Examination
Design of Embedded DSP Processors, TSEA26

Date	2014-10-31
Room	R34, R35, R36
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 9.30 and 11
Course administrator	Andreas Ehliar
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: Miscellaneous questions(6p)

(a) (2p) Create a saturation unit which is capable of saturating a 24 bit two's complement number to a 16 bit two's complement number. This saturation unit should also have an output which shows whether saturation was performed or not.

(b) (3p) Discuss how you can decide how many bits you will require in your accumulator for a certain FIR filter. (If you want to, you can assume that all inputs and outputs are in fractional format.)

(c) (1p) Explain briefly what hardware multiplexing is

Question 2: Address Generation Unit(8p)

Create an AGU with support for the following operations:

- NOP
- ADDR = AR
- ADDR = RF
- ADDR = AR + Imm
- ADDR = AR; AR = AR + 1
- ADDR = AR; AR = AR - 1
- ADDR = AR; AR = AR + STEP
- ADDR = AR; AR = AR - STEP
- STEP = RF
- AR = RF

Inputs/outputs

- RF: A 16 bit input from the register file
- Imm: A 16 bit input from the instruction word
- ADDR: A 16 bit address output to the memory
- And of course whatever clock and control signals you deem necessary

Your tasks:

(a) (4p) Draw a schematic, and a control table of this AGU

(b) (2p) Modify the schematic and control table so that you also have all operations that are required to support efficient addressing of circular buffers (i.e. modulo addressing). Only a step size of one is required.

(c) (2p) Suppose you want to implement a *single cycle* PUSH and POP instruction for access to a software based stack. Are the AGU operations you have implemented so far sufficient? If so, prove it by explaining how you would implement a PUSH and POP instruction by using these addressing modes. If not, add whatever operations you deem necessary to the AGU and explain how you, with the help of your modified AGU, can implement PUSH and POP.

Question 3: Arithmetic Logic Unit(10p)

Create an ALU with support for the following operations:

- OP1: RESULT = OpA + OpB
- OP2: RESULT = OpA - OpB
- OP3: RESULT = SAT(OpA + OpB)
- OP4: RESULT = SAT(OpA - OpB)
- OP5: RESULT = XOR(OpA, OpB)
- OP6: RESULT = OR(OpA, OpB)
- OP7: RESULT = AND(OpA, OpB)

Inputs/Outputs:

- OpA, OpB: 32 bit inputs
- RESULT: 32 bit output
- MODE: 2 bit input interpreted as follows:
  - **00**: OpA, OpB, and RESULT should be interpreted as 32 bit wide values.
  - **01**: OpA, OpB, and RESULT should be interpreted as two concatenated 16 bit wide values
  - **10**: OpA, OpB, and RESULT should be interpreted as four concatenated 8 bit wide values
  - **11**: Undefined (do whatever you wish)
- Whatever clock and control signals you deem necessary

Example:

In order to solve this exercise it is important to understand how the MODE signal works. The following is an example of how OpA, OpB, and RESULT are handled depending on what MODE is set to.

Mode	OpA	OpB	RESULT
00	`0x12345678`	`0xffffffff`	`0x12345677`
01	`0x1234 5678`	`0xffff ffff`	`0x1233 5677`
10	`0x12 34 56 78`	`0xff ff ff ff`	`0x11 33 55 77`

Your task: Draw a schematic and a control table for the AGU described above. When drawing this schematic you do not need to explain how your saturation blocks work, as this has been explicitly covered by another question on this exam. However, you do need to annotate the bit width of the inputs and outputs to your saturation block.

Hint 1: You should reduce the number of adders required in this solution. An N bit wide adder can be created by cascading two $N/2$ bit wide adders.

Hint 2: In order to make your schematic less messy, you can divide it into more than one part and use named signals to connect the parts. (E.g., divide it into a part with adders and a part with post-processing.) Another solution you could consider is to make a hierarchical schematic where a certain module is instantiated several times.

