr1];
   if (xremainder == 1 and yremainder == 1)  return DM0[addr0]   + DM1[addr1+1];
endfunction
```

(a) (6p) Select a number of suitable addressing modes and other necessary AGU operations which would be suitable for your AGU when running the pseudo code listed above under the given constraints and translate `sum_4x4_block` and `sumpixels_2x` into *commented* assembler code (you do not have to translate `fir` into assembler).

Note that it must be clear what each addressing mode and other AGU operation actually do (if you solve the next part of this question correctly, you can assume that the examiner will be able to figure out the functionality of your addressing modes and other AGU operations).

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

Solution proposal, question 1

a)

One reason is that as the number of pipeline stages are increased, the cost (in terms of clock cycles) of a mispredicted jump increases. At a certain point, the penalty of adding another clock cycle latency to a mispredicted branch will reduce the overall performance of the processor, even though the clock frequency itself could be increased (marginally) by adding pipeline stages.

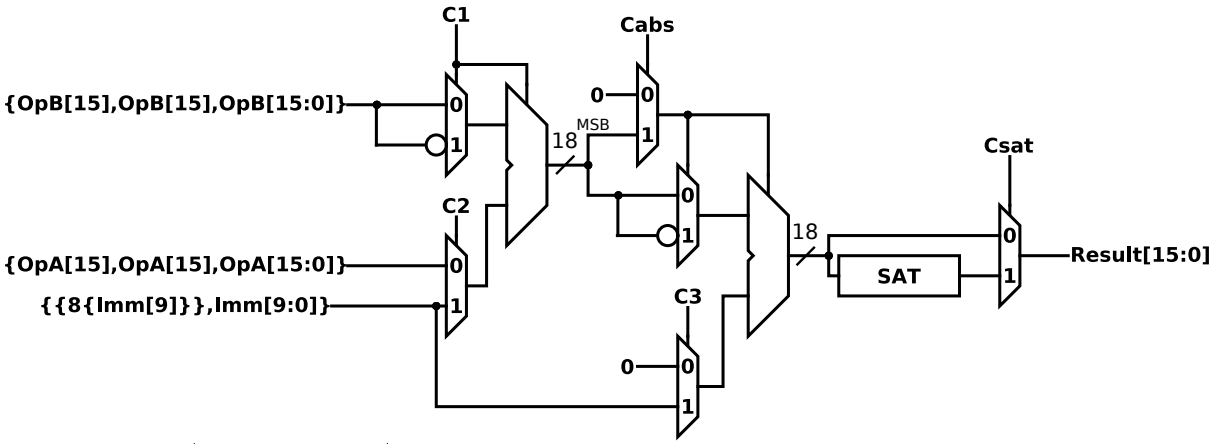
b)

A typical cache requires at least two memory reads (from the tag memory and the memory containing the stored data). If low latency is required, all of the ways in the cache needs to be read out simultaneously as well. If low power consumption is desired it is often beneficial to use a scratch pad memory instead, as only one memory access is required, rather than two.

c)

See for example the pipeline diagram of the Senior processor present in the lab compendium.

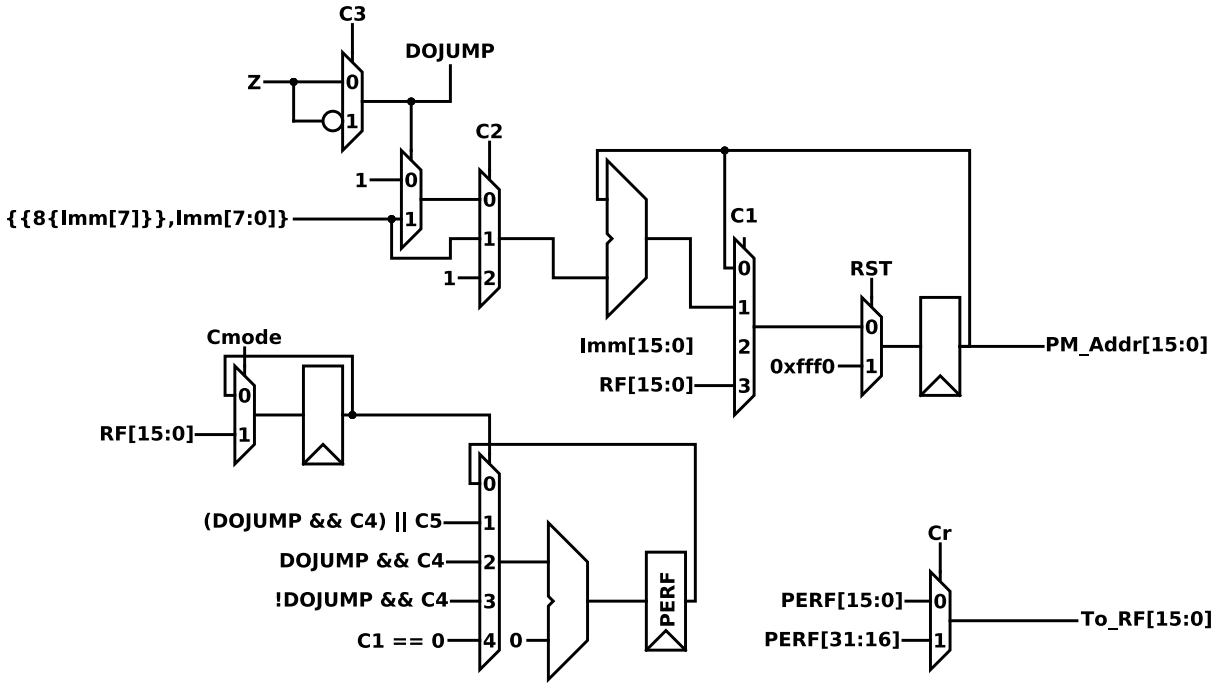
Solution proposal, question 2



Operation	C1	C2	C3	Cabs	Csat
OP1	0	0	1	0	S
OP2	1	0	1	0	S
OP3	1	0	0	1	S
OP4	0	0	0	1	S
OP5	0	1	0	1	S

