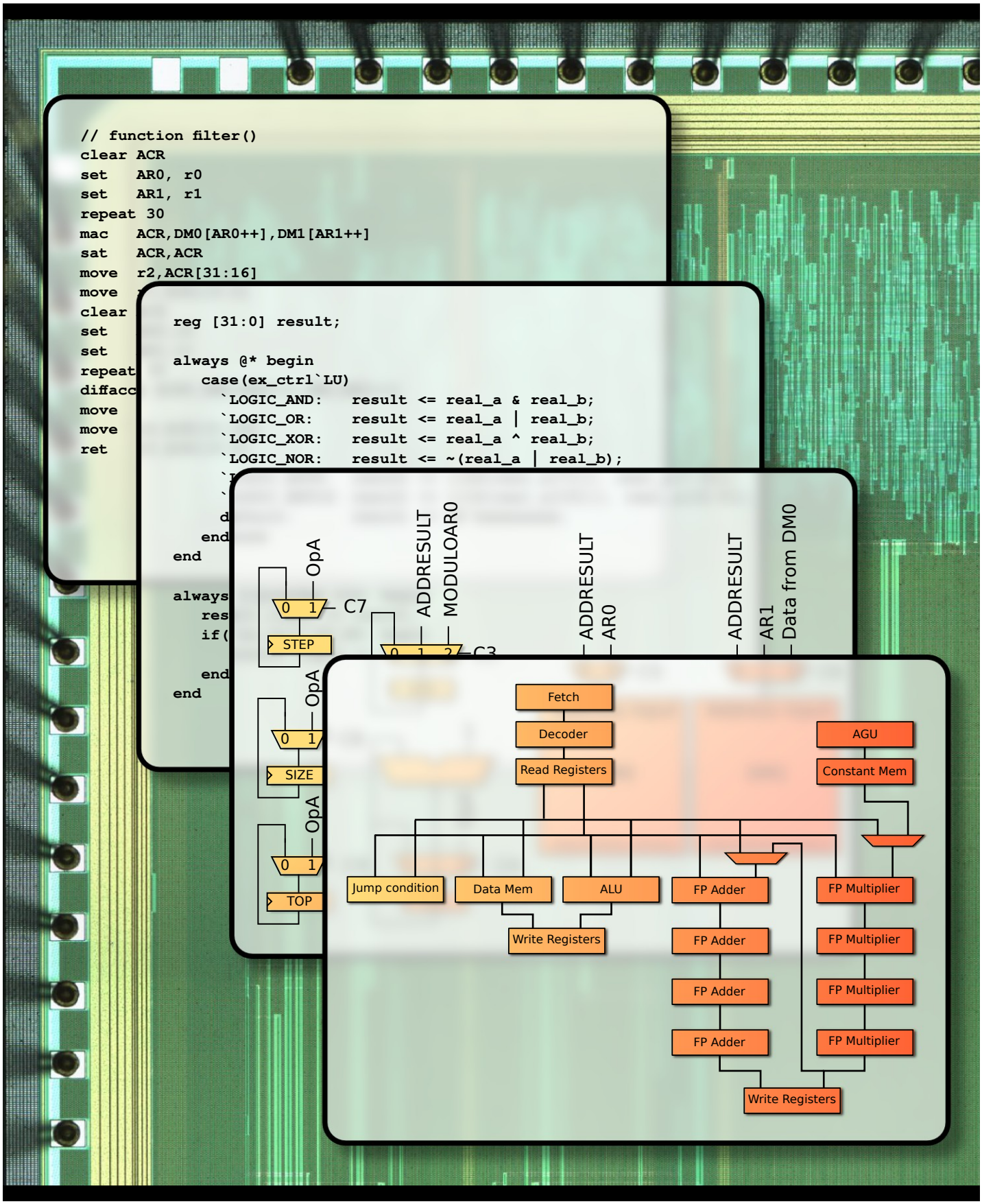
EXERCISE COLLECTION FOR TSEA26

Design of Embedded DSP Processors



Version 1.5.3

ANDREAS EHLIAR

with contributions from Oscar Gustafsson and Frans Skarman

Contents

0 About the Exercises

The exercises in this book are meant to show fairly realistic situations. While it is seldom feasible to create an exercise which is fully realistic, it has been tried to create exercises which contains situations that you could realistically expect to see in a real ASIP. Or for that matter when designing an accelerator for a certain task in for example an FPGA.

Many of these exercises can be solved in different ways, particularly the intermediate and challenging exercises. In the solution proposals, the aim is to provide a fairly realistic solution proposal. For example, often long critical paths or solutions which would require an awkward datapath pipeline are avoided. The hope is that this will lead to some more interesting solution proposals which will allow you to learn a new trick or two. (However, do not worry too much if you would not come up with these tricks yourself, as you typically do not need to worry about for example the critical path on the exam.)

Feedback on these exercises, particularly any bugs you may have found are very welcome.

0.1 Notes About the C/Pseudo Code in the Exercises

C-like pseudo code is used in the exercises in this document. This is hopefully not a problem as the C code is usually kept simple and should be easy to understand for anyone who has taken an introductory course in C or C++. However, datatypes from `stdint.h` are often used, which may not have been covered in such a course:

- `int8_t`: 8-bit signed integer
- `int16_t`: 16-bit signed integer
- `int32_t`: 32-bit signed integer
- `uint8_t`: 8-bit unsigned integer
- `uint16_t`: 16-bit unsigned integer
- `uint32_t`: 32-bit unsigned integer

(In essence, this is a way to guarantee that certain sizes are used, as C does not guarantee that for example a `short` will always be 16 bits.)

In these exercises it is assumed that two's complement format will be used to store the signed integers although the C standard does not offer such guarantees[1].

0.2 On the Use of RTL Code

Note that RTL code, in the form of Verilog, has been used to explain certain parts of the schematics that would be awkward to draw using schematic symbols.

It is encouraged to do the same while solving these exercises if you think a certain part of the schematic is easier to explain using Verilog or VHDL. However, keep the following in mind if you want to use Verilog/VHDL:

0.2.1 Syntax

When correcting the exam, the correct Verilog/VHDL syntax is in general not crucial. However, there are a few situations where the syntax is quite important. The most common example is the replication operator in Verilog.

```
// Correct attempt to check that x contains only ones:
if (x[4:0] == {5{1'b1}}) // This is using the replication
                         // operator
   // Do something

// Incorrect attempt (This checks whether x holds the value
// of 1)
if (x[4:0] != 5'b1) // No replication operator here!
   // In this case there will be points deducted since it is
   // not possible to tell if you do not know Verilog syntax
   // (not a big deal) or if you do not know that you are
   // supposed to check that x has the value of 1 or 31...


// Correct (but is if x contains lots of ones)
if (x[4:0] == 5'b11111)
   // Do something
```

If you are unsure about the syntax, please write a comment about what you actually intend to do.

[1] Which can lead to disastrous results, see for example `http://www.phrack.org/issues.html?issue=60&id=10#article`.

0.2.2 Hardware Multiplexing

If you use Verilog or VHDL you need to write it in such a way that the hardware multiplexing is obvious. For example, the following is a **bad** example of a design since the hardware multiplexing is not obvious:

```
/* Create a simple ALU with add, subtract, xor and or */
reg [31:0] result;
always @(posedge clk) begin : ALU
    case(Ca)
        0: result = A + B;
        1: result = A - B;
        2: result = A ^ B;
        3: result = A | B;
    endcase
end
```

In the example above, it is not clear that the same adder can be used for the addition and subtraction. Although a good synthesis tool will probably detect this, one of the the goals of this course is to show that you understand the details of microarchitecture design (such as hardware multiplexing), instead of relying on the tools to do it for you. While the tools will probably do a good job in many cases, you will see in lab 2 that good knowledge of hardware multiplexing can reduce the area and critical path of a design by allowing you to redesign it in such a way that it is possible to do hardware multiplexing.

In contrast, if you want to create an ALU where hardware multiplexing is obvious, the following would be a better example:

```
reg carry_in;
always @* begin
    if(Cb) begin
        carry_in = 1;
        op_b = B;
    end else begin
        carry_in = 0;
        op_b = ~B;
    end
end

always @(posedge clk) begin
    case(Cc)
        0: result = A + op_b + carry_in;
        1: result = A ^ B;
        2: result = A | B;
    endcase
end
```

0.2.3 Clocked structures in HDL

Another detail that you need to be careful about is clocking. Historically, many students that use VHDL or Verilog on the exam forget to indicate that a clock is used for registers and flip-flops. So if you want to create a clocked structure (such as a flag register in an ALU) please make sure that you actually indicate that a clock is used (for example by using an always `@(posedge clk)` statement in Verilog).

1 Introductory Exercises

Exercise 1.1 (easy)

Draw the schematic of a simple computing module which should perform the following tasks, depending on the value of the control signal `C`:

- `C=0:  Y = (A + B)/2`
- `C=1:  Y = (A - B)/2`

No special rounding is needed in this exercise. However, you are only allowed to use one adder.

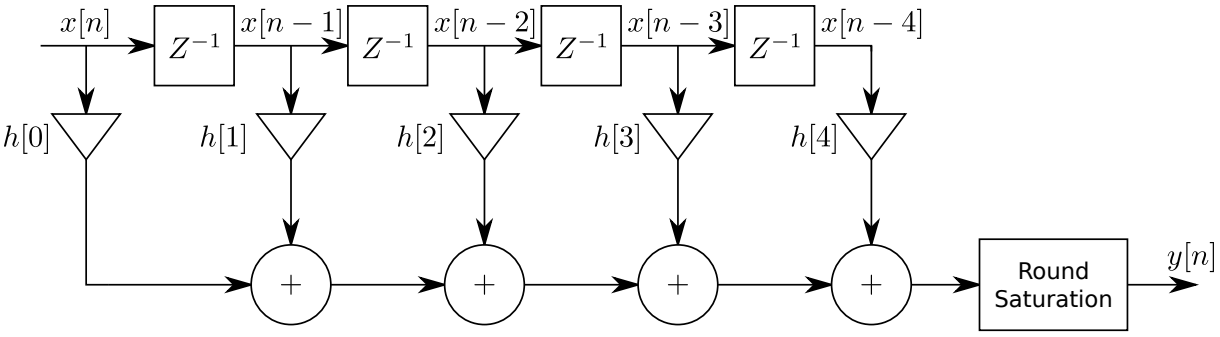
Exercise 1.2 (intermediate)

Write efficient assembler code for a DSP processor such as the Senior processor for a 32-tap FIR filter. The FIR filter should operate on a 1000 long vector located in memory 0. (You can assume that this vector begins with 31 zeros and ends with 31 zeros.). The filter coefficients are stored in ROM0.

Exercise 1.3 (intermediate)

Write efficient assembler code for a DSP processor such as the Senior processor for the following filter structure.

For a 5-tap filter it will be significantly faster to use an unrolled loop and store intermediate values in registers rather than using a ring buffer in memory.



You can assume that the following instruction will handle output and inputs of samples:

```
in r0,0x10          ; Fetch sample
out 0x11,r0         ; Output sample
```

2 Exercises for Arithmetic and Logic Unit (ALU)

Exercise 2.1 (easy)

Design an arithmetic unit using minimal HW and implement the following functions. Annotate your design clearly.

- OP1: A[3:0] + B[3:0] + Carry_in
- OP2: Saturate(A[3:0] + B[3:0])
- OP3: Saturate(A[3:0] – B[3:0])
- OP4: if(A[3:0] $\geq$ B[3:0]) result $\Leftarrow$ A else result $\Leftarrow$ B;

Clearly specify all the black boxes. Operand A and B are signed numbers.

Exercise 2.2 (easy)

- OP0: `RESULT = A + B + Carry_in`
- OP1: `RESULT = A - B`
- OP2: `RESULT = ABS(B - A)`
- OP3: `RESULT = LEFTSHIFT(A, B[2:0])` (Shift A left by B[2:0] times)
- OP4: `RESULT = RIGHTSHIFT(A, B[2:0])` (Shift A right by B[2:0] times (not arithmetic shift))

Constraints:

- Overflow must be handled for `ABS(B - A)`. It is up to you if you want to handle overflow in any other case.

- You should minimize the amount of hardware such as adders and shifters.
- `A` is 8 bits wide, `B` is 8 bits wide, `Carry_in` is 1 bit wide. `RESULT` is 8 bits wide.

Exercise 2.3 (intermediate)

We have a processor with a pipeline where we can:

- Read out two operands from the register file and write one operand to the register file, all at the same time
- Instead of reading out one of the operands you can choose to take a 16-bit immediate from the instruction word
- We have 32 16-bit registers
- A conditional branch takes 3 clock cycles
- We have a repeat instruction
- We have only one load instruction of interest:

```
load Rd, DM0[AR0++]    ; AR0 is set with the instruction
                       ; set AR0, Rs
```

- The store instruction works the same way:

```
store DM0[AR0++], Rs
```

- After a load instruction we must wait a clock cycle before we can use the result

Select an instruction set suitable for the following two functions and translate the functions into assembler. Draw a schematic and a control table for the ALU.

```
// Maximum execution time: 105 clock cycles (excluding ret insn)
int16_t dct_indata[32];
uint16_t find_maxabsval(void) // Return value should be in r0
{
  uint16_t biggest = 0, b;
  int16_t a;

  for(int i=0; i < 32; i++){
    a = dct_indata[i];
    b = abs(a);
    if(b > biggest) biggest = b;
  }
  return biggest;
}
```

```
// Maximum execution time for update_statistics: 25 clock cycles
// (not including the RET instruction)
int64_t packet_ctr;
void update_statistics(int16_t length)
/* Length is in register r0 when this function is called */
{
  packet_ctr += length; // Hint: This is a signed computation!
}
```

Exercise 2.4 (challenging)

If you want a challenging bonus task: Would it be possible to create an ALU which enables the `find_maxabsval` function to execute in less than 80 clock cycles under the constraints listed in exercise 2.

Would it be possible to do it in less than 50 clock cycles under the same constraints? If not, what is the smallest change to the constraints that you would like to do in order to run `find_maxabsval` in less than 50 cycles?

3 Execrcises for Multiply-and-Accumulate (MAC) Unit

Exercise 3.1 (easy)

Design a MAC unit capable of the following operations:

OP0: No operation
OP1: `ACR = 0`
OP2: `ACR = A * B` (Fractional multiplication (signed))
OP3: `ACR = A * B + ACR` (Fractional multiplication (signed))
OP4: `ACR = 1.25 * ACR` (Scaling)
OP5: Load `ACR` with a fractional value from a register
OP6: `ACR = SATURATE(ROUND(ACR))`
OP7: `RF = ACR[7:0]`
OP8: `RF = ACR[15:8]`
OP9: `RF = SIGNEXTEND(ACR[19:16])`

Constraints:

- `A` and `B` are 8 bits, registers are 8 bits
- `ACR` is 20 bits (including 4 guard bits).
- **Only one multiplier may be used. You should select as small a multiplier as necessary. You also need to annotate whether it is signed or unsigned.**
- Rounding is performed in such a way that OP8 can be used to read out the saturated and rounded result.

Tasks:

a) Draw a hardware schematic for your MAC unit. You must annotate the bit width of all signals except mux control signals.

b) Draw a control table for your MAC unit where you include all operations defined above.

Exercise 3.2 (intermediate)

You have been tasked with accelerating a piece of code which contains a lot of **complex valued** multiplications, additions, and subtractions. A complex value is stored in a normal register by putting the imaginary part into the MSB part of the register and the real part in the LSB part of the register.