Question 4: Multiply and Accumulate unit(15p)

Create a MAC unit which supports the following two functions:

```
// The following is a four tap FIR filter which runs for 100
// iterations. (ptr0 points to the input buffer, ptr1 to the
// coefficients and ptr2 to the output buffer). You can assume that
// ptr0 to ptr2 are available in either address registers or normal
// registers (your choice).
function filter(ptr0, ptr1, ptr2)

    repeat 100
        ptr3 = ptr0++;
        ptr4 = ptr1;
        tmp = 0;
        repeat 4
            tmp = tmp + DM0[ptr3++] * DM1[ptr4++];
        endrepeat
        tmp = (tmp + 0x4000);
        tmp = tmp >> 15;  // Arithmetic shift
        DM0[ptr2++] = saturate(tmp);
    endrepeat
endfunction

// This is a signed 32x32 bit multiplication with a 32 bit result.
function mult_32x32(ptr0, ptr1, ptr2)
   // val1, val2, and result are 32 bit signed integers
   val1 = DM0[ptr0];                    // Hint: Store val1 in two 16 bit registers
   val1 = val1 + DM0[ptr0+1] << 16;     // (Same for val2 and results)

   val2 = DM0[ptr1];
   val2 = val2 + DM0[ptr1+1] << 16;

   result = SATURATE(val1*val2); // ***

   DM0[ptr2] = result & 0xffff;
   DM0[ptr2+1] = (result >> 16) & 0xffff;
endfunction
```

Constraints:

- The register file has 32 registers that are 16 bit wide. Two read ports and one write port are available.
- DM0 and DM1 are 16 bit wide
- **The `filter` function must execute in at most 1600 clock cycles.**
- **It is important (for power reasons) that the `filter` function does not access the DM0 and DM1 unless necessary. For maximum points you need to get down to less than 130 read accesses in total.** There are no limitations on the number of write accesses.
- **The line marked with `***` in `mult_32x32` must execute in at most 12 clock cycles.**
- You can decide how many accumulation registers you need, and the width of these accumulation registers.
- When drawing your schematic, **you do not need to explain the content of your saturation block** (since this is explicitly covered by a separate question in this exam). However, you still need to annotate the number of bits coming in and out from this block.
- You are allowed to choose your own inputs and outputs in this problem (within reason)

Your tasks:

(a) (8p) Create an instruction set for your MAC unit and translate the `filter` and `mult_32x32` function into assembler. In the `mult_32x32` function I am mostly interested in how you handle the line marked with `***`, so this is the only part of that function that you *have* to translate into assembler. (Although you need to comment your source code so that it is clearly understandable how the inputs and outputs to your part of the code are handled. (E.g., in what register is the high part of val1 stored, etc).)

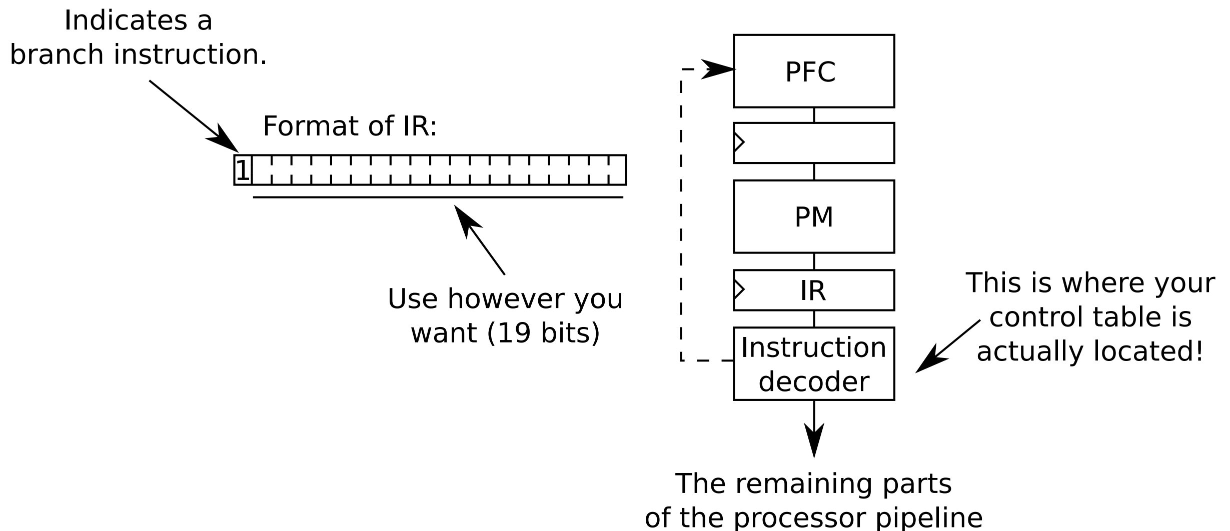
(b) (7p) Draw a schematic of your MAC unit and a control table.