```
always @* begin : SAT
    if(in[17]) begin
        if(in[16:15] != 3) begin
            out = 16'h8000;
        end else begin
            out = in[15:0]
        end
    end else begin
        if(in[16:15] != 0) begin
            out = 16'h7fff;
        end else begin
            out = in[15:0]
        end
    end
end
```

Solution proposal, question 3



Note: All signals are 16 bit wide except signals that will end up in multiplexer inputs.

Operation	C1	C2	C3	C4	C5	Cmode	Cr
OP1	0	-	-	0	0	0	-
OP2	1	2	-	0	0	0	-
OP3	2	-	-	0	1	0	-
OP4	1	1	-	0	1	0	-
OP5	3	-	-	0	1	0	-
OP6	1	0	0	1	0	0	-
OP7	1	0	1	1	0	0	-
PERFMODE(RF)	1	2	-	0	0	1	-
RF=HIGHPERF	1	2	-	0	0	0	1
RF=LOWPERF	1	2	-	0	0	0	1

Note: this assumes that the performance counter related control signals can be generated in the same pipeline stage as the PFC related control signals. If this doesn't hold two control tables should most likely be used, one for the PFC signals (C1-C3) and one for the performance counter signals (Cmode and Cr). Otherwise it will not be possible to use branches close to the instruction which accesses the performance counter.

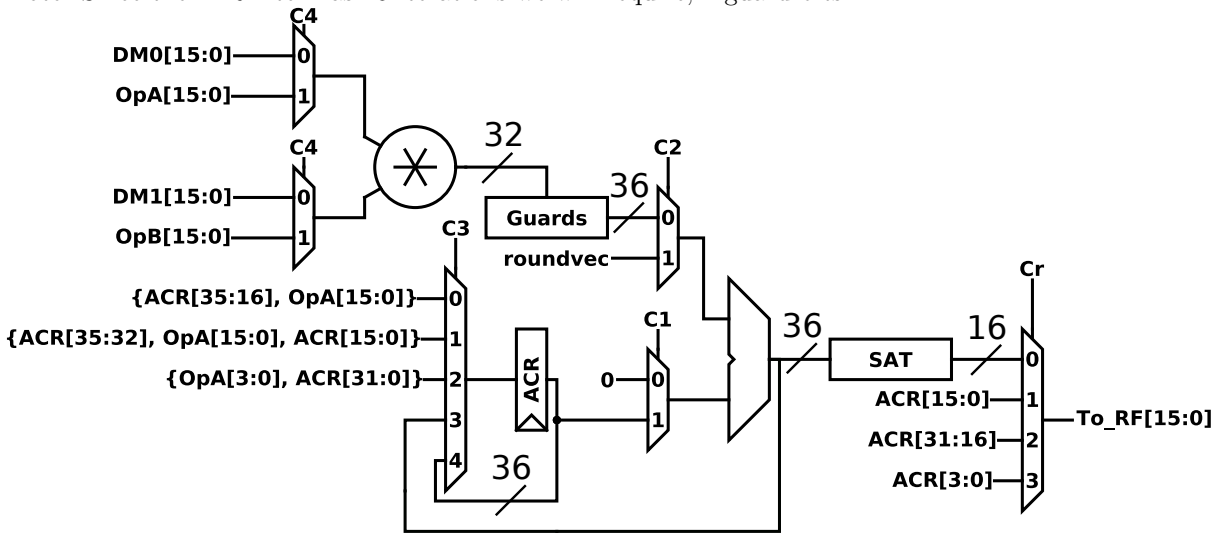
Solution proposal, question 4