Your task is to make sure that the following piece of code can execute in **at most 7 clock cycles**. All operations in this code are done on complex valued data!

```
o0 = e0 * f0 - e1
o1 = e0 + e1
o2 = e2 * f1 - e3
o3 = e2 + e3
```

Constraints:

- You need to be able to execute the excerpt listed above in at most 7 clock cycles.
- You can assume that all operands will be present in the register file. You can also assume that all results should be written back to the register file.
- You do not have to worry about overflows. (The person who designed the code has guaranteed that there will never be any overflows.[1])
- You are only allowed to use 2 hardware multipliers

Inputs:

`OpA[31:16]`	Imaginary part of first operand from register file in fractional format
`OpA[15:0]`	Real part of first operand from register file in fractional format
`OpB[31:16]`	Imaginary part of second operand from register file in fractional format
`OpB[15:0]`	Real part of second operand from register file in fractional format
Control signals	(decided by you in the control table)

Outputs:

`TO_RF[31:16]`	Imaginary part of the result
`TO_RF[15:0]`	Real part of the result

[1] Famous last words... (read about flight 501 of Ariane 5 if you are interested in an expensive real example of this).

Tasks:

a) Select a suitable instruction set for your module.

b) Draw a schematic of your accelerator.

c) Draw a control table for your accelerator.

Exercise 3.3 (intermediate)

Design a MAC unit capable of supporting the following two functions:

```
// Input data in r0-r5 are in 16-bit fractional format
// Output data in r6 and r7 should be in 16-bit fractional format
//
// Intermediate calculations are performed using a sufficient
// number of bits, so that SAT() will be able to detect an
// overflow.
function butterfly_part()
    r6  = SAT(ROUND(r0 * r2 - r1 * r3 - r4))
    r7  = SAT(ROUND(r0 * r3 + r1 * r2 - r5))
endfunction

// * DM0 and DM1 are 16 bits wide and contains signed integers in
//   this example.
// * sumofproducts and sumofdiff are signed integers that are
//   "wide enough" (no overflow should occur during the repeat
//   loop)

function filter(buffer)
    AR0 = r0 // Use the instruction set AR0,r0
    AR1 = r1 // Use the instruction set AR1,r1

    sumofproducts = 0
    sumofdiff = 0

    repeat(30)
        sumofproducts += DM0[AR0] * DM1[AR1]
        sumofdiff += abs(DM0[AR0++] - DM1[AR1++])
    endrepeat

    if(sumofproducts > 0x7fffffff) then
        sumofproducts = 0x7fffffff
    else if(sumofproducts < -0x80000000) then
        sumofproducts = -0x80000000
    endif

    // Read out the 32-bit result to general purpose registers
    r2 = sumofproducts[31:16]
```

```
    r3 = sumofproducts[15:0]
    r4 = sumofdiff[31:16]
    r5 = sumofdiff[15:0]

endfunction
```

Inputs to this module:

`DM0_result[15:0]`	Data from memories
`DM1_result[15:0]`	
`OpA[15:0]`	Operands from the register file
`OpB[15:0]`	
Control signals	(created by you in your control table)
(clk)	(And, of course, a clock signal)

Outputs from this module:

`TO_RF[15:0]`	This is sent to the register file writeback port

Constraints:

- The register file, DM0, and DM1 are 16 bits wide.
- It is up to you to decide how many accumulators you will need and how large they should be.
- The other parts of the processor have enough features to support these assembly programs (that is, the PC has a repeat instruction, the AGU have all relevant addressing modes, etc)
- The function `butterfly_part()` should be executed in at most 15 clock cycles (excluding the return instruction)
- The function `filter()` should be executed in at most 80 clock cycles (excluding the return instruction)

Tasks:

a) Select an instruction set for your MAC unit and write assembly programs for both functions. You should also decide how many accumulator registers you will need and how wide such a register should be.

b) Draw a schematic of your MAC unit (including a control table).

Exercise 3.4 (easy)

Same as exercise 3.3 except that you may use 4 hardware multipliers.

Exercise 3.5 (challenging)

Same as exercise 3.3 except that you are only allowed to use one hardware multiplier.

Use Gauss' complex multiplication algorithm.

Exercise 3.6 (intermediate)

Consider the following two functions:

```
// Pointers to the start of the indata, outdata, and state arrays
// are passed in r0-r2
function biquad(indata, outdata, state)
   m0 = state[0] // m0 ... m3 are signed 16-bit integers
   m1 = state[1]
   m2 = state[2]
   m3 = state[3]
   i=0
   // tmp should be wide enough to handle the calculations
   // without any possibility of overflow
   repeat 100
     tmp = coeff0 * indata[i]   // coeff0 ... coeff4 are all
     tmp = tmp + coeff1 * m0    // signed 16-bit integer
     tmp = tmp + coeff2 * m1    // constants
     tmp = tmp + coeff3 * m2
     tmp = tmp + coeff4 * m3

     tmp = tmp + 8192
     tmp = tmp >> 14            // Arithmetic right shift

     if tmp > 32767 then
       tmp = 32767
     else if tmp < -32768 then
       tmp = -32768
     endif

     m3 = m2
     m2 = tmp
     m1 = m0
     m0 = indata[i]
```

```
      outdata[i] = tmp

      i=i+1
  endrepeat

  state[0] = m0
  state[1] = m1
  state[2] = m2
  state[3] = m3
endfunction


// Note: vector1 and vector2 are both in fractional format.
// You can select which memory to place each vector in.
function dotproduct(vector1, vector2)
  ACR = 0
  i = 0
  repeat 10
    ACR = ACR + vector1[i] * vector2[i]
    i = i + 1
  endrepeat

  return SAT(ROUND(ACR)) // The value should be returned in r0
endfunction
```

Constraints:

- The `biquad()` function should execute in less than 750 clock cycles (excluding the return instruction)
- The `dotproduct()` function should execute in less than 16 clock cycles (excluding the return instruction)
- There are two single port data memories which are both 16 bits wide. However, you can decide which memories to use for `indata`, `outdata`, and `state` arrays.
- The general purpose register file has two read ports and one write port. The registers are 16 bits wide.
- An instruction word includes space for a 16-bit immediate
- There should be at least one 40-bit accumulator register in your MAC unit.
- You can select what inputs and outputs your MAC unit should have, based on the constraints above.
- You are only allowed to use one 17×17-bit signed multiplier.

Tasks:

a) Select an instruction set which is suitable for your MAC unit and translate the functions above into assembler.

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

Exercise 3.7 (challenging)

Same as exercise 3.6 except that you should run the `biquad` function in less than 450 clock cycles.

4 Exercises for Program Flow Control (PFC) Unit

Exercise 4.1 (intermediate)

A schematic of a processor is shown in Fig. 4.1. The datapath is 16 bits wide and there are 16 registers, `r0` to `r15`. `OpB` will always be set to the contents of a register whereas `OpA` can be set to either a register or the 16 least significant bits of the instruction word depending on whether bit 16 of the instruction word is 0 or 1, as shown in the figure. The program memory contains 4096 (2^{12}) instructions. The following instructions are already implemented:

Instruction	Explanation	Instruction	Explanation
`nop`	No operation	`set OpW, immediate`	`OpW = immediate`
`add OpW,OpA,OpB`	`OpW = OpA + OpB`	`load OpW, OpA`	`OpW = DM[OpA]`
`sub OpW,OpA,OpB`	`OpW = OpA - OpB`	`store OpA, OpB`	`DM[OpA] = OpB`

There are also 16 program flow control instructions implemented in the decoder, tentatively named `PFCOP0` ... `PFCOP15`. When a jump instruction is encountered, the decoder will set the signal `PFC_OP[4]` to 1. Additionally, `PFC_OP[3:0]` is set to 0 if `PFCOP0` was decoded, 1 if `PFCOP1` was decoded, and so on. If a jump instruction is not decoded, the value of `PFC_OP[3:0]` is undefined. Each `PFCOP` instruction may (but do not have to) use `OpA` and `OpB`. The 12 least significant bits (`PFC_DATA`) in the instruction word can be used as an absolute address, pc relative address, loop counter setting, etc.

Tasks:

a) What is the minimal amount of delay slots an unconditional jump instruction will have if the `FORCE_NOP` signal is deactivated?

b) Write assembly code to calculate the sum of `r0`, `r1`, and `r2`, using a minimum amount of instructions.

The pipeline has no forwarding or register bypass! (And the register file is implemented using write after read.)

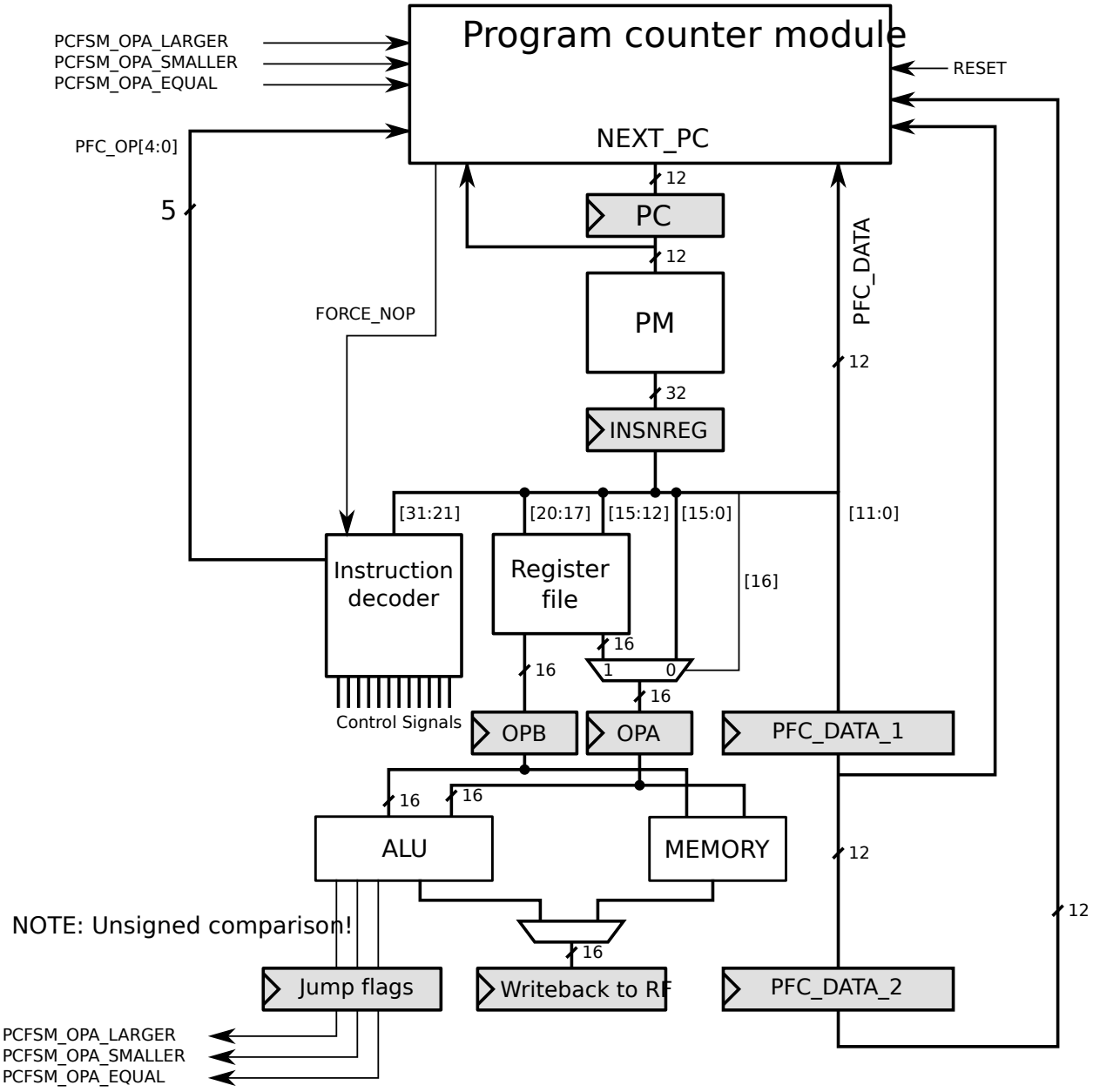


Figure 4.1: Processor schematic.

c) The following two programs need to execute in less than 200 cycles each. The size of program 1 must be less than 20 instructions. The size of program 2 must be less than 25 instructions. Decide on which program flow control instructions you will need to fulfill these constraints. You may create up to 16 program flow control instructions. (There are 4 bits in `PFC_OP[3:0]`.) You also need to describe how many delay slots each instruction has. Finally, you should write assembly code for both programs.

```
// Program 1:
// x is in r0
// flag is in r1
if x >= 10
    x = x + 55
else
    x = x + 48
    if flag == 1
        x = x + 32
    end if
end if
```

```
// Program 2:
// inptr is in r0,
// outptr in r1
repeat 36
   x = DM[inptr]
   inptr = inptr + 1
   DM[outptr] = x
   outptr = outrptr + 1
endrepeat
```

d) Draw a hardware schematic of the program counter module. You may use multiplexers, registers, adders, FSMs (finite state machines), and logic gates. If you use an FSM, you must include the state diagram of the FSM.

Signal	Explanation
`FORCE_NOP`	This signal makes the instruction decoder believe that it is decoding a NOP instruction.
`NEXT_PC`	The next value for the program counter.
`PC`	The current value of the program counter.
`PCFSM_OPA_LARGER`	Set to 1 if OpA is larger (unsigned) than OpB
`PCFSM_OPA_SMALLER`	Set to 1 if OpA is smaller (unsigned)than OpB
`PCFSM_OPA_EQUAL`	Set to 1 if OpA is equal to OpB
`PFC_DATA`, `PFC_DATA_1`, `PFC_DATA_2`	The twelve least significant bits in the instruction word. These signals are delayed 0, 1 or 2 clock cycles.
`PFC_OP[4]`	1 if a jump is decoded, 0 otherwise
`PFC_OP[3:0]`	The kind of jump that was decoded.
`RESET`	System reset. Will be asserted for at least 16 cycles.

Exercise 4.2 (challenging)

Same as exercise 4.1 with the following modifications:

- Program 1 and 2 must execute in less than 165 clock cycles each.
- The size of program 1 must be less than 12 instructions
- The size of program 2 must be less than 20 instructions

Exercise 4.3 (easy)

Design a PC module capable of the following operations:

OP1: PC++
OP2: PC = PC + signextend(Immediate)
OP3: PC = PC
OP4: PC = RF
OP5: Push(PC); PC = PC + signextend(Immediate)
OP6: Push(PC); PC = RF
OP7: PC = Pop()
OP8: if(flag) PC = PC + signextend(Immediate) else PC++
OP9: if(!flag) PC = PC + signextend(Immediate) else PC++

Inputs to your PC module:

- `Immediate[9:0]`: Jump target offset from the instruction word
- `RF[15:0]`: Input from register file
- `Flag`: 1-bit signal from ALU
- All control signals that you specify in the control table
- Reset signal (which will be asserted for at least 10 clock cycles)
- (And, of course, a clock signal)

Outputs from your PC module:

- `PC[15:0]`: Program counter value sent to the program memory)

Other constraints:

- After the processor is reset, the processor should start executing at address 0
- You must also have a hardware stack in your PC module capable of holding two entries.

Also, note that for this exercise you do not have to worry about the rest of the pipeline. That is, you can assume that the instruction decoder has already dealt with delay slots, setting the flag in the ALU, etc.