Question 5: Program Flow Controller(11p)

Implement a program flow controller suitable for the the programs listed below:

```
function fir_kernel(ptr1, ptr2, ptr3)
    tmp = 0
    repeat 32
        tmp = tmp + DM0[ptr1++]*DM1[ptr2++];
    endrepeat
    DM1[ptr3] = saturate(round(tmp))
    return
endfunction

function handle_filter(ptr0)
    ptr1 = DM0[ptr0]
    ptr2 = DM0[ptr0+1]
    ptr3 = DM0[ptr0+2]

    if ptr3 == 0 then
        return
    endif
    if ptr2 == 0 then
        ptr2 = DM0[ptr0+3]
    endif

    fir_kernel(ptr1,ptr2,ptr3)
    return
endfunction
```

Your program flow controller should be implemented in the PFC block in the figure below. **Hint:** Study the figure to ensure that you understand how the pipeline of the parts you are not allowed to modify will impact your PFC operations (e.g., the number of delay slots).



Constraints:

- `fir_kernel` must run in at most 50 clock cycles
- `handle_filter` must run in at most 30 clock cycles (excluding the time spent in `fir_kernel`)
- In this task it is **important** that you handle control hazards correctly in your assembler code. (E.g., write your assembler code so that the correct number of delay slots are used in your PFC instructions.)
- You must support two layers of subroutine calls. You may not use function inlining to get around this when writing your assembler code.
- The address to the program memory is 16 bits wide. You will not know where in memory each subroutine is located.
- You may use whatever inputs and outputs you require.

(a) (5p) Select the PFC operations you will require and translate the programs listed above to assembler. For each of your selected PFC operation you must specify an instruction encoding as well, based on the figure on the previous page.

(b) (4p) Draw a schematic and a control table of your PFC unit

(c) (2p) You will get these points if you correctly handle control hazards in this task (e.g., delay slots)

Solution proposal, question 1

a)

```
always @* begin
    did_sat = 0;
    if(in[23:15] == 9'b111111111) begin
        out = in[15:0];
    end else if(in[23:15] == 9'b000000000)
        out = in[15:0];
    end else if(in[23]) begin
        did_sat = 1;
        out = 16'h8000;
    end else begin
        out = 16'h7fff;
        did_sat = 1;
    end
end
```

b)

The number format doesn't matter as long as fixed point arithmetic is used since the multiplier and adder does not know where the radix point is. To simplify calculations I will thus interpret the inputs to the multiplier as an integer value.

To calculate the largest and smallest possible value the following equations can be used:

$$maxsum = \sum_{i=1}^{N} max(c_i \cdot largestpos, c_i \cdot largestneg) \tag{1}$$

$$minsum = \sum_{i=1}^{N} min(c_i \cdot largestpos, c_i \cdot largestneg) \tag{2}$$

(where $largestneg$ is the largest negative sample value allowed and $largestpos$ is the largest positive value allowed)