```
;;; Assembler code for interrupt handler
interrupt_handler:
        push r1
        ADINPUT r1    ; 20 ns latency from the time when the first insn is executed.

        ; Accumulator
        push r0                  ; If we want lower latency we could include an output
        move r0,LOW(acr)         ; in the MAC unit to allow us to push the acr register
        push r0                  ; directly to the stack.
        move r0,HIGH(acr)
        push r0
        move r0,GUARDS(acr)
        push r0

        ; Address registers
        push ar0
        set  ar0, #coeff_start
        push ar1
        load r0, DM0[saved_bufferptr]
        move ar1,r0
        st   DM1[ar1], r1      ; Store input value in sample buffer

        ; Ring buffer related regs
        push bottom
        set  bottom, #samples_bottom
        push top
        set top, #samples_top

        push loopstatus    ; Assuming interrupts are allowed during a repeat loop
                           ; (which they probably must be due to the strict real-time
                           ; constraints in this system)

        mul acr, DM0[ar0++], DM1[ar1%++]
        repeat endloop, 14
        mac acr, DM0[ar0++], DM1[ar1%++]
endloop:

        mac acr, DM0[ar0], DM1[ar1]  ; Make sure not to increment ar1 here.
        satrnd r0, acr


        ADOUTPUT r0             ; Latency: ~39 cycles = 390 ns
                                ; (We could reduce the latency slightly by unrolling
                                ;  the loop completely and not saving loopstatus.)

        pop loopstatus
        pop top
        pop bottom
        pop ar1
        pop ar0
        pop r0
        move GUARDS(acr),r0
        pop r0
        move HIGH(acr), r0
        pop r0
```

```
        move LOW(acr), r0
        pop r0
        pop r1
        reti


;;; This is another variant where we reduce the latency by precalculating 15 out of 16 taps in the
;;; previous iteration to reduce the latency.
interrupt_handler:
        push r1
        ADINPUT r1    ; 20 ns latency from the time when the first insn is executed.