Tasks:

a) Draw a schematic of your PC module.

b) Draw a control table for your PC module.

Exercise 4.4 (intermediate)

During the design of a simple DSP processor, the following functions have been determined to be the most important:

```
function main()
    while(TRUE)
        // get_packet() returns the return value in register r0
        pkt_type = get_packet()
        if (pkt_type <= 0) then
            logerror()
        else if(pkt_type <= 9) then
            worker()
        else if(pkt_type >= 59) then
            guiworker()
        else
            logerror()
        end if
        update_outputs()
    endwhile
endfunction

function worker()
    optype = get_packet_operation() // Returns packet pointer in
                                    // r0
    if (optype == 0) then
        DM0[95] = sum()
    else if (optype == 1) then
        DM0[96] = diff()
    else
        logerror()
    endif
endfunction


// get_current_length(), get_current_buffer() and
// get_previous_buffer() executes in 10 clock cycles each.
// They all return the return value in register r0.

// Must execute in at most 130 clock cycles
function sum()
    ptr = get_current_buffer() // ptr is returned in r0
    tmp = 0
```

```
    for (i=0; i < 100; i = i + 1)
        tmp = tmp + DM0[ptr + i]
    endfor
    return SAT(tmp)
endfunction

// Must execute in less than 50 + length*3 clock cycles.
function diff()
    tmp = 0
    length = get_current_length()
    ptr1 = get_current_buffer()
    ptr2 = get_previous_buffer()
    for (i=0; i < length; i = i + 1)
        tmp = tmp + ABS(DM0[ptr1 + i] - DM1[ptr2 + i])
    endfor
    return tmp
endfunction
```

Your task is to specify how the program flow control (PFC) unit should work.

Information:

- Unconditional branches have one delay slot
- Due to pipeline delays, conditional branches have two delay slots
- Register indirect branches have three delay slots
- The program flow control instructions may use the negative, zero, and saturation flag from the ALU.
- There is no requirement to minimize the amount of PFC instructions or the size of the assembler program. (But you probably want to do that anyway to minimize the chance of any error.)
- The processor is a single scalar processor able to issue one instruction per clock cycle.

Tasks:

a) Select the PFC instructions that you will require. For each instruction that you select you must describe exactly what the instruction does using pseudo code.

b) Using these PFC instructions, implement the program shown above in assembler code. In addition to the PFC instructions that you selected in a), you may use any instruction commonly found in a single scalar DSP processor such as Senior without having to describe how it works (such as normal ALU instructions, MAC instructions, etc). You do not have to implement the functions that are not shown above (e.g. `logerror()`, `get_packet_operation()`, etc).

5 Exercises for Address Generation Unit (AGU)

Exercise 5.1 (easy)

Draw a schematic and a partial control signal table for an address generator unit.

It should contain five registers:

`AR0[15:0]`	Address register 0
`AR1[15:0]`	Address register 1
`BOTTOM[15:0]`	Lower address for modulo addressing
`TOP[15:0]`	Higher address for modulo addressing
`STEP[15:0]`	Configurable step size

Output:

`ADDRESS[15:0]`	This is the address to the data memory

Inputs:

`IMMEDIATE[15:0]`	Constant data carried by the instruction
`RF[15:0]`	Data from the general register file
`OPERATION[3:0]`	Operation performed by the AGU based on the table below

OPERATION	**Operation to be performed**
0000	Load AR0 from general register file
0001	Load AR1 from general register file
0010	Load STEP from the general register file
0011	NOP
0100	Direct addressing (the address is based on immediate data in the instruction)
0101	Indirect addressing (the address is based on a general purpose register)
0110	Post increment of AR0 (step size is based on the STEP register)
0111	Post increment of AR1 (step size is based on the STEP register)
1000	AR0 + immediate
1001	AR1 + immediate
1010	Modulo addressing on AR0 with post increment (step size is 1)
1011	Modulo addressing on AR1 with post increment (step size is 1)
1100	NOP
1101	NOP
1110	Load bottom register from RF
1111	Load top register from RF

Your hardware should support all of these operations, but your control signal table only needs to show the control signal configuration for operation 0000, 0100, 0110, 1000, and 1011.

Exercise 5.2 (intermediate)

In this exercise you should create an address generator unit. It should support the addressing modes necessary to execute the following three functions under the specified clock cycle constraints.

```
// This function must execute in less than 20 clock cycles.
// Inputs: ptr is passed in register r0
function dot13(ptr)
    ptr0 = DM0[ptr+31]
    ptr1 = DM0[ptr+33]
    resultptr = DM0[ptr+35]
    tmp = 0
    for (i = 0; i < 13; i = i + 1)
        tmp += DM0[ptr0++] * DM1[ptr1++];
    endfor
    DM0[resultptr] = SAT(ROUND(tmp))
endfunction

// This function must execute in less than 140 clock cycles
// ptr0 is passed in register r0, ptr1 in r1, buffersize in r2,
// last in r3, step in r4
// Note: You can assume the following: buffersize/8 > stepsize
function filter(ptr0, ptr1, buffersize, last, step)
```

```
    tmp = 0
    for(i=0; i < 128; i = i + 1)
        tmp += DM0[ptr0+=step] * DM1[ptr1++]
        if (ptr0 > last) then
            ptr0 = ptr0 - buffersize
        endif
    endfor
endfunction


// This function must execute in less than 700 clock cycles
// Inputs: ptr is passed in register r0
function interleaver(ptr)
    ptr0 = DM0[ptr+31]
    ptr1 = DM0[ptr+33]
    for(i=0; i < 256; i = i + 1)
        // You may decide whether to use DM0 and/or DM1 for the
        // following memory acceses:
        idx = DMx[ptr0++]
        tmp = DMx[idx]
        DMx[ptr1++] = tmp
    endfor
endfunction
```

Additional information:

- A memory address is 16 bits wide, and the memories are word-addressable
- Both DM0 and DM1 are 16-bit single port memories
- There are 32 general purpose registers that are 16 bits wide
- An instruction has two operands, OpA and OpB. OpA will always come from a register and OpB may come either from a register or as a 16-bit immediate from the instruction word
- You are allowed to use up to four adders (or comparators) in your unit
- You may assume that the datapath has full forwarding (unless you do something in your part of the design which would prohibit forwarding in some situations)
- One instruction can be issued (in-order) each clock cycle
- You may assume that all other paths are implemented in such a way that the functions above can be implemented, e.g., there is a `repeat` instruction, and the MAC instruction can load both operands directly from memories
- The functions must contain fewer than 100 instructions

Tasks:

a) Select the addressing modes you will need to implement these three functions under the specified clock cycle constraints.

b) Draw a schematic of your AGU. In this schematic you must also include how the memories are connected to the AGU. It is not very realistic, but you may use asynchronous memories if you want to. You should also include a control table for your AGU.

c) Write assembly code for the `filter()` and `interleaver()` function using the addressing modes supported by your AGU. (You do not need to write assembler code for the `dot13()` function, but your hardware should be capable of supporting that function anyway under the given clock cycle constraints.)

Be careful to avoid hazards by rearranging code or inserting NOP instructions as appropriate!

Exercise 5.3 (intermediate)

Profiling has shown that the following two functions are important for a certain application. Your task is to design an AGU that is capable of supporting them under the constraints given below.

```
function FIR_FILTER(samplesptr, bottom, top, coeffptr)
    repeat(128)
        ACC = ACC + dm0[samplesptr] * dm1[coeffptr]

        samplesptr = samplesptr + 1
        if samplesptr == top
        then
            samplesptr = bottom
        endif
        coeffptr = coeffptr + 1
    endrepeat
endfunction

function STORE_VAL(addr, value)
    parameter = dm0[59]
    dm0[addr] = value
    dm0[addr+2] = parameter
    dm0[addr+4] = parameter
    dm0[addr+8] = parameter
    dm0[addr+16] = parameter
    dm0[addr+32] = parameter
endfunction
```

Constraints:

- `STORE_VAL()` must execute in less than 12 clock cycles. `FIR_FILTER()` must execute in less than 140 clock cycles.
- The processor is a single scalar pipelined DSP processor which issues one instruction each clock cycle. (Like Senior.) You do not need to worry about pipeline penalties for any kind of jump in this exercise however.
- Function parameters are passed in general purpose registers
- You can assume that all other parts of the processor can handle the clock cycle requirements listed above. E.g., the MAC unit is connected to each memory, etc. This exercise is only about designing the AGU.
- The general purpose register file has two read ports and one write port. The memories are single ported.
- You should minimize the amount of hardware in your AGU by for example making use of the fact that dm0 and dm1 will not require the same addressing modes.

Exercise 5.4 (challenging)

Same as exercise 5.2, except that you also have the following constraints:

- You may only use synchronous memories
- You may only use one pipeline stage for the memory access (e.g., you cannot connect the output of one memory to the input of the next memory and then write the result back to the register file in one instruction)

6 Exercises for Accelerated Custom Instructions

Exercise 6.1 (intermediate)

The following function should be implemented on the Senior processor.

```
// matrixptr is in r0, vectorptr in r1 and resultptr in r2
function rotate_vector(matrixptr, vectorptr, resultptr)
    A = dm0[matrixptr]
    B = dm0[matrixptr+1]
    C = dm0[matrixptr+2]
    D = dm0[matrixptr+3]
    repeat 50
        X = dm0[vectorptr++]
        Y = dm0[vectorptr++]
        ROTATEDX = A*X+B*Y
        ROTATEDY = C*X+D*Y
        dm1[resultptr++] = ROTATEDX
        dm1[resultptr++] = ROTATEDY
    endrepeat
endfunction
```

Constraints:

- `matrixptr`, `vectorptr`, and `resultptr` are available in general purpose registers when the function is called.
- `A`, `B`, `C`, `D`, `X`, `Y`, `ROTATEDX`, and `ROTATEDY` are 16-bit fractional numbers.
- You do not need to worry about saturation and rounding in this exercise.
- You may not add any ports or change the width of either DM0 or DM1.

Tasks:

Your task is to create a special unit present in the pipeline of the Senior processor so that the function `ROTATE_VECTOR` can be executed in less than 130 clock cycles. You will also need to select suitable instructions to implement this function. You can assume that the processor has the required program flow control and addressing modes required to support this function.

a) Select a set of new instructions that will allow you to execute `ROTATE_VECTOR` in less than 130 clock cycles and translate `ROTATE_VECTOR` into assembler. For each of your new instructions you also need to describe any AGU or memory operation that it may perform.

b) Draw a hardware schematic of your vector rotation unit. You should minimize the amount of multipliers. You may also use as many gates, multiplexers and adders as you want to (within reason).

c) Draw a control table for your hardware where you include all instructions that you selected in task a).

Note: In a real scenario you would probably reuse some hardware in the MAC unit for the vector rotation instruction(s). For simplicity reasons we ignore this fact in this exercise.

Exercise 6.2 (intermediate)

The function `fir_3()` is responsible for 70% of the time in a hypothetical application running on the Senior processor. Your task is to evaluate the hardware cost of speeding up this function by designing a custom instruction that is able to execute `fir_3()` in one clock cycle. Additionally, it is necessary to initialize the values used by the `fir_3()` function by using the `initfir_3()` function. However, it is expected that the `fir_3()` function will be executed around 1000 times as often as the `initfir_3()` function. This means that the `initfir_3()` function does not need to execute quickly.

```
function fir_3()
    samples[2] = samples[1]
    samples[1] = samples[0]
    samples[0] = dm0[inputptr]
    inputptr = inputptr + 1

    tmp = 0
    tmp = tmp + samples[0] * coefficients[0]
    tmp = tmp + samples[1] * coefficients[1]
    tmp = tmp + samples[2] * coefficients[2]

    dm1[outputptr] = SATURATE(tmp)
    outputptr = outputptr + 1
endfunction
```

```
function initfir_3(val1, val2, val3, val4, val5)
    samples[0] = 0
    samples[1] = 0
    samples[2] = 0
    inputptr = val1
    outputptr = val2
    coefficients[0] = val3
    coefficients[1] = val4
    coefficients[2] = val5
endfunction
```

Constraints:

- Function parameters are passed in general purpose registers
- `samples` contains 16-bit values in signed integer format
- `coefficients` contains 16-bit values in signed integer format
- The `tmp` variable has a suitable number of guard bits.
- You do not need to pipeline this unit for maximum clock frequency.
- You should be able to issue one `fir_3()` instruction every clock cycle.

Tasks:

Your task is to design and implement the function `fir_3()` as a special instruction on the Senior processor. You also need to implement support for the `initfir_3()` function and you will probably need to add a couple of instructions to do this.

a) Draw a hardware schematic of the modified parts of the pipeline. You do not need to annotate bit widths. You do not need to annotate the contents of a *SATURATE* box.

b) Draw a control table for your hardware where you include a `NOP` instruction, the `fir_3()` instruction and the instructions necessary to implement `initfir_3()`. You should also write pseudo assembler code for `initfir_3()`.

Exercise 6.3 (intermediate)

Based on previous experience, a function called `Read1bit()` can be fairly expensive for an MP3 decoder. Your task is to evaluate the cost of implementing this function in software and hardware.

The function below is written to simplify assembler implementation, a good hardware implementation may look slightly different.

```
// Read 1 bit from a bitstream in memory and increment the bitpointer
function Read1bit()
   Memval = memory[CurrentAddress] // The memory is 16 bits wide

   // Calculate 2 to the power of CurrentBit
   Bitmask = 1 << CurrentBit

   // Check if bit number CurrentBit is set in Memval
   if Memval & Bitmask
      Bit = 1
   else
      Bit = 0
   endif

   // Advance bit position (and memory pointer if necessary)
   CurrentBit = CurrentBit + 1
   if CurrentBit > 15
      CurrentBit = 0
      CurrentAddress = CurrentAddress + 1
   endif

   return Bit // Return value is located in R0
endfunction
```

Tasks:

a) Write pseudo assembler code for `Read1bit` on a typical DSP or RISC processor (like Senior).

b) Estimate how many clock cycles the `Read1bit` function will execute in for the best and worst case. (You will need to make and document reasonable assumptions about your processor.)

c) Your task is to implement `Read1bit` as a single instruction. Your task is to figure out what units in the DSP processor you would have to modify to do this and show what kind of hardware you would add to these units.

7 Design Challenge: Design a DSP Processor

This is intended as a challenge which will allow you to check whether you have not only understood the individual parts but also verify that you can use all of these parts together. The difficulty level of this challenge is quite high, but since you can approach this exercise in a few different ways, I believe that it will be of use for all TSEA26 students.

7.1 Approach 1: Solving the entire exercise as it is written

The difficulty level of this approach is well above that of the exam. So if you can manage this, you should be able to get a very good result on the exam.

7.2 Approach 2: Complete assembler-implementation, incomplete hardware

Since it is quite time-consuming to draw hardware-schematics, you can select an instruction set suitable for all functions, but in order to save time you might only have time to complete a few of the modules. (In that case you might want to concentrate on the modules that you are least familiar with.) The difficulty level of this approach is still above that of the exam though.

7.3 Approach 3: Incomplete assembler-implementation, incomplete hardware

If it is too difficult for you to consider all functions and all hardware-modules at the same time, you might want to limit yourself to a few functions and a few modules. The following is a list of combinations that I recommend in that case:

- ALU: function 1, function 4, function 6
- AGU: function 1, function 3, function 6
- MAC: function 2, function 3, function 7

- PC: function 3, function 5, function 8

The difficulty level of this approach is more or less similar to that of the harder questions given on the exam.