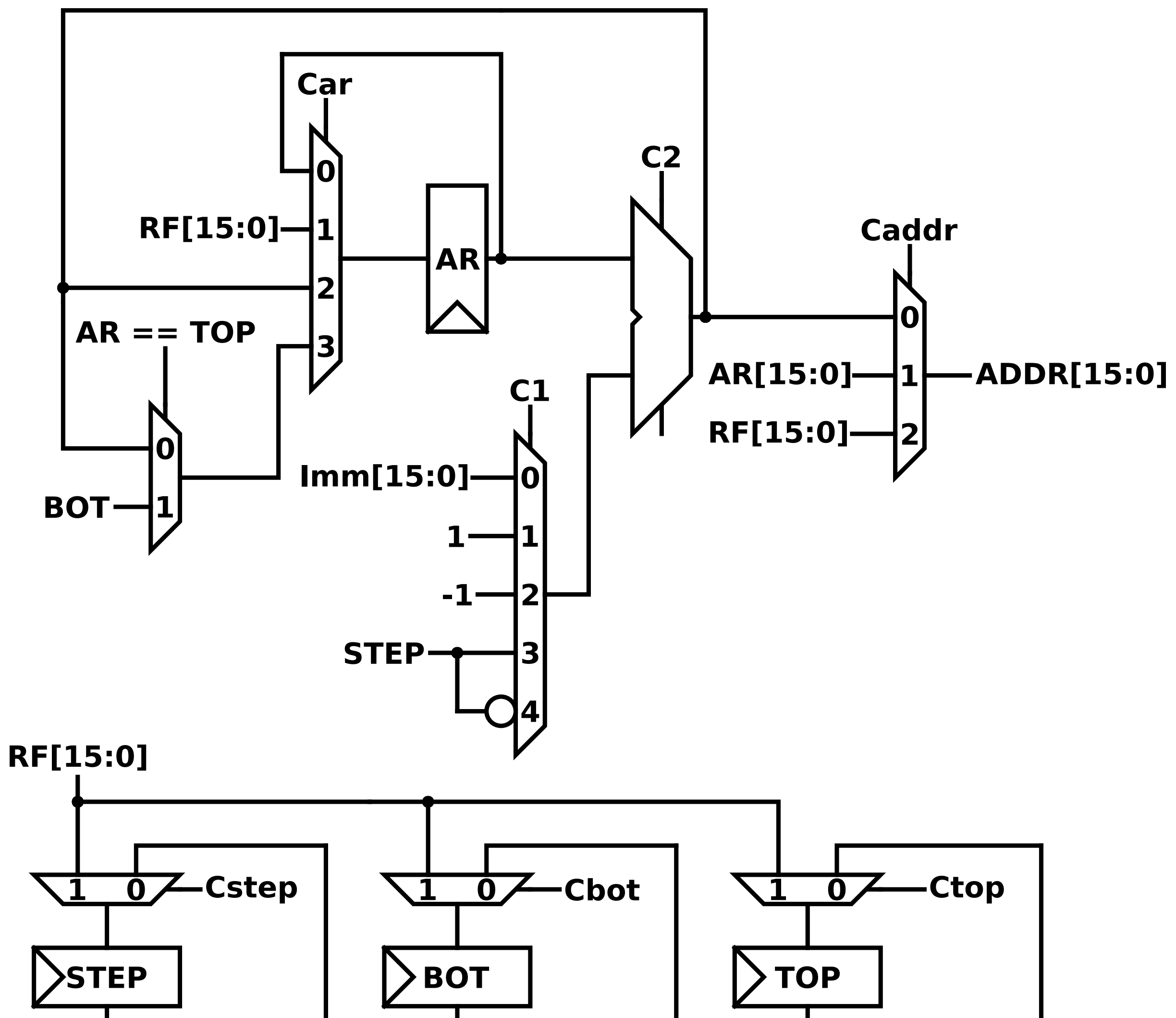
To ensure that an overflow will never occur the following condition must hold, where W is the bit width of the accumulator:

$2^{W-1} - 1 \geq maxsum$ and $minsum \geq -2^{W-1}$

c)

See the textbook

Solution proposal, question 2



Control table:

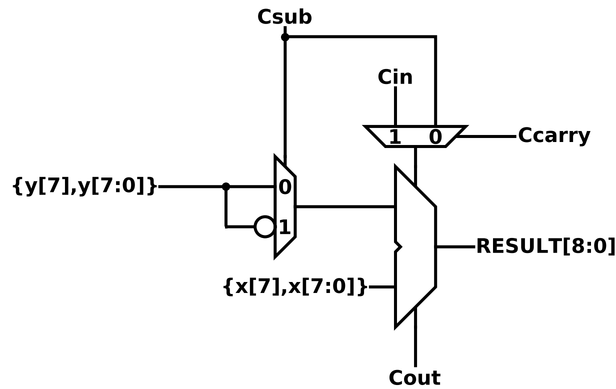
Operation	Car	Caddr	C1	C2	Cstep	Cbot	Ctop	Comments
NOP	0	-	-	-	0	0	0	
ADDR=AR	0	1	-	-	0	0	0	
ADDR=RF	0	2	-	-	0	0	0	
ADDR=AR+Imm	0	0	0	0	0	0	0	
ADDR=AR++	2	1	1	0	0	0	0	
ADDR=AR−−	2	1	2	0	0	0	0	
ADDR=AR+=STEP	2	1	3	0	0	0	0	
ADDR=AR-=STEP	2	1	4	1	0	0	0	
STEP=RF	0	-	-	-	1	0	0	
AR=RF	1	-	-	-	0	0	0	
ADDR=AR%++	3	0	1	0	0	0	0	Added for modulo
BOT=RF	0	-	-	-	0	1	0	addressing mode
TOP=RF	0	-	-	-	0	0	1	
ADDR=−−AR	2	0	2	0	0	0	0	Added for stack

For the stack to work correctly it is necessary to add either preincrement or predecrement mode. In the control table I added predecrement mode as this is typically used when pushing an item on the stack. When using pop this corresponds to postincrement mode:

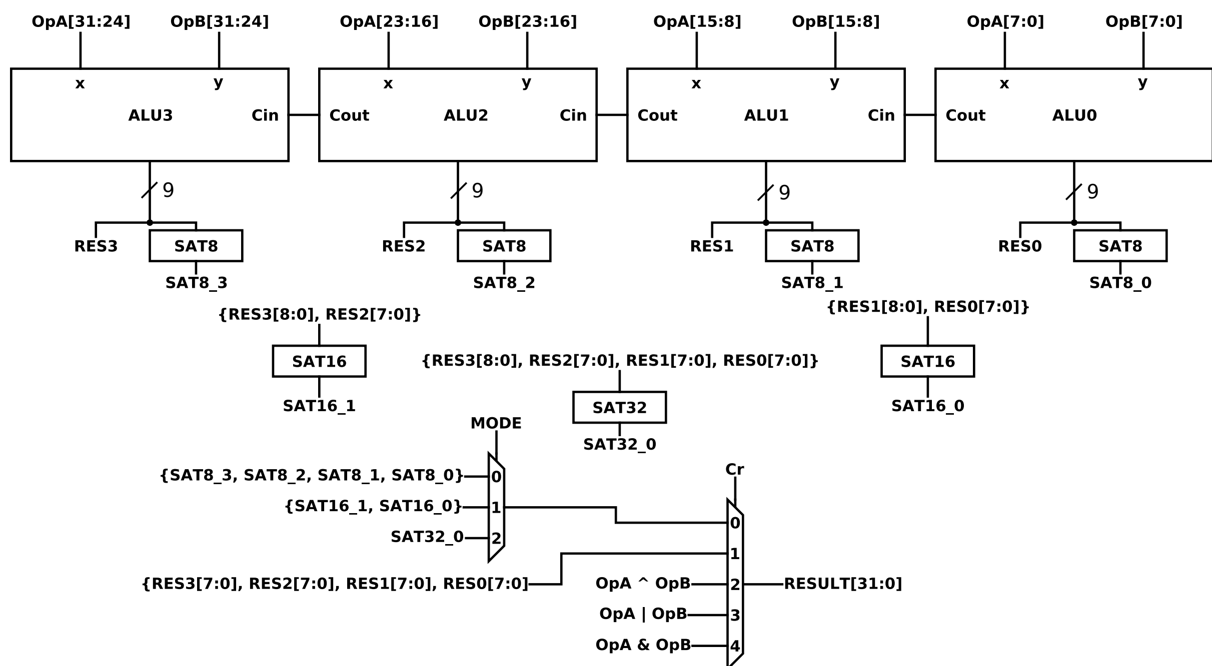
```
push r0     <->     move DM[--ar0], r0
pop  r0     <->     move r0, DM[ar0++]
```