        ; Accumulator
        push r0
        move r0,LOW(acr)
        push r0
        move r0,HIGH(acr)
        push r0
        move r0,GUARDS(acr)
        push r0

        move r0, DM0[savedacr]
        move LOW(ACR),r0
        move r0, DM0[savedacr+1]
        move HIGH(ACR),r0
        move r0, DM0[savedacr+2]
        move GUARDS(ACR),r0

        set  r0,#lastcoeff
        mac  acr, r0,r1
        satrnd r0, acr
        ADOUTPUT r0             ; 190 ns latency from the first insn

        ; Address registers
        push ar0
        push ar1
        set  ar0, #coeff_start
        load r0, DM0[saved_bufferptr]
        move ar1, r0

        push bottom
        push top
        set  bottom, #samples_bottom
        set  top, #samples_top



        push loopstatus    ; Assuming interrupts are allowed during a repeat loop
                           ; (which they probably must be due to the strict real-time
                           ; constraints in this system


        mul   acr, DM0[ar0++], DM1[ar1%++]
        repeat endloop, 14
        mac   acr, DM0[ar0++], DM1[ar1%++]
endloop:
```

```
store DM1[ar1], r1      ; Store input value in sample buffer
mov   r0,ar1            ; Save bufferptr
store DM1[saved_bufferptr], r0

;;  Save acr for last sample (calculated next interrupt)
move r0,LOW(acr)
store DM0[savedacr]
move r0,HIGH(acr)
store DM0[savedacr+1]
move r0,GUARDS(acr)
store DM0[savedacr+2]

pop loopstatus
pop top
pop bottom
pop ar1
pop ar0

;;  Restore accumulator
pop r0
move GUARDS(acr),r0
pop r0
move HIGH(acr), r0
pop r0
move LOW(acr), r0

pop r0
pop r1
reti
```

Note: Since the FIR filter has 16 iterations we will require, 4 guard bits.



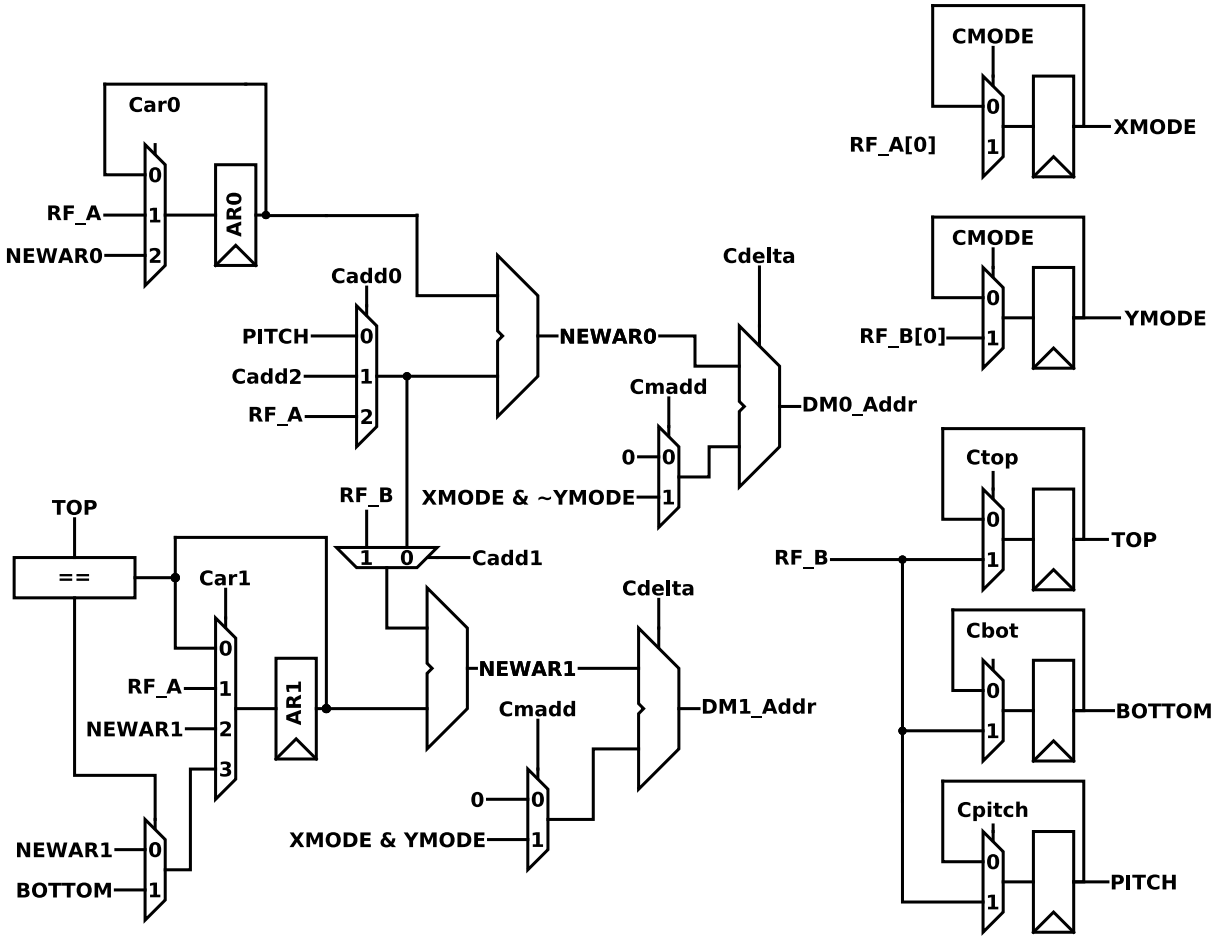
Operation	C1	C2	C3	C4	Cr
NOP	-	-	4	-	-
RF=LOW(ACR)	-	-	4	-	1
RF=HIGH(ACR)	-	-	4	-	2
RF=GUARD(ACR)	-	-	4	-	3
LOW(ACR)=RF	-	-	0	-	-
HIGH(ACR)[=RF	-	-	1	-	-
GUARD(ACR)=RF	-	-	2	-	-
MUL (DM)	0	0	3	0	-
MAC (DM)	1	0	3	0	-
MAC (RF)	1	0	3	1	-
satrnd	1	1	3	-	0