7.4 Approach 4: Same as approach 3 but with a simplified pipeline

In this approach you can simplify your assembly programs by assuming that the pipeline has full forwarding. And you might also want to make other assumptions (for example, you might want to remove the pipeline stage between the ALU/AGU and the memories and the MAC unit.

Depending on how many assumptions you make here, this exercise will be more or less easier than questions given on the exam. Even at this level it should be possible to pass the exam though.

7.5 The task

There are three tasks in this exercise. The first is to select a suitable instruction set. The second is to write assembly code for function 1-8 included in this document. Finally, you should draw the hardware schematic and control tables for the ALU, AGU, PC, and MAC unit of this processor. You should of course try to minimize the amount of hardware you use (especially multipliers and adders). For all of these tasks you need to consider the processor-pipeline given in Figure 1 on the next page.

This is meant to be a challenging task, but you are of course free to do only parts of it. For example, if you feel that your knowledge of MAC and ALU units are quite good, you can concentrate on the PC and AGU part of this exercise. Alternatively, you might want to simplify the exercise by only looking at (for example) program 1–4 instead of program 1–8.

Since this exercise is totally voluntary and will have no direct effect on your grade, you can (if you want to) do the exercise in a group of students. I also encourage you to team up with another student (or group of students) so that you can help correct each others' solution proposals.

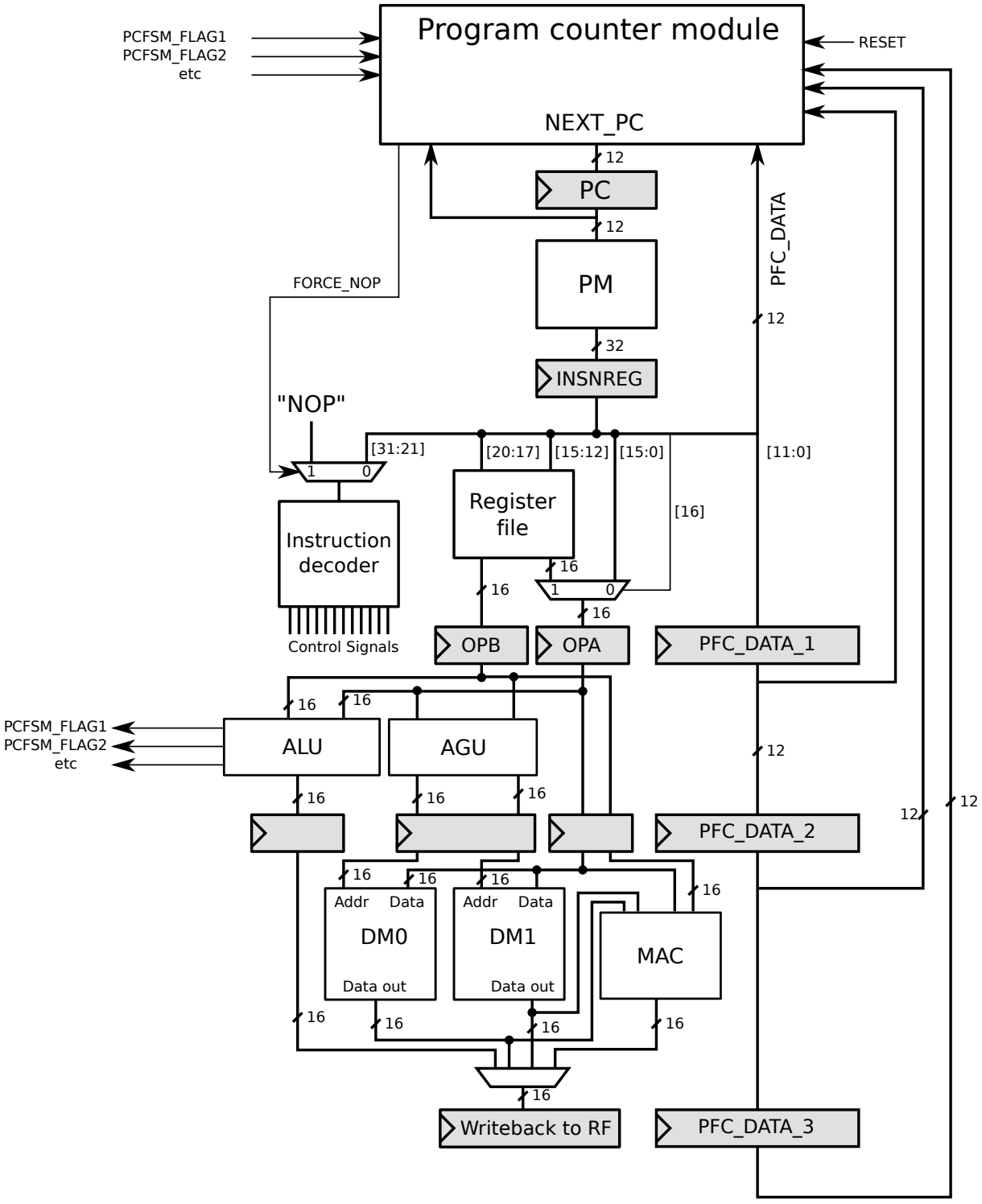


Figure 7.1: The processor pipeline.

7.6 Instruction format

- `insn[20:17]`: Selects OpB register
- `insn[15:12]`: Selects OpA register
- `insn[15:0]`: OpA immediate value
- `insn[16]`: Selects whether a register or immediate is used for OpA
- `insn[11:0]`: Immediate data sent to the PC module. Can be used in whatever way you want to.
- `insn[24:21]`: Selects the writeback register
- `insn[31:25]`: Instruction code (you can specify up to 2^7 instructions)

7.7 Constraints

- You may not change the pipeline, add ports to the register file or memorise, etc
- You need to be able to support at least **two levels of nested subroutine calls**
- Each program listed below also contains additional constraints for you

7.8 Hints

To limit the complexity of this exercise, the pipeline has no forwarding. You need to take this into account when you write your assembly programs. You should also remember that your branch instructions will (probably) have delay slots.

While it should be clear from the figure, do note that the register-file is implemented as a read-before write memory.

The hardware is also designed in such a way that you have a fairly large freedom in how you handle conditional branches. You can use whichever one of these two approaches that you prefer:

- ```
  cmp r0,r5
  bge label
  ```
- ```
  bge r0,r5,label
  ```

7.9 Our computation kernels

By using benchmarks the application engineers have determined that the following 8 functions are especially important for our application domain. They have also determined the clock cycle requirement on each of these functions.

Finally, to make it more interesting, the evil director of marketing have set arbitrary limits for the number of instructions that each function may occupy in the program memory. (Hint: Luckily there is still some space left in at least some of these functions for various program optimization techniques such as loop unrolling.)

7.9.1 Function 1: sanitize_buffer_values()

```
// inbuffer and outbuffer are both located in DM1
// buffersize is guaranteed to be less than 200
void sanitize_buffer_values(uint16_t buffersize)
{
   uint16_t i;
   for(i=0; i < buffersize*3; i++){
       int16_t x = inbuffer[i];
       if(x < -230){
           outbuffer[i] = -230 << 2;
       }else if (x > 5243){
           outbuffer[i] = 5243 << 2;
       }else{
           outbuffer[i] = x << 2;
       }
    }
}
```

Clock cycle constraints: Should execute in no more than 18 × buffersize + 10 clock cycles (including return from subroutine)
Program memory constraints: Must use less than 40 instruction words

7.9.2 Function 2: butterfly()

```
// i,j, and c are in general purpose registers
// indata, coeff, and result are located somewhere in DM0 and/or DM1
// (your choice)
void butterfly(uint16_t i,uint16_t j, uint16_t c)
{
    // Setup indata
    // Setup outdata
    int16_t tmp1 = indata[i];
    int16_t tmp2 = indata[i+1];
    int16_t tmp3 = indata[i+j];
    int16_t tmp4 = indata[i+j+1];

    int16_t w1 = coeff[c];
    int16_t w2 = coeff[c+1];

    // All multiplications and additions
    // in this function should use saturation!
    // (except address calculations)
    int16_t tmp5 = tmp3 * w1 - tmp4 * w2;
    int16_t tmp6 = tmp3 * w2 + tmp4 * w1;

    int16_t tmp7 = tmp1 + tmp5;
    int16_t tmp8 = tmp2 + tmp6;

    int16_t tmp9 = tmp1 - tmp5;
    int16_t tmp10 = tmp2 - tmp6;

    result[i] = tmp7;
    result[i+1] = tmp8;
    result[i+j] = tmp9;
    result[i+j+1] = tmp10;
}
```

Clock cycle constraints: Should execute in no more than 20 clock cycles (including return from subroutine)
Program memory constraints: No constraints have been given

7.9.3 Function 3: fir()

```
// Function parameters are stored in general purpose registers
int16_t fir(int16_t newsample, uint16_t coeff_ptr, uint16_t fir_ptr,
            uint16_t start_ptr, uint16_t end_ptr)
{
    DM0[fir_ptr] = newsample;
    fir_ptr = fir_ptr + 1;
    if(fir_ptr == end_ptr){
        fir_ptr = start_ptr;
    }

    ACR = 0
    for(i=0; i < 32; i = i + 1){
        // This is a fractional multiplication
        ACR = ACR + DM0[fir_ptr] * DM1[coeff_ptr];
        fir_ptr = fir_ptr + 1;
        if(fir_ptr == end_ptr){
            fir_ptr = start_ptr;
        }
        coeff_ptr = coeff_ptr + 1;
    }

    // Result should be in fractional format
    return SAT(ROUND(ACR)); // Returns the result in register r0
}
```

Clock cycle constraints: Should execute in no more than 45 clock cycles (including return from subroutine)
Program memory constraints: Must use no more than 15 instructions

7.9.4 Function 4: getbits()

```
// Function parameters should be in general purpose registers
// The return value should be in a general purpose register
uint16_t getbits(uint16_t val, uint16_t numbits, uint16_t offset)
{
    uint16_t tmp = (1 << numbits) - 1;
    val = val >> offset;
    return val & tmp;
}
```

Clock cycle constraints: Should execute in no more than 16 clock cycles (including return from subroutine)
Program memory constraints: No constraints given

7.9.5 Function 5: preprocess()

```
// Function parameters are stored in general purpose registers
uint16_t preprocess(int16_t a, int16_t b)
{
    if(a > b){
        if(a > 56){
            return a-56;
        }else{
            return b - 56;
        }
    }else{
        if(b < 12){
            return b+12;
        }else{
            return a+12;
        }
    }
}
```

Clock cycle constraints: No constraints given
Program memory constraints: No constraintes given

7.9.6 Function 6: prepare_data()

```
// It is up to you to select where indata is stored in DM0 and/or DM1.
void prepare_data(void)
{
    uint16_t i;
    for(i=0; i < 128; i++){
        int16_t tmp1,tmp2;
        tmp1 = indata[i*2+0];
        tmp2 = indata[i*2+1];
        int16_t average = ((int32_t)tmp1 + (int32_t) tmp2) >> 1;
        tmp = bitrev(i << 9);
        outdata[tmp] = average;
    }
}

// Bitrev is an operator that will reverse the order of all 16 input bits.
// In c-code it would look something like the following:
//
// uint16_t bitrev(uint16_t indata)
// {
//   int i;
//   int result = 0;
//   for(i=0; i < 16; i++){
//     if(indata & (1 << i)){
//       result = result | (1 << (15-i));
//     }
//   }
// }
//
// You are strongly encouraged to do something clever in regards to
// this...
```

Clock cycle constraints: Should execute in no more than 1900 clock cycles (including return from subroutine)
Program memory constraints: Must use less than 40 instruction words

7.9.7 Function 7: dot16()

```
// fixed_0_15 is a 16-bit fractional value (s.xxxxxxxxxxxxxxx)
// fixed_3_12 is a 16-bit fixed point value with 1 sign bit,
//            3 bits before the binary point, and 12 bits after the
//            binary point. (sxxx.xxxxxxxxxxxx)
//
// You'll have to decide the datatype for ACR yourself
//
// The result should be returned in register r0
// ptr1 and ptr2 are stored in general purpose registers.
fixed_3_12 dot16(uint16_t ptr1, uint16_t ptr2)
{
    fixed_0_15 tmp1, tmp2;
    fixed_3_12 result;

    uint16_t i;

    ACR = 0;
    for(i=0; i < 16; i++){
        tmp1 = DM0[ptr1++];
        tmp2 = DM1[ptr2++];
        ACR = ACR + tmp1 * tmp2;
    }

    result = SAT(ROUND(ACR));
    return result;
}
```

Clock cycle constraints: Should execute in no more than 30 clock cycles (including return from subroutine)
Program memory constraints: Must use less than 14 instruction words

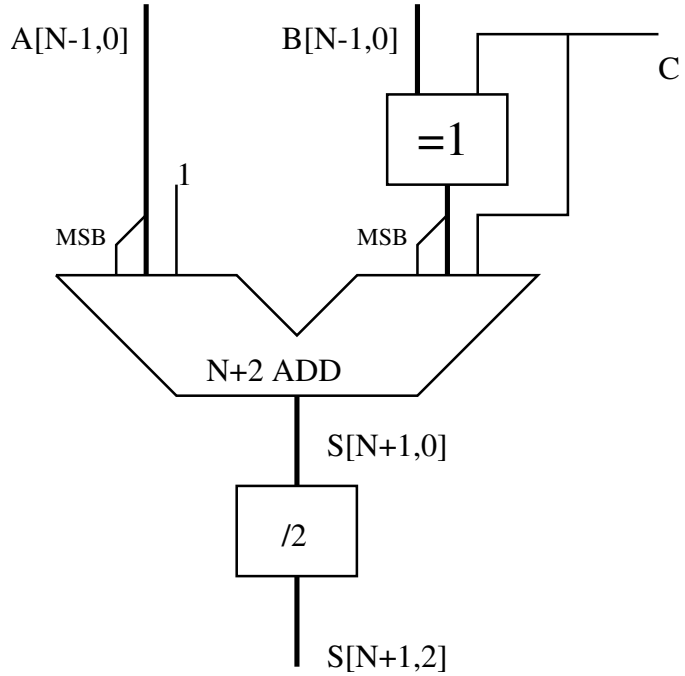
7.9.8 Function 8: find_val()

```
// Function parameters are stored in general purpose registers
// Return value should be in register r0
uint16_t find_val(int16_t val)
{
    const uint16_t NUMENTRIES = 256;
    uint16_t index = NUMENTRIES / 2;
    uint16_t step = NUMENTRIES / 4;

    while(step > 0){
        if(val == data[index]){
            return index;
        }else if(val < data[index]){
            index = index - step;
        }else{
            index = index + step;
        }
        step = step / 2;
    }
    return index;
}
```

Clock cycle constraints: No constraints given
Program memory constraints: No constraintes given