Solution proposal, question 3

To simplify the schematic, a simple 8 bit wide ALU is first designed:



This ALU is instantiated 4 times:



The SAT8 unit saturates a 9 bit number to 8 bits. The SAT16 unit is similar and saturates a 17 bit number to 16 bits. Finally, the SAT32 unit saturates a 33 bit number to 32 bits.

Control table for Csub and Cr:

	ALU*.Csub	Cr
op1	0	1
op2	1	1
op3	0	0
op4	1	0
op5	-	2
op6	-	3
op7	-	4

Control table for ALU*.Cc:

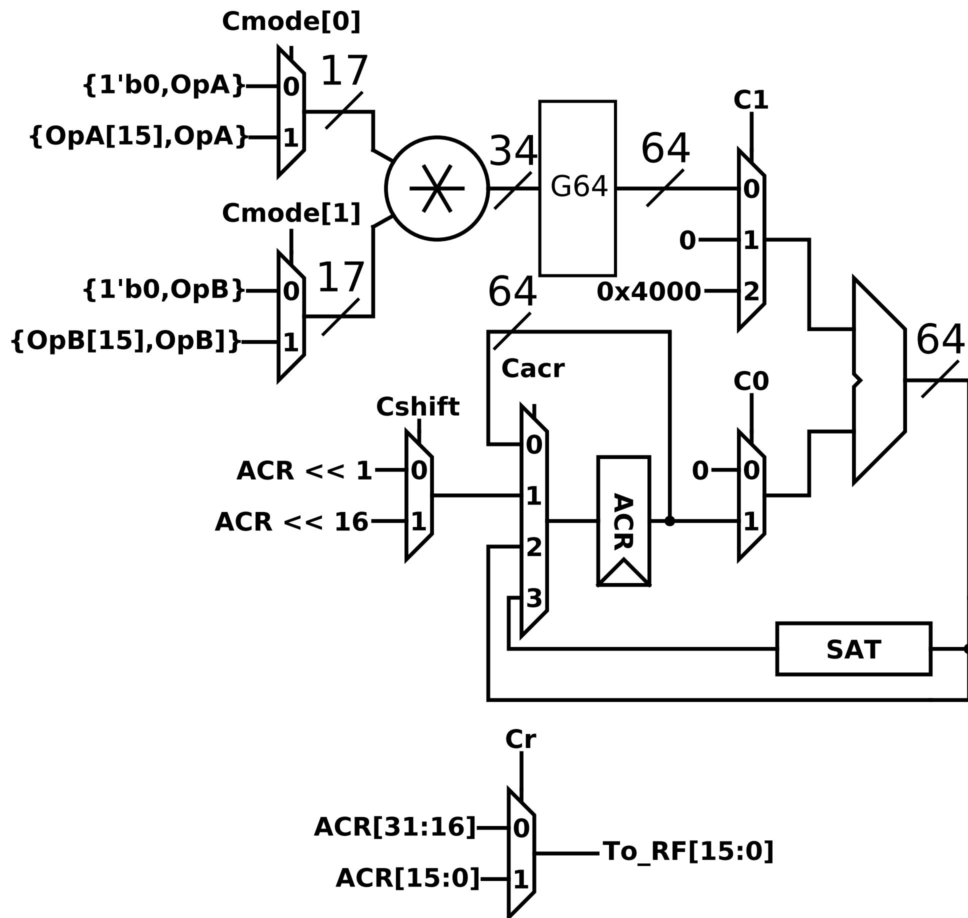
MODE	ALU3.Cc	ALU2.Cc	ALU1.Cc	ALU0.Cc
00	0	0	0	1
01	0	1	0	1
10	1	1	1	1