```
roundvec = 36'h2000;
```

```
always @* begin : SAT
    out = in[29:14]
    if(in[35:29] != 7'b11111111) begin
        if(in[35:29] != 7'b0) begin
            out = {in[35], {15{in[35]}}};
        end
    end
end
```

Solution proposal, question 5

```
fir:   ; Not required by the question, but included here for completeness
    mov ar0,r0
    mov ar1,r1
    mov top, r2
    mov bottom,r3

    clr acr
    repeat endloop, 20
    mac acr,DM0[ar0++], DM1[ar1%++]
endloop:
    ret  ; Assume return value in acr


;;; r0 = baseaddr0
;;; r1 = baseaddr1
;;; r2 = pitch
;;; r3 = x
;;; r4 = y

sum_4x4_block: ; sumpixels_2x is inlined here a couple of times
        mul r6, r4,r2   ; r6 = pitch * y

        setxy  r3,r4    ; ar0 = r3, ar1 = r3; XMODE = r3[0]; YMODE = r4[0]
        updatepitch r6,r2 ; ar0 += r6, ar1 += r6, PITCH = r2
        updateaddr r0,r1  ; ar0 += r0, ar1 += r1

        add  r0,     DM0, DM1, XYDELTA(0,0)  ; Set r0 to DM0 + DM1

        add3 r0, r0, DM0, DM1, XYDELTA(+,++) ; Add DM0+DM1 to r0, use offset 1 for
                                             ;both memories, increment y with pitch

        add3 r0, r0, DM0, DM1, XYDELTA(0,0)  ; and so on
        add3 r0, r0, DM0, DM1, XYDELTA(+,++)

        add3 r0, r0, DM0, DM1, XYDELTA(0,0)
        add3 r0, r0, DM0, DM1, XYDELTA(+,++)

        add3 r0, r0, DM0, DM1, XYDELTA(0,0)
        add3 r0, r0, DM0, DM1, XYDELTA(+,++)

        ret                     ; 12 cycles (excluding ret)



;;; Description of xydelta mode:

        XYDELTA(0,0):     ADDR_TO_DM0 = ar0 + (XMODE & ~YMODE)
                          ADDR_TO_DM1 = ar1 + (XMODE &  YMODE)
        XYDELTA(+,++):    ADDR_TO_DM0 = ar0 + 1 + (XMODE & ~YMODE) ; ar0+= PITCH
                          ADDR_TO_DM1 = ar1 + 1 + (XMODE &  YMODE) ; ar1+= PITCH
```



Notes: All signals are 16 bits wide (except for XMODE and YMODE that are 1 bit wide and control signals that have the obvious width).

Operation	Car0	Car1	Cadd0	Cadd1	Cadd2	Cmadd	Cdelta	CMODE	Ctop	Cbot	Cpitch
NOP	0	0	-	-	-	-	-	0	0	0	0
ar0++/ar1%++	2	3	1	0	1	0	0	0	0	0	0
load TOP	0	0	-	-	-	-	-	0	1	0	0
load BOT	0	0	-	-	-	-	-	0	0	1	0
setxy	1	1	-	-	-	-	-	1	0	0	0
updatepitch	2	2	0	0	-	-	-	0	0	0	0
updateaddr	2	2	2	1	1	-	-	0	0	0	0
XYDELTA(0,0)	0	0	0	0	0	-	-	0	0	0	0
XYDELTA(+,++)	2	2	0	0	-	1	1	0	0	0	0

Why would you want this addressing mode?

At first glance, this seems like a very odd addressing mode. However, this kind of addressing can be useful in for example image processing applications when multiple pixels needs to be loaded at the same time. Consider for example an image where two adjacent pixels (either horizontal or vertical) needs to be loaded in one clock cycle. If the image is organized as follows:

```
0101010101010101010101010101010101.....   pitch = the width of the image
1010101010101010101010101010101010.....   0: Pixel is found in DM0
0101010101010101010101010101.....         1: Pixel is found in DM1
101010101010....
.....
```

That is, with this organization of the frame buffer into two memory banks, two adjacent pixels can be loaded at the same time since they are always located in different memory banks. The drawback is that every odd line needs a somewhat awkward memory addressing scheme.

(It could also be noted that this exercise was made slightly easier by the fact that an addition operation was used since addition is commutative. If for example a 2D convolution kernel is used instead, it will be necessary to add multiplexers at the memory outputs so that it is possible to swap the output from the memories depending on the x and y address (alternatively, the coefficients can be swapped).

Multiplier based solution

The proposed schematic does not contain any multipliers. However, a more obvious solution would set ar0 to the x coordinate and ar1 to the y coordinate and include a multiplier. This could also net you a full score as long as you ensure that the multiplier is not wider than necessary (i.e. you take advantage of the fact that *pitch* is always a multiple of 64).

```
;;; Assembler code for solution where y is multiplied with pitch for
;;; every clock cycle:
sum_4x4_block:
        setbaseaddr r0,r1    ; ar0 = r0, ar1 = r1
        setpitch    r2
        setxy       r3,r4
        add         r0, DM0[XY(ar0,x,y)], DM1[XY(ar1,x,y)]
        add3        r0,r0, DM0[XY(ar0,x+2,y)], DM1[XY(ar1,x+2,y++)]
        repeat      endloop,3
        add3        r0,r0, DM0[XY(ar0,x,y)], DM1[XY(ar1,x+2,y)]
        add3        r0,r0, DM0[XY(ar0,x+2,y)], DM1[XY(ar1,x+2,y++)]
endloop:
        ret      ; (Could be reduced to 11 cycles by unrolling the loop
```

(The schematic for this variant is not included here but should be relatively straight forward to implement. The important issue here is that the AGU needs to check the LSB bits of the X and Y part of the address and adjust the address accordingly.)

Revision history (version 1.2):

Changes from exam version

- Clarified that DM0, DM1, and ADINPUT/ADOUTPUT are 16 bits wide
- It is more realistic if sum_4x4_block take both baseaddr0 and baseaddr1 rather than just baseaddr.

Changes from v1.0

- Added solution proposal for question 3
- Clarified that MSB bit is sent to mux in question 2.
- Added bit width annotations for question 2
- Added assembler code for multiplier based solution to question 5

Changes from v1.1

- Added adder to performance counter in PFC task
- Added missing signals to control table in PFC task