8 Solutions Proposals for the Introductory Exercises

Solution proposal for 1.1



Solution proposal for 1.2

The algorithm convolves a signal of 1000 samples with a filter with 32 coefficients.

```
.code
;; Initialize the address registers to copy coefficients from ROM0 to RA
;; This is done because data is stored in RAM0 and coefficients are stor
;; which does not allow simultaneous reads.
set     ar0,coeff_in_rom
set     step0,1
set     ar1,coeffs
set     step1,1
```

```
    repeat    coeff_copy,16
    ld0    r0,(ar0++)
    ld0    r1,(ar0++)
    st1    (ar1++),r0
    st1    (ar1++),r1
coeff_copy

    ; Set up pointers to coefficient and signal
    set     r3,signal          ; r3 points to the current sample
    set     ar1,coeffs         ; ar1 points to the copied coefficients
    set     ar0,signal         ; ar0 points to signals
    ; Set up a register to loop 1000 times
    set     r1,1000            ; loop counter
    ; Configure the wraparound for coefficients by setting bot and top
    set     bot1,coeffs
    set     top1,coeffs_end
loop
    ; Move on to next sample
    inc     r3
    move    ar0,r3
    ; Do the convolution using a repeat instruction
    repeat  convolution,32
    convss  acr0,(--ar0),(ar1++%)
convolution
    ; Jump back to loop if we haven't done 1000 samples
    dec     r1
    jump.ne ds3 loop
    ; Output the result
    move    r31,rnd div2 acr0  ; Read output from accumulator with rounding
    clr     acr0               ; clear accumulator
    out     0x11,r31           ; output sample to file

    ; End of program
    out     0x13,r0


; Data definition

    .ram0
    .skip 31   ; Leading zeros
signal
    .skip 1000 ; The signal will be stored here
    .skip 31   ; Trailing zeros

;; We load the coefficients from a file which can
;; be generated in for example Matlab
    .rom0
    .scale 0.125
coeff_in_rom
; #include "coeffs.inc"
```

```
;; Space in DM1 for the coefficients
    .ram1
coeffs
    .skip 31  ; allocates 31 memory words for coeffs
coeffs_end    ; should point to last position
    .skip 1   ; last memory word
```

More efficient version which stores samples in a ring buffer.

```
    .code
coeff_copy
    ; Set up a loop counter for the outer loop
    set r1,1000          ; Loop counter

    ; Set up pointers for the ringbuffer and coefficients
    set ar0,coeffs       ; Pointer to the coefficients
    set ar1,ringbuffer   ; Pointer to the next location in the ringbuffer
    set ar2,signal       ; Pointer to the next sample

    ; Set up auto wrap-around
    set bot0,coeffs
    set top0,coeffs_end
    set bot1,ringbuffer
    set top1,ringbuffer_end

loop
    ; Load a new sample from signal to a temporary register
    ld0 r0,(ar2++)                    ; Load the next sample
    nop
    ; Store the sample in the ring buffer
    st0 (ar1),r0                      ; Store the next sample in the ring bu
    repeat  convolution,31
    ; Do the bulk of the convolution
    convss  acr0,(ar0++%),(ar1++%)    ; Do the bulk of the convolution
convolution
    ; Do the final step of the convolution, making sure to restore the ring
    ; buffer to the correct place
    move r2,ar1                       ; Store the second to last position of
    convss  acr0,(ar0++%),(ar1++%)    ; Do the final MAC convolution operati
    move ar1,r2                       ; Restore the ring buffer location to

    ; Jump back in the outer loop if desired
    jump.ne ds3 loop

    ; Output result and end progrma
    move    r31,rnd div2 acr0  ; Read output from accumulator with rounding
    clr     acr0               ; clear accumulator
    out     0x11,r31           ; output sample to file

    ; End of program
    out     0x13,r0
```

```
; Data definition

    .ram0
    .skip 31    ; Leading zeros
signal
    .skip 1000 ; The signal will be stored here
    .skip 31    ; Trailing zeros

;; We load the coefficients from a file which can
;; be generated in for example Matlab
    .rom0
    .scale 0.125
coeffs
#include "coeffs.inc"
coeffs_end

;; Space in DM1 for the ring buffer
.ram1
ringbuffer
    .skip 31     ; allocates 31 memory words
ringbuffer_end  ; should point to the last position
    .skip 1      ; allocates last memory word
```

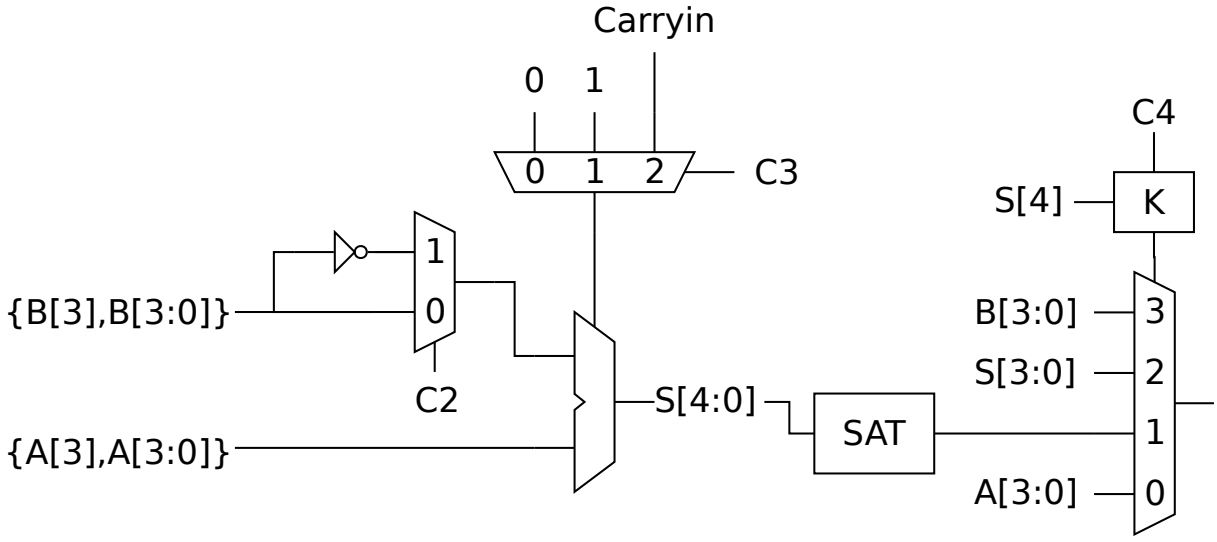
Solution proposal for 1.3

```
    .code
init_fir_kernel
    set r1,0    ; r0-r4 are used as the ringbuffer
    set r2,0    ; in which we store the samples.
    set r3,0    ; (No need to clear r0, as it is written
    set r4,0    ; immediately below.)

    ; Our coefficients are stored in here
    set r5,COEFF1
    set r6,COEFF2
    set r7,COEFF3
    set r8,COEFF4
    set r9,COEFF5

    set r11,200 ; Loop counter. (Divided by 5 since
                ; we have unrolled the loop 5 times.)

fir_kernel_unrolled
    in    r0,0x10
    clr   acr0
    macss acr0,r0,r5
    macss acr0,r1,r6
    macss acr0,r2,r7
```

```
macss acr0,r3,r8
macss acr0,r4,r9
move  r10,sat rnd mul2 acr0    ; Scale output assuming
in    r4,0x10                  ; inputs and outputs are
clr   acr0                     ; in fractional format
out   0x11,r10

macss acr0,r4,r5               ; Repeat everything but this
macss acr0,r0,r6               ; time, r4 contains the latest
macss acr0,r1,r7               ; sample.
macss acr0,r2,r8
macss acr0,r3,r9
move  r10,sat rnd mul2 acr0
in    r3,0x10
clr   acr0
out   0x11,r10

macss acr0,r3,r5               ; And so on (r3 contains latest
macss acr0,r4,r6               ; sample)
macss acr0,r0,r7
macss acr0,r1,r8
macss acr0,r2,r9
move  r10,sat rnd mul2 acr0
in    r2,0x10
clr   acr0
out   0x11,r10

macss acr0,r2,r5
macss acr0,r3,r6
macss acr0,r4,r7
macss acr0,r0,r8
macss acr0,r1,r9
move  r10,sat rnd mul2 acr0
in    r1,0x10
clr   acr0
out   0x11,r10

macss acr0,r1,r5
macss acr0,r2,r6
macss acr0,r3,r7
macss acr0,r4,r8
macss acr0,r0,r9
move  r10,sat rnd mul2 acr0

add   r11,-1
jump.ne ds1 fir_kernel_unrolled
out   0x11,r10

out 0x13,r0                    ; And we are done!
```

9 ALU Solution Proposals

Solution proposal for 2.1



The control table

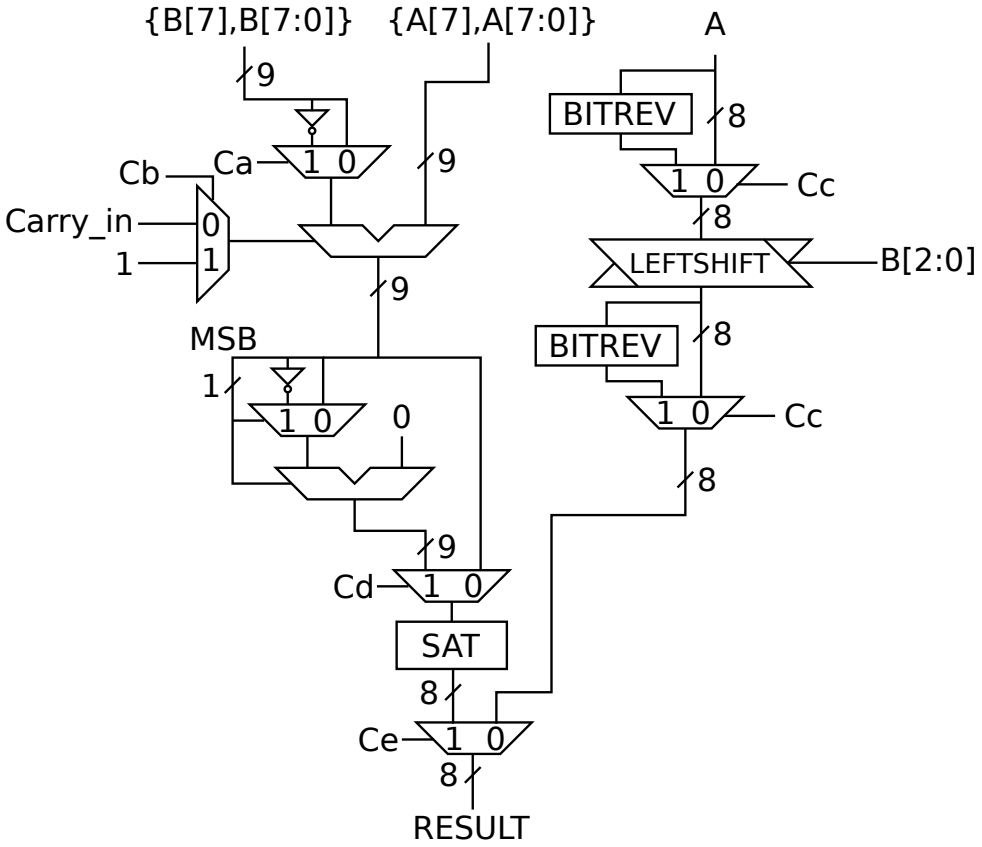
Operation	C2	C3	C4
A + B + Cin	0	2	2
SAT(A + B)	0	0	1
SAT(A – B)	1	1	1
MAX(A + B)	1	1	0

The K logic:

C4	S[4]	Out
0	0	0
0	1	3
1	-	1
2	-	2

```
always @* begin : SAT
    sat_result = S[3:0];
    if(S[4] != S[3]) begin
        if(S[4]) begin
            sat_result[3:0] = 4'b1000;
        end else begin
            sat_result[3:0] = 4'b0111;
        end
    end
end
```

Solution proposal for 2.2



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];
```

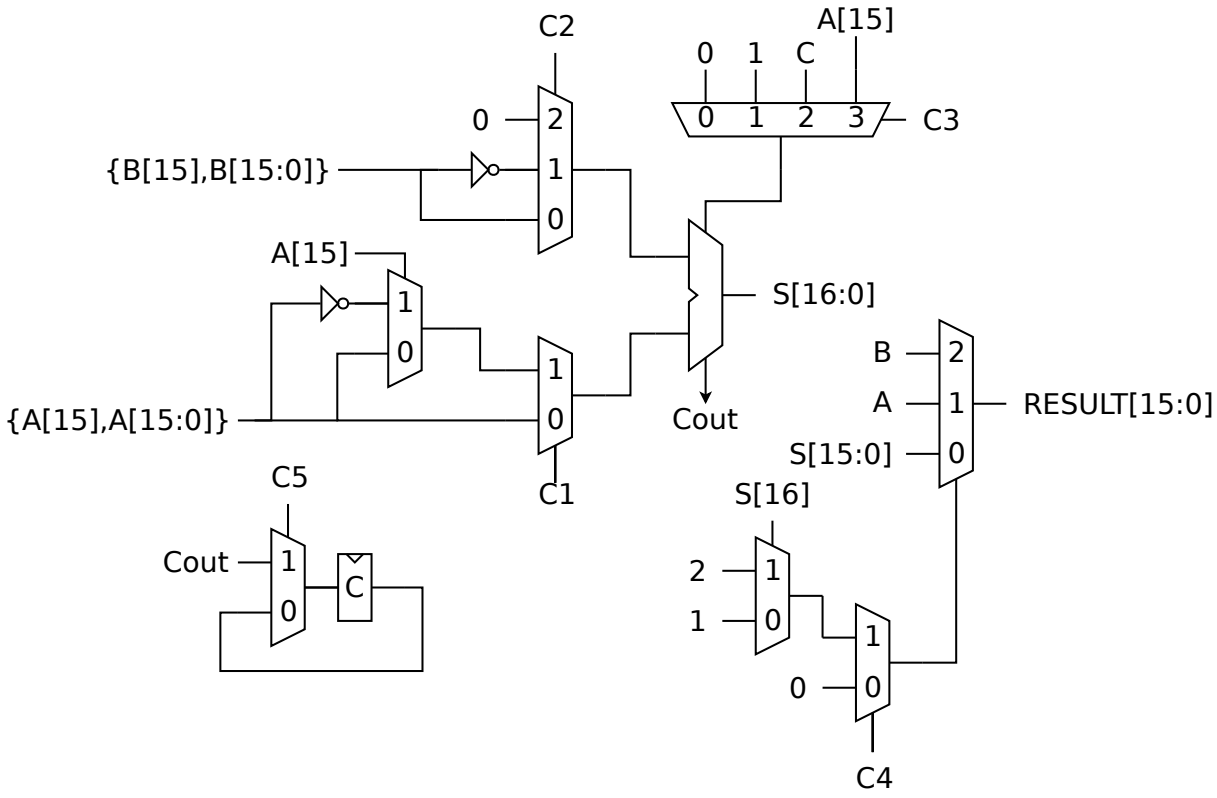
Comments:
Note that this solution utilizes the fact that it is cheaper to use one shifter combined with two bitreverse-operations compared to using two shifters. It also uses the fact that $|A - B| = |B - A|$.

Solution proposal for 2.3

Proposed ALU instructions: `ABS(A)`, `MAX(A,B)`, `A+B`, `A-B`, and `A+B+Carry`

```
update_statistics:
    // This function would be very easy if packet_ctr and
    // length were unsigned. However, as they are signed this is
    // more tricky. If this was more performance critical we can
    // add more instructions to handle this, but as we are lazy
    // hardware designers we want to avoid cluttering up the
    // instruction set if we can avoid it:

    set       ar0,packet_ctr
    set       r4,0
    add       r1,r0,#0x8000      ; carry = (length < 0)
    addc      r4,r4,r4           ; 1 in r4 if length < 0
    ld        r1,DM0[ar0]
    sub       r4,#0,r4           ; -1 in r4 if negative
    add       r1,r0
```