Solution proposal, question 4

Assembler code for question 4:

```
filter:
        move ar2, r2            ; Output ptr
        load r16, DM1[r1]    ; Load taps into r16-r19
        load r17, DM1[r1+1]
        load r18, DM1[r1+2]
        load r19, DM1[r1+3]

        move ar0, r0
        load r8,  DM0[ar0++]    ; x[0]
        load r9,  DM0[ar0++]
        load r10, DM0[ar0++]
        load r11, DM0[ar0++]  ; x[3]

;;;  To avoid memory accesses we unroll the entire loop
;;;  (Another way to solve this would be to place a shift register
;;;  for 4 samples in the MAC unit itself, although this variant is a
;;;  bit more general.)
        repeat endloop, 25

        mul.ss    acr, r8,r16
        mac.ss    acr, r9,r17
        mac.ss    acr, r10,r18
        mac.ss    acr, r11,r19
        lshift1 acr              ; Plenty of time, this allows us to split
        satrnd acr               ; this into two instructions.
        move   r31, HIGH(ACR)   ; If we wanted to save even more hardware
        store  DM0[ar2++], r31   ; complexity satrnd could be divided into
        load   r8, DM0[ar0++]    ; mac acr,#1,#0x4000 and sat acr

        mul.ss    acr, r9,r16    ; (That is, using the mac instruction to add
        mac.ss    acr, r10,r17   ; the roundvector rather than a specialized
        mac.ss    acr, r11,r18   ; instruction.)
        mac.ss    acr, r8,r19
        lshift1 acr
        satrnd acr
        move   r31, HIGH(ACR)
        store  DM0[ar2++], r31
        load   r9, DM0[ar0++]

        mul.ss    acr, r10,r16
        mac.ss    acr, r11,r17
        mac.ss    acr, r8,r18
        mac.ss    acr, r9,r19
        lshift1 acr
        satrnd acr
        move   r31, HIGH(ACR)
        store  DM0[ar2++], r31
        load   r10, DM0[ar0++]

        mul.ss    acr, r11,r16
        mac.ss    acr, r8,r17
        mac.ss    acr, r9,r18
        mac.ss    acr, r10,r19
        lshift1 acr
```

```
        satrnd acr
        move   r31, HIGH(ACR)
        store  DM0[ar2++], r31
        load   r11, DM0[ar0++]
endloop:
        ret

mult_32x32:
        load r4, DM0[r0]        ; LSB part val1
        load r5, DM0[r0+1]      ; MSB part val1
        load r6, DM0[r1]        ; LSB part val2
        load r7, DM0[r1+1]      ; MSB part val2

        mul.ss acr, r5,r7
        lshift16 acr           ; acr = acr << 16
        mac.su acr, r5,r6
        mac.su acr, r7,r4
        lshift16 acr
        mac.uu acr, r4,r6
        sat32 acr

        move r4,LOW(acr)
        move r5,HIGH(acr)
        store DM0[r2], r4
        store DM0[r2+1], r5
        ret
```



A 64 bit accumulator is selected to allow the entire result of the 64 bit multiplication to be stored in the accumulator before the value is saturated to 32 bits. To avoid the need for two saturation units we also ensure that the ACR is left shifted once before using saturation. (This could of course be integrated into the MAC unit, but there is plenty of time, so there is no need to do so in this particular case.)

```
always begin : G64
    out = {{32{in[31]}}, in[31:0]};
end
```

Control table

Cmode[1:0] tells whether signed/unsigned operation is used according to the following table:

	Cmode[1:0]
mul/mac.ss:	11
mul/mac.su:	10
mul/mac.us:	01
mul/mac.uu:	00
Non-mac/mul:	-