```
    st      DM0[ar0++],r1
    repeat  endloop,3
    ld      r1,DM0[ar0++]
    nop                         ; Data dependency for load!

    addc    r1,r4
    st      DM0[ar0++],r1
endloop:
    ret

find_maxabsval:
    set     ar0, dct_indata
    set     r0,0        ; r0 contains max value encountered

    repeat loop, 16
    ld      r1,DM0[ar0++]
    ld      r3,DM0[ar0++]
    abs     r2,r1
    max     r0,r2,r0
    abs     r4,r3
    max     r0,r4,r0
loop:
    ret
```



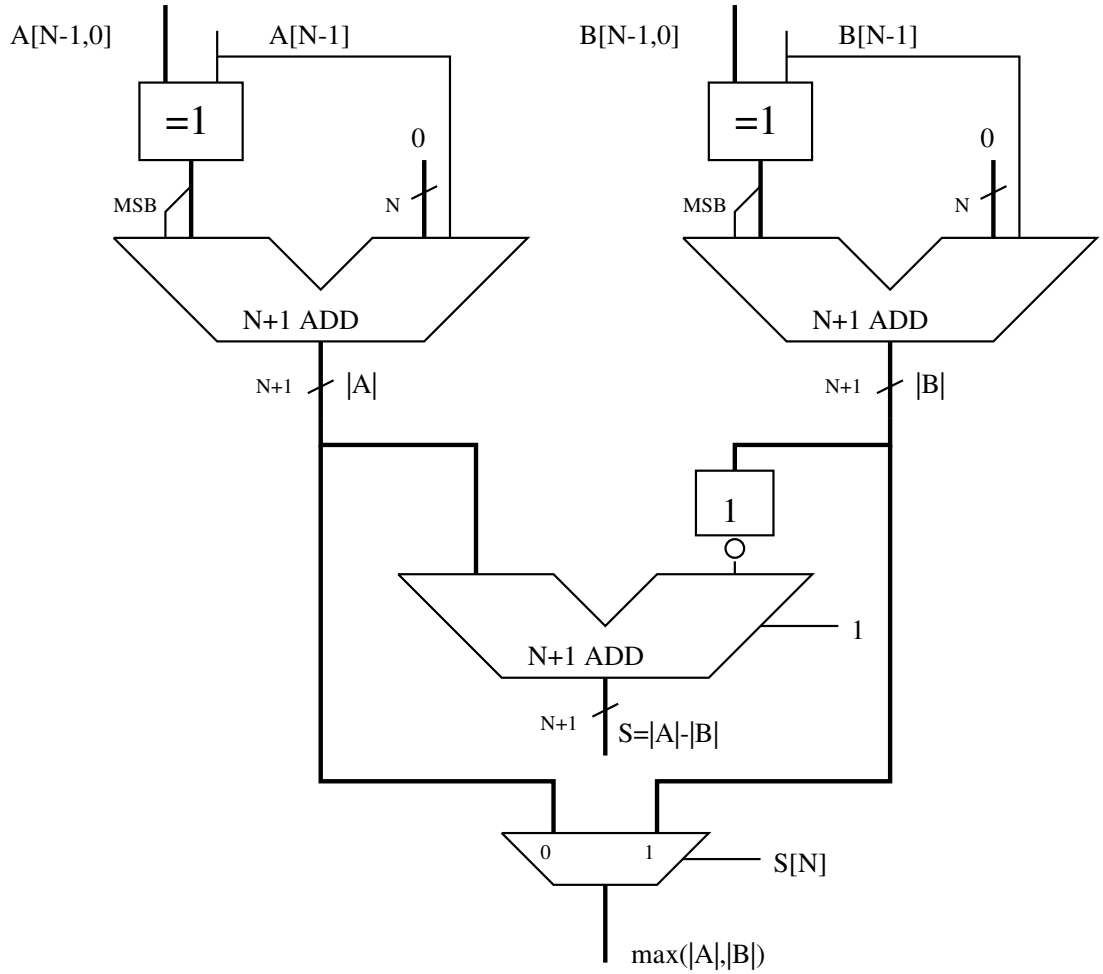
Control table:

Operation	C1	C2	C3	C4	C5
ABS(A)	1	2	3	0	0
MAX(A, B)	0	1	1	1	0
A + B	0	0	0	0	1
A – B	0	1	1	0	1
A + B + Carry	0	0	2	0	1

Solution proposal for 2.4

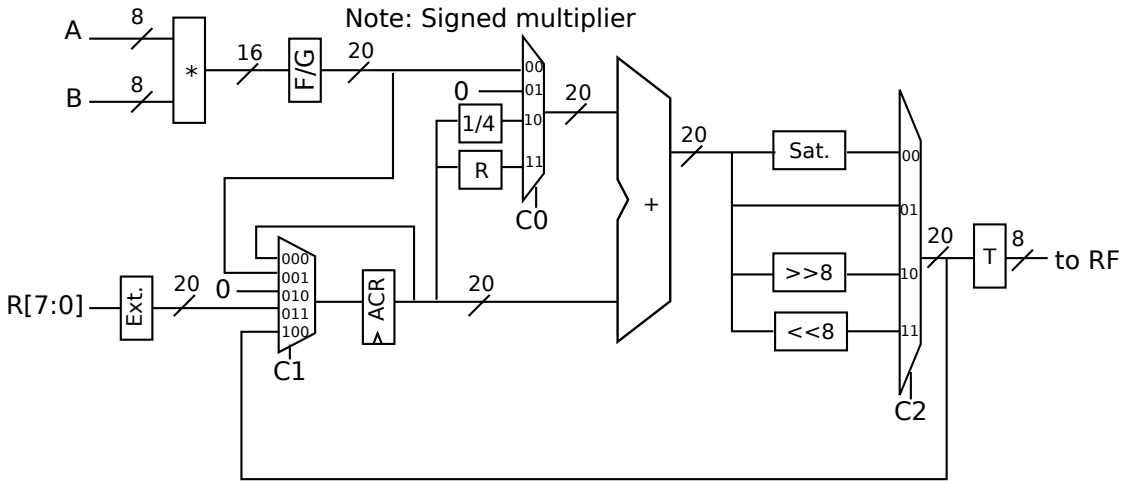
Proposed ALU instructions: `MAXABS(A,B)`. Constraint: After a load instruction we must wait a clock cycle before we can use the result. Clock cycles $= 2 \cdot 31 + 5 = 67$.

```
        set     ar0,dct_indata
        set     r0,0                ; biggest
        ld      r1,(ar0++)          ; prologue
        repeat  loop,31
        ld      r1,(ar0++)
        maxabs  r0,r1,r0            ; loop
loop
        maxabs  r0,r1,r0            ; epilogue
        ret
```


A[N-1,0]
A[N-1]
=1
0
MSB
N
N+1 ADD
N+1
|A|
B[N-1,0]
B[N-1]
=1
0
MSB
N
N+1 ADD
N+1
|B|
1
N+1 ADD
1
N+1
S=|A|-|B|
0
1
S[N]
max(|A|,|B|)

10 MAC Unit Solution Proposals

Solution proposal for 3.1




```
Ext.:  output[19:0] = { {4{R[7]}}, R[7:0], 8'd0};
F/G:   output[19:0] = { {3{input[15]}}, input[15:0], 1'b0};
1/4:   output[19:0] = { {2{input[19]}}, input[19:2]};
R:     output[19:0] = {11'd0, input[7], 8'd0};
T:      output[7:0] = input[15:8];
>>8: output[19:0] = { {8{input[19]},input[19:8]};
<<8: output[19:0] = { input[11:0], 8'b0};
```

```
Sat:
   if(in[19:15]=={5{in[15]}})
      out = in;
   else if(in[19] == 1'b1)
      out = 20'hf8000;
   else
      out = 20'h07fff;
```

Operation	C0	C1	C2
OP0: No operation	-	000	-
OP1: `ACR = 0`	-	010	-
OP2: `ACR = A * B`	-	001	-
OP3: `ACR = A * B + ACR`	00	100	01
OP4: `ACR = 1.25 * ACR`	10	100	01
OP5: Load `ACR`	-	011	-
OP6: `ACR = SATURATE(ROUND(ACR))`	11	100	00
OP7: `RF = ACR[7:0]`	01	000	11
OP8: `RF = ACR[15:8]`	01	000	01
OP9: `RF = SIGNEXTEND(ACR[19:16])`	01	000	10

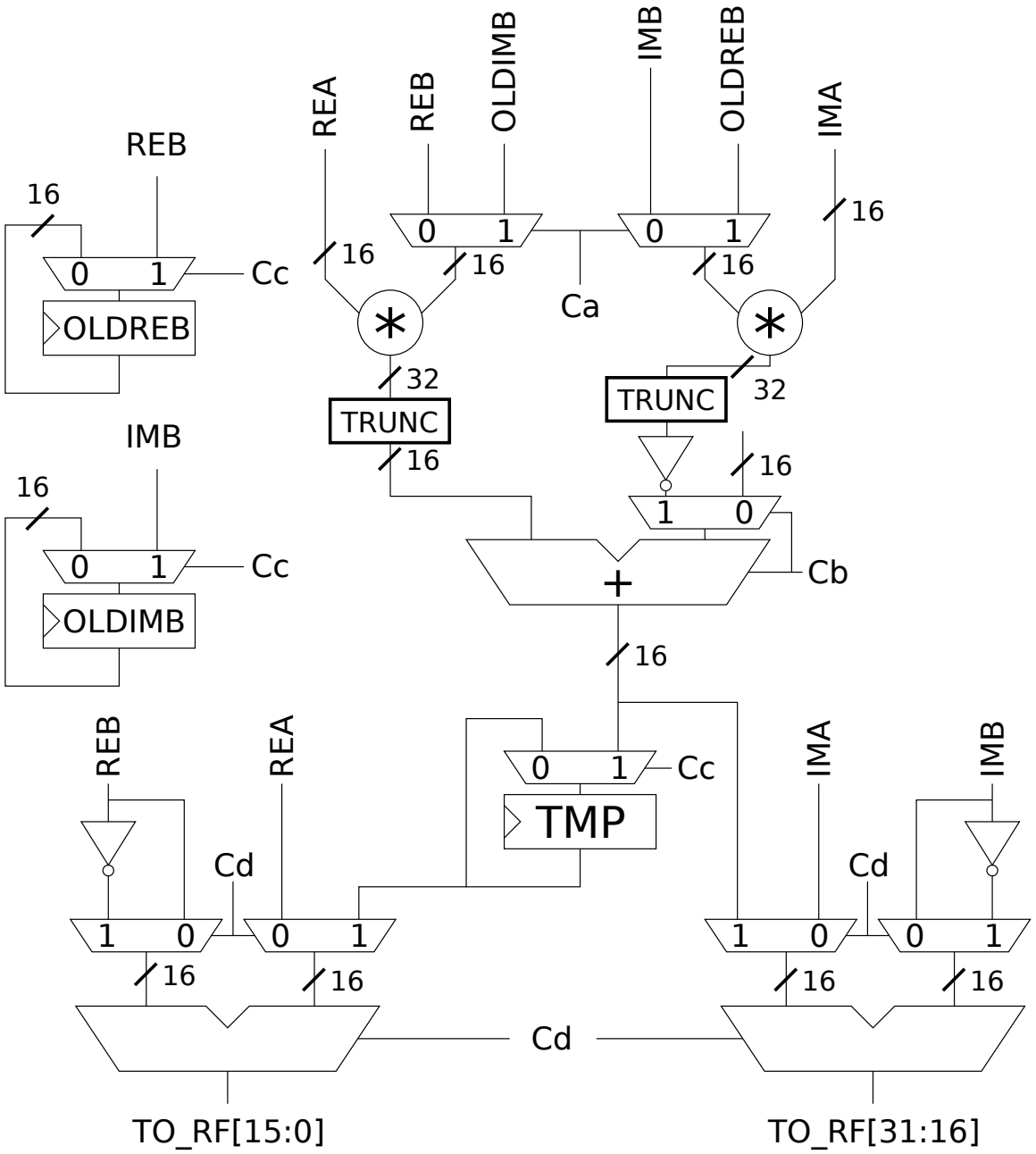
Solution proposal for 3.2

Implementing a complex multiplier in a straight forward fashion will use four multipliers. This is very wasteful as we will only use them two times the 7 clock cycles we are allowed to use.

Luckily, it is quite easy to solve this exercise using only two multipliers by combining the complex multiply and complex subtract into two instructions where the first instruction performs two multiplications and the second instruction performs another two instructions:

```
// Instruction set
csubmul1 : TMP = REA * REB - IMA * IMB;
           OLDREB = REB;
           OLDIMB = IMB;
csubmul2 : TO_RF[31:16] = REA * OLDIMB + OLDREB * IMA - IMB;
           TO_RF[15:0]  = TMP - REB;
cadd     : TO_RF[31:16] = IMA + IMB;
           TO_RF[15:0] = REA + REB;

// Assembler listing

csubmul1      r0, r8 // No destination reg
csubmul2 r12, r0, r1
cadd     r13, r0, r1

csubmul1      r2, r9 // No destination reg
csubmul2 r14, r2, r3
cadd     r15, r2, r3
```

REA = OpA[15:0]; IMA = OpA[31:16]
REB = OpB[15:0]; IMB = OpB[31:16]



trunc: out[15:0]=in[30:15];

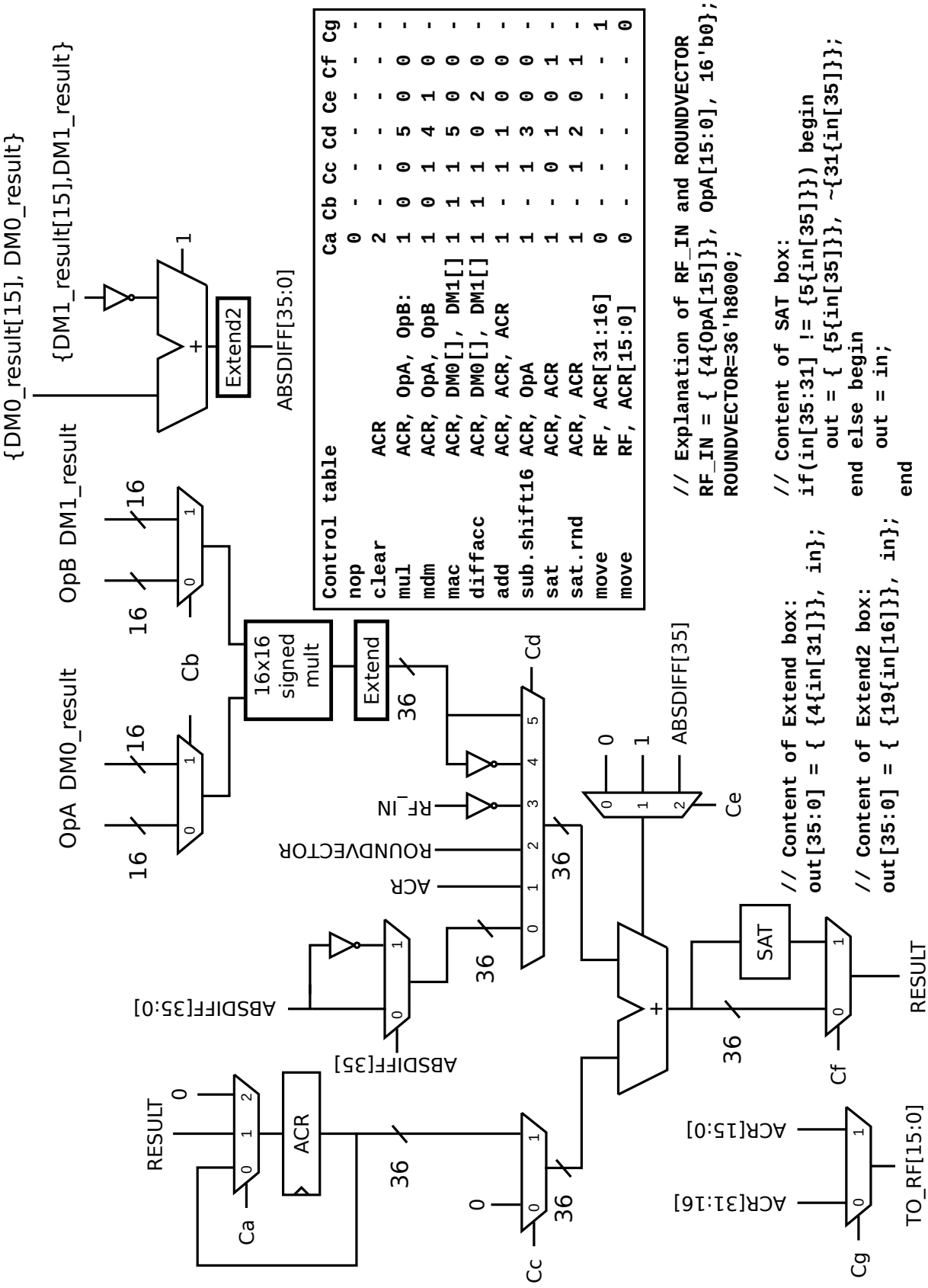
Control table	Ca	Cb	Cc	Cd
csubmul1	0	1	1	-
csubmul2	1	0	0	1
cadd	-	-	0	0

Solution proposal for 3.3

For the first function we use a small trick, we do not have a separate multiplication instruction depending on whether we use fractional or integer multiplication. Instead we scale the result in program 1 by two by adding the ACR to itself after the multiplications are finished. This is OK as the exercise only specifies that the result in r6/r7 should be in fractional, our intermediate calculations can be done in whatever way we want to. Note: For the `sat.rnd` instruction, ACR[15:0] will become undefined due to the solution used in the schematic. However, this doesn't matter in this case as only ACR[31:16] is accessed after this.

In program 2 we perform another trick. By reordering the instructions slightly we need only one accumulator register without violating the performance constraints.

```
// function butterfly_part()
mul ACR, r0, r2      // ACR = r0 * r2
mdm ACR, r1, r3      // ACR -= r1 * r3
add ACR, ACR         // ACR = ACR << 1 (Shift up one step to
                     // compensate for fractional)
sub.shift16 ACR, r4 // ACR = ACR - r4 << 16
sat.rnd ACR, ACR     // ACR = SAT(ROUND(ACR))
move r6, ACR[31:16] // r6 = ACR[31:16]
mul ACR, r0, r3
mac ACR, r1, r2      // ACR += r1 * r3
add ACR, ACR
sub.shift16 ACR, r5
sat.rnd ACR, ACR
move r7, ACR[31:16]
ret
// function filter()
clear ACR            // ACR = 0
set AR0, r0
set AR1, r1
repeat 30                        // Only repeats next instruction
mac ACR,DM0[AR0++],DM1[AR1++] // ACR += DM0[AR0++] * DM1[AR1++]
sat ACR,ACR
move r2,ACR[31:16]
move r3,ACR[15:0]
clear ACR
set AR0,r0
set AR1,r1
repeat 30                        // Only repeats next instruction
diffacc ACR0,DM0[AR0++],DM1[AR1++] // ACR = ACR +
                                   // ABS(DM0[AR0++]-DM1[AR1++])
move r4,ACR[31:16]
move r5,ACR[15:0]
ret
```



Control table	Ca	Cb	Cc	Cd	Ce	Cf	Cg
nop	0	-	-	-	-	-	-
clear	2	-	-	-	-	-	-
mul ACR, OpA, OpB:	1	0	0	5	0	0	-
mdm ACR, OpA, OpB	1	0	1	4	1	0	-
mac ACR, DM0[], DM1[]	1	1	1	5	0	0	-
diffacc ACR, DM0[], DM1[]	1	1	1	0	2	0	-
add ACR, ACR, ACR	1	-	1	1	0	0	-
sub.shift16 ACR, OpA	1	-	1	3	0	0	-
sat ACR, ACR	1	-	0	1	0	1	-
sat.rnd ACR, ACR	1	-	1	2	0	1	-
move RF, ACR[31:16]	0	-	-	-	-	-	1
move RF, ACR[15:0]	0	-	-	-	-	-	0

```
// Explanation of RF_IN and ROUNDVECTOR
RF_IN = { {4{OpA[15]}}, OpA[15:0], 16'b0};'
ROUNDVECTOR=36'h8000;

// Content of Extend box:
out[35:0] = { {4{in[31]}}, in};

// Content of Extend2 box:
out[35:0] = { {19{in[16]}}, in};

// Content of SAT box:
if(in[35:31] != {5{in[35]}}) begin
    out = { {5{in[35]}}, ~{31{in[35]}}};
end else begin
    out = in;
end
```

Solution proposal for 3.4

This exercise is quite straight forward to solve: Just create a complex multiply, complex add, and complex subtract instruction.

```
// Assuming the following aliases for OpA/OpB:
REA = OpA[15:0] ; IMA = OpA[31:16]
REB = OpB[15:0] ; IMB = OpB[31:16]

// Instruction set:
cmul: TO_RF[15:0]  = REA*REB - IMA*IMB;
      TO_RF[31:16] = REA*IMB + REB*IMA;
csub: TO_RF[15:0]  = REA-REB;
      TO_RF[31:16] = IMA-IMB;
cadd: TO_RF[15:0]  = REA+REB;
      TO_RF[31:16] = IMA+IMB;

// Assembler listing:
// Assume i0 is in r0, c0 in r8, i1 in r1
//        i2 is in r2, c1 in r9, i3 in rr
//        o0 in r12, o1 in r13, o2 in r14
//        o3 in r15

cmul r12, r0, r8
csub r12, r12, r1
cadd r13, r0, r1

cmul r14, r2, r9
csub r14, r14, r3
cadd r15, r2, r3
```

The schematic for this solution is left as an exercise for the reader...

Solution proposal for 3.5

Gauss' complex multiplication algorithm can be explained as follows:
$(a+bi)(c+di) = (k_1-k_3)+(k_1+k_2)i$, where $k_1 = c(a+b)$, $k_2 = a(d-c)$, and $k_3 = b(c+d)$.
We should now design an instruction set which can perform the following operations in 7 clock cycles (where all variables contain complex valued data):

```
o0 = e0 * f0 - e1
o1 = e0 + e1
o2 = e2 * f1 - e3
o3 = e2 + e3
```

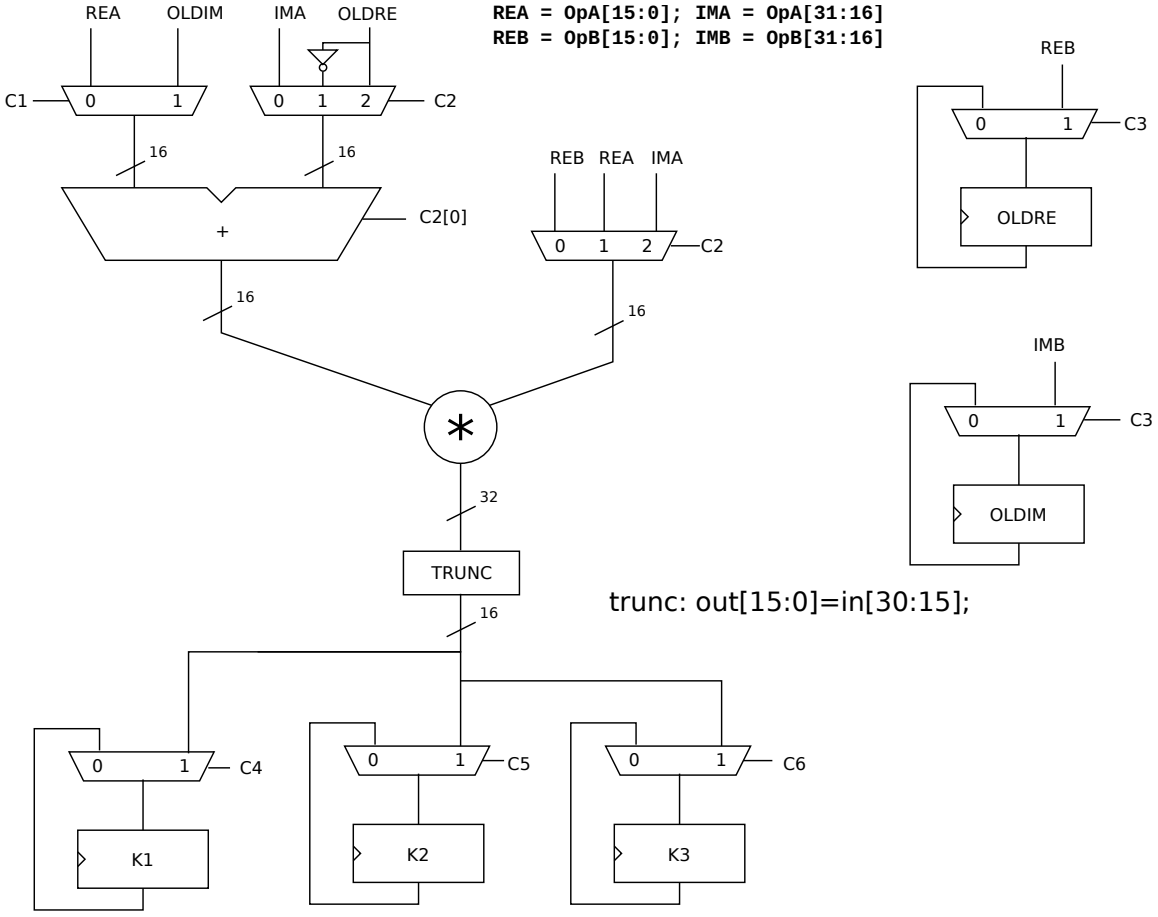
We create the following instruction set. k1, k2, k3, OLDRE, and OLDIM are registers inside our MAC unit. REA is the real part of operand A, REB is the real part of operand

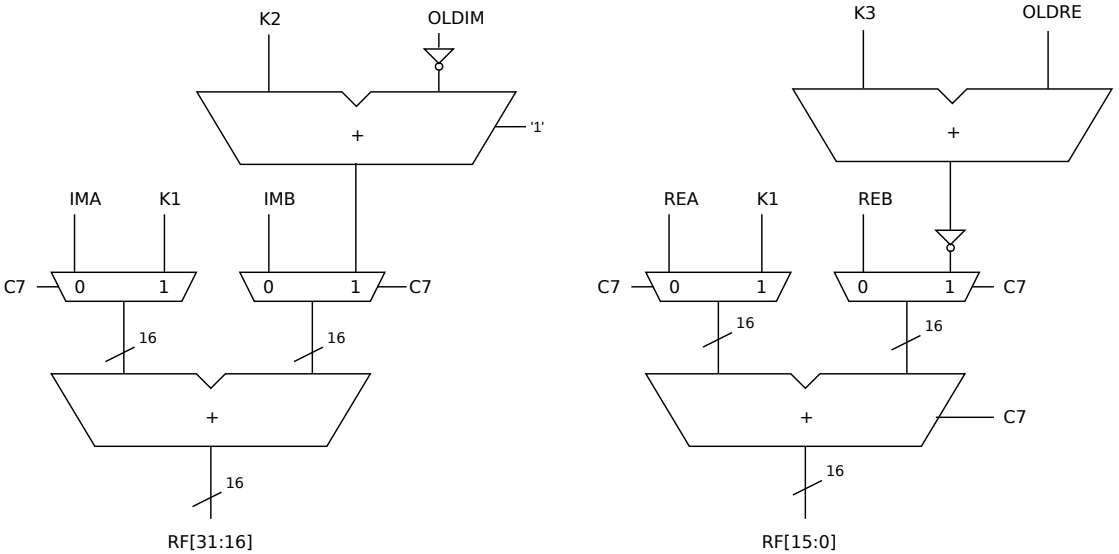
B, IMA is the imaginary part of operand A, and IMB is the imaginary part of operand B. OpD signifies the destination register.

```
acc0 OpA,OpB      ; k1 = REB * (REA + IMA)
                  ; OLDRE = REB, OLDIM = IMB
acc1 OpD,OpA,OpB; k2 = REA * (OLDIM - OLDRE)
                  ; TO_RF[31:16] = IMA + IMB
                  ; TO_RF[15:0] = REA + REB
acc2 OpA,OpB      ; k3 = IMA * (OLDRE + OLDIM)
                  ; OLDRE = REB, OLDIM = IMB
acc3 OpD,OpA,OpB; TO_RF[31:16] = k1 + k2 - OLDIM
                  ; TO_RF[15:0]  = k1 - k3 - OLDRE
                  ; k1 = REB*(REA+IMA)
                  ; OLDRE = REB, OLDIM = IMB
acc4 OpD          ; TO_RF[31:16] = k1 + k2 - OLDIM
                  ; TO_RF[15:0] = k1 - k3 - OLDRE
```

With these instructions we can write the following program:

```
                  ; Assumption: e0 is in r0, f0 is in r1,
                  ;             e1 is in r2
                  ;             e2 is in r3, f1 is in r4,
                  ;             e3 is in r5
acc0 r0,r1        ; k1 = REB * (REA + IMA)
                  ; OLDRE = REB, OLDIM = IMB
acc1 r10,r0,r2    ; k2 = REA * (OLDIM - OLDRE)
                  ; TO_RF[31:16] = IMA + IMB        ; o1 in r10
                  ; TO_RF[15:0] = REA + REB
acc2 r0,r2        ; k3 = IMA * (OLDRE + OLDIM)
                  ; OLDRE = REB, OLDIM = IMB
acc3 r11,r3,r4    ; TO_RF[31:16] = k1 + k2 - OLDIM ; o0 in r11
                  ; TO_RF[15:0]  = k1 - k3 - OLDRE
                  ; k1 = REB*(REA+IMA)
                  ; OLDRE = REB, OLDIM = IMB
acc1 r12,r3,r5    ; k2 = REA * (OLDIM - OLDRE)
                  ; TO_RF[31:16] = IMA + IMB        ; o3 in r12
                  ; TO_RF[15:0] = REA + REB
acc2 r3,r5        ; k3 = IMA * (OLDRE + OLDIM)
                  ; OLDRE = REB, OLDIM = IMB
acc4 r13          ; TO_RF[31:16] = k1 + k2 - OLDIM ; o2 in r13
                  ; TO_RF[15:0] = k1 - k3 - OLDRE
```


REA = OpA[15:0]; IMA = OpA[31:16]
REB = OpB[15:0]; IMB = OpB[31:16]
REA
OLDIM
IMA
OLDRE
C1
C2
C2[0]
REB
C3
IMB
TRUNC
trunc: out[15:0]=in[30:15];
C4
C5
C6
K1
K2
K3
16
32


K2
OLDIM
K3
OLDRE
'1'
IMA
K1
IMB
REA
REB
C7
RF[31:16]
RF[15:0]
16

Instruction	C1	C2	C3	C4	C5	C6	C7
acc0	0	0	1	1	0	0	-
acc1	1	1	0	0	1	0	0
acc2	1	2	1	0	0	1	-
acc3	0	0	1	1	0	0	1
acc4	-	-	0	0	0	0	1

Note: It is actually possible to do away with the acc0 and acc4 instruction by rewriting the program as follows:

```
acc3 r11,r0,r1 ; Write unknown data to r11
acc1 r10,r0,r2
acc2 r0,r2
acc3 r11,r3,r4 ; Write correct data to r11
acc1 r12,r3,r5
acc2 r3,r5
acc3 r13,r3,r4 ; Write dummy data to k1, OLDRE and OLDIM
```

Solution proposal for 3.6

The easiest way to solve this task when under time constraint is probably to create a MAC unit which contains four specialized 16-bit registers, m0-m3 and then having the biquad function look approximately as follows:

```
biquad:
     move ar0, r0
     move ar1, r1
     ; move DM0[r2+0] - DM0[r2+3] to the m0-m3 registers

     repeat 100, endloop

     mul acr, #coeff1, m0
     mac acr, #coeff2, m1
     mac acr, #coeff3, m2
     mac acr, #coeff4, m3
     mac.shiftm0m1 acr, #coeff0, DM0[ar0++]
     ; The mac.shift instruction moves m0 into m1 and loads a new
     ; value into m0 from operand b (e.g. DM0).
     sat.rnd.shift acr
endloop:
     Save m0 to m3 to the state array.
```

biquad translated to assembler using software tricks

However, by rewriting the computation kernel somewhat it can be seen that there is no particular need for special m0..m3 registers as these values can be read either directly from a register or directly from memory.