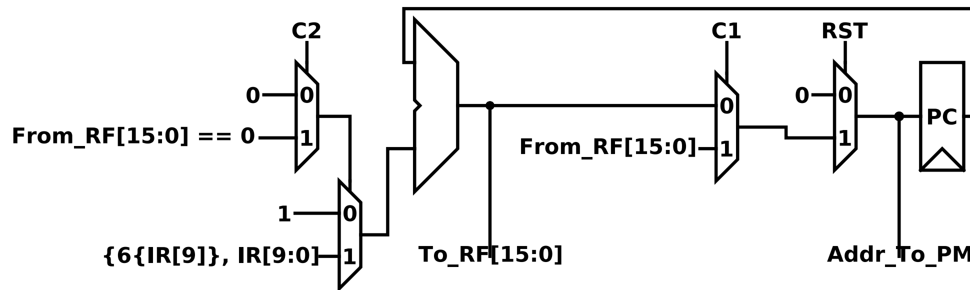
Operation	C0	C1	Cacr	Cshift	Cr
nop	-	-	0	-	-
mul.xx	0	0	2	-	-
mac.xx	1	0	2	-	-
sat32	1	1	3	-	-
satrnd	1	2	3	-	-
lshift1	-	-	1	0	-
lshift16	-	-	1	1	-
move rf, HIGH(acr)	-	-	0	-	0
move rf, LOW(acr)	-	-	0	-	1

Solution proposal, question 5

```
; ptr1 in ar0, ptr2 in ar1, ptr3 in r2
fir_kernel:
; Unroll loop to avoid need for repeat instruction

clr acr0

set r0,#8
loop:
add r0,r0,#-1
mac acr,DM0[ar0++], DM1[ar1++]
mac acr,DM0[ar0++], DM1[ar1++]
mac acr,DM0[ar0++], DM1[ar1++]
bne r0, loop    ; Jump if r0 is not equal to 0
mac acr,DM0[ar0++], DM1[ar1++]  ; Delay slot

        move r0, SATRND(acr)
bra r31 ; Link register
store DM1[r2], r0       ; Delay slot


handle_filter:
load r2, DM0[r0+2] ; Load r2 (ptr3) and r5 (ptr2) first so that the value
load r5, DM0[r0+1] ; is available for the bne:s further down even if the latency
load r4, DM0[r0]   ; of the memory is fairly high.
move ar0,r4

        bne  r2,continue
move ar1,r5    ; Delay slot

continue:
bne r5, noupdateptr2
load r5, DM0[r0+3]  ; Delay slot

move ar1,r5

noupdateptr2:
push r31 ; Store old value of link register
jal fir_kernel ; r31 = nextpc
nop            ; Delay slot

pop r31
nop        ; (Number of nops depends on the latency of the memory
nop
bra r31
nop        ; Delay slot
```

Proposed instruction format for branches:

```
100 xxxxx aaaaaaaaaaaa    ; BNE checks if register x is 0. If so jumps to PC with offset a
101 0aaaa aaaaaaaaaaaa    ; jump and link to address a
110 xxxxx 000000000000    ; jump to register x
```



To reduce the complexity of the hardware a link register is used to store the return address. This means that there are no limitations (aside from memory in DM0/DM1) on the number of subroutine calls that are possible to make. Similarly, only the bne instruction turned out to be needed here. (Although beq would be trivial to implement as well.) It is assumed that the instruction decoder ensures that the value presented on To_RF is saved to register r31 when running the jump and link instruction.

Control table:

	C1	C2
bne	0	1
jal rX,label	0	0
bra rX	1	0

```
; jump and link handling:

Address    Insn
0          nop
1          jal 5    ; <-- When this is located in IR
2          nop      ; <-- PC already points to this location
3          nop      ; Thus the PC that should be saved to the RF is this. (Which is
4          nop      ; why To_RF comes from the adder and not directly from PC.)
```

Revision history for v1.2

Difference from exam version

- Typo corrections
- Clarified that 130 accesses corresponded to 130 *read* accesses

Differences from v1.0

- Fixed bit-width annotations for MAC unit solution proposal
- Fixed formatting of assembler code for MAC unit solution proposal
- Fixed typos

Differences from v1.1

- Changed solution proposal to add did_sat signal in question 1.
- Clarified that modulo addressing was only required for a step size of one.
- Fixed a few typos.