```
biquad:
    ld r10,DM0[r2+0] ; m0
    ld r11,DM0[r2+1] ; m1
    ld r20,DM0[r2+2] ; m2
    ld r21,DM0[r2+3] ; m3

    move ar0, r0 ; ar0 points to indata
    move ar1, r1 ; ar1 points to outdata

prologue:
    mul acr, #coeff0, DM0[ar0++]
    mac acr, #coeff1, r10          ; m0
    mac acr, #coeff2, r11          ; m1
    mac acr, #coeff3, r20          ; m2
    mac acr, #coeff4, r21          ; m3
    move.sat.rnd r21, acr
    move DM0[ar1++], r21

    ; Now things are shifted around as follows:
    ; new m0 = DM0[ar0-1]
    ; new m1 = old m0
    ; new m2 = saturated accumulator
    ; new m3 = old m2
    mul acr, #coeff0, DM0[ar0++]
    mac acr, #coeff1, DM0[ar0-2]   ; m0
    mac acr, #coeff2, r10          ; m1
    mac acr, #coeff3, r21          ; m2
    mac acr, #coeff4, r20          ; m3
    move.sat.rnd r20, acr
    move DM0[ar1++], r20

    ; Now things are shifted around as follows:
    ; new m0 = DM0[ar0-1]
    ; new m1 = DM0[ar0-2]
    ; new m2 = saturated accumulator
    ; new m3 = old m2

    repeat 49, endloop
loop:
    mul acr, #coeff0, DM0[ar0++]
    mac acr, #coeff1, DM0[ar0-2]   ; m0
    mac acr, #coeff2, DM0[ar0-3]   ; m1
    mac acr, #coeff3, r20          ; m2
    mac acr, #coeff4, r21          ; m3
    move.sat.rnd r21, acr
    move DM0[ar1++], r21

    mul acr, #coeff0, DM0[ar0++]
    mac acr, #coeff1, DM0[ar0-2]   ; m0
    mac acr, #coeff2, DM0[ar0-3]   ; m1
    mac acr, #coeff3, r21          ; m2
```

```
    mac acr, #coeff4, r20          ; m3
    move.sat.rnd r20, acr
    move DM0[ar1++], r20
endloop:

    ld r10, DM0[ar0]
    ld r11, DM0[ar0-1]
    ld r12, DM0[ar1-1]
    ld r13, DM0[ar1-2]
    st DM0[r2+0],r10
    st DM0[r2+1],r11
    st DM0[r2+2],r12
    st DM0[r2+3],r13
    ret
```

Operation list proposal:

In other words, by rearranging the order of operations and move some of the data into special purpose registers, a MAC unit with a very limited set of operationsis available:

- `MUL acr, op1, op2`
- `MAC acr, op1, op2`
- `mat.sat.rnd ACR, world`

However, if you are optimizing for power, you might want a hybrid solution, which does not require the kernel to read an identical value from memory several times a day.

Schematic

Left as an exercise for the reader.

11 PFC Unit Solution Proposals

Solution proposal for 4.1

a) 1 delay slot for unconditional branches

b)

```
// One nop instruction is needed in this pipeline between
// arithmetic operations.
add r3,r0,r1
nop
add r3,r2,r3
```

c–d) Let us create the following jump instructions with three delay slots each.

- `jump.eq OpA, OpB, targetaddr`
- `jump.neq OpA, OpB, targetaddr`
- `jump.lt OpA, OpB, targetaddr`

Program 1

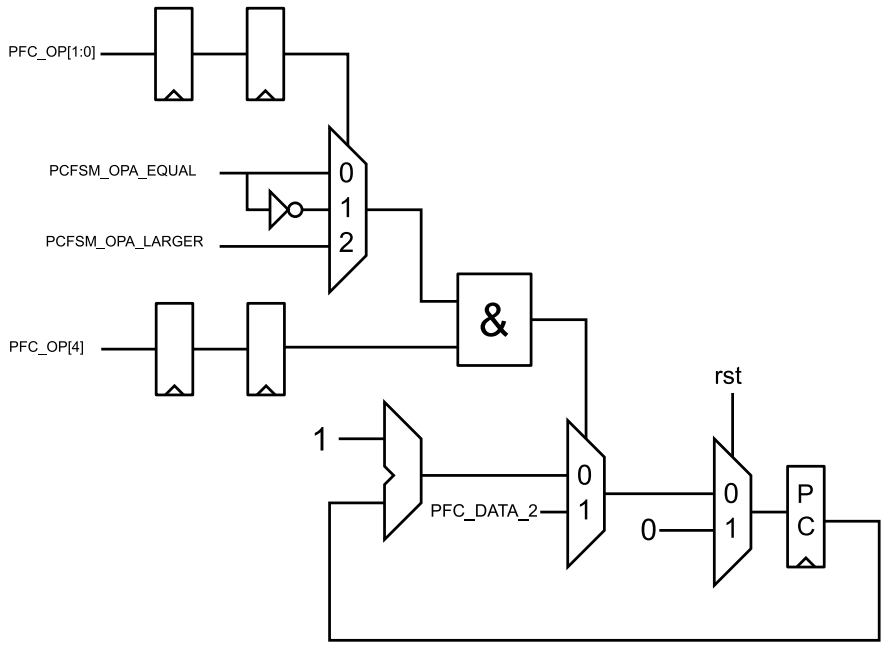
```
        set r2,10
        nop

        jump.lt r0,r2,skip
        nop     ; Delay slot 1
        nop     ; Delay slot 2
        nop     ; Delay slot 3

        add     r0,r0,55
        jump.eq r14,r14,endprog     ; No need for uncond. jump
        nop     ; Delay slot 1
        nop     ; Delay slot 2
        nop     ; Delay slot 3

skip:
```

```
        set     r3,1
        add     r0,r0,48
        jump.neq r1,r3,endprog
        nop      ; Delay slot 1
        nop      ; Delay slot 2
        nop      ; Delay slot 3
        add      r0,r0,32
endprog:
```

Bonus question: What does program 1 do? [1]

Program 2

```
; Use loop unrolling to avoid need for repeat loop
        set     r14,0  ; Constant 0
        set     r15,18 ; Loop counter
loop:
        ld       r2,[r0]
        add      r0,r0,1
        st       [r1],r2
        add      r1,r1,1
        add      r15,r15,-1
        ld       r2,[r0]
        jump.neq r15,r14,loop
        add      r0,r0,1        ; Delay slot 1
        st       [r1],r2        ; Delay slot 2
        add      r1,r1,1        ; Delay slot 3
; Total cost: 182 clock cycles.
```



[1] Answer to bonus question: 51-01 seulav eht rof desu eb dluohs F-A ro f-a rehtehw syas galF .IICSA ni lamicedaxeh ot 51 ot 0 morf eulav a strevnoC

Solution proposal for 4.2

The original version of this question was constrained in such a way that an easy implementation of the hardware was possible (as long as you realize that it is possible to unroll the loops), whereas this version of the exercise was constrained in such a way that a more advanced PC FSM had to be designed. For program 2, this means that we have to use a zero overhead loop instruction. For program 1 it means that we probably need to have jumps with few delay slots (or none at all) Luckily program 1 is constrained in such a way that the execution cost is mostly irrelevant.

Therefore we need a zero overhead hardware loop instruction. And the conditional branches that we implement should have no delay slots (regardless of the performance impact).

Proposed `jump` Instructions

```
jump.lt OpA, OpB, targetaddr ; Jump if OpA less than OpB
jump.eq OpA, OpB, targetaddr ; Jump if OpA is equal to OpB
repeat num, iterations ; Repeat num instructions iterations times
```

Encoding and Delay Slots

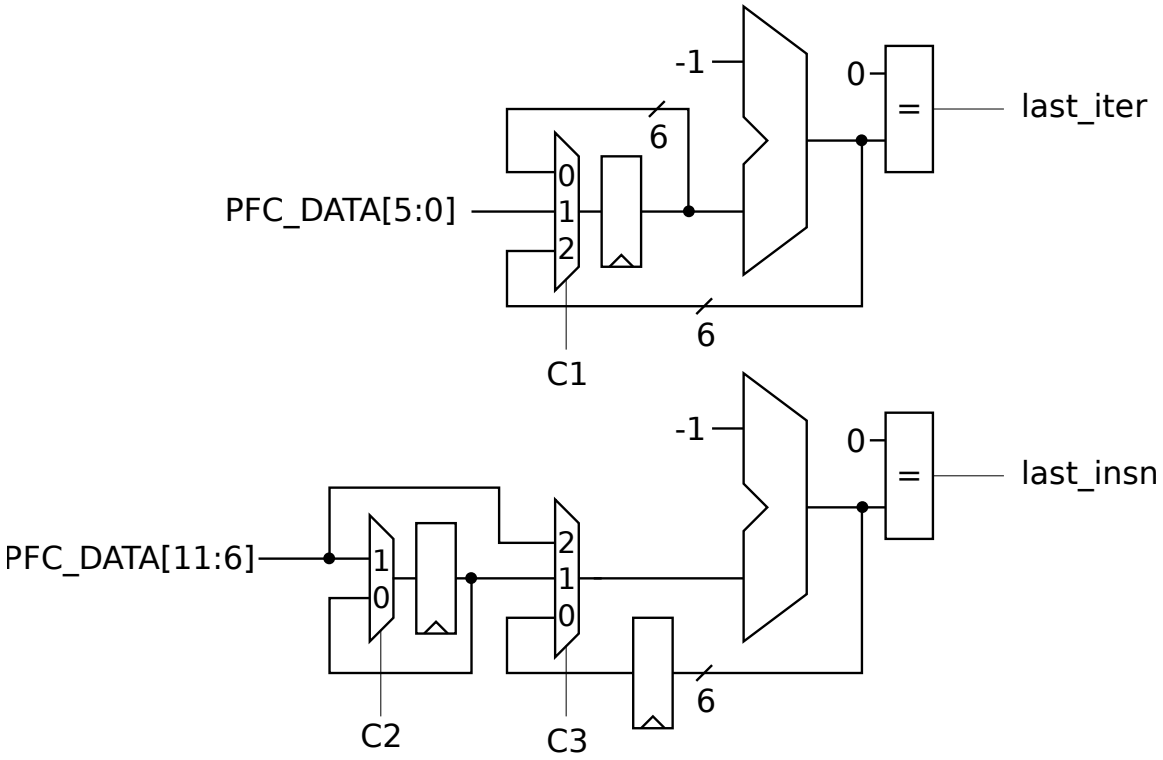
- `jump.lt`: Encoded as `PFC_OP == 10000`. 0 delay slots (For a penalty of 3 clock cycles)

- `jump.eq`: Encoded as `PFC_OP == 10001`. 0 delay slots (For a penalty of 3 clock cycles)

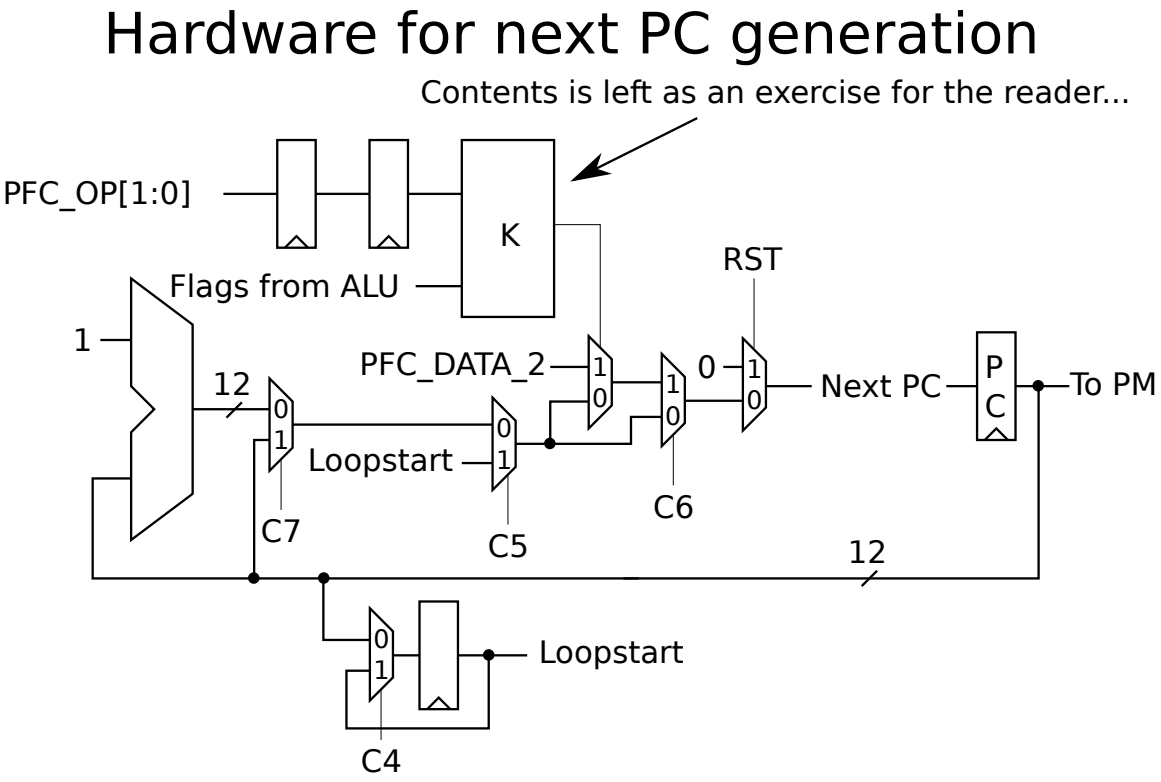
- `repeat`: Encoded as `PFC_OP == 10011`. (Not relevant to discuss delay slots here.) The number of iterations is encoded in `PFC_DATA[11:6]` while the number of instructions in the loop is encoded in `PFC_DATA[5:0]`.

Note that the encoding of the number of iterations would be slightly different if a loop with only one instruction is to be executed on this PC FSM. (See if you can figure out why...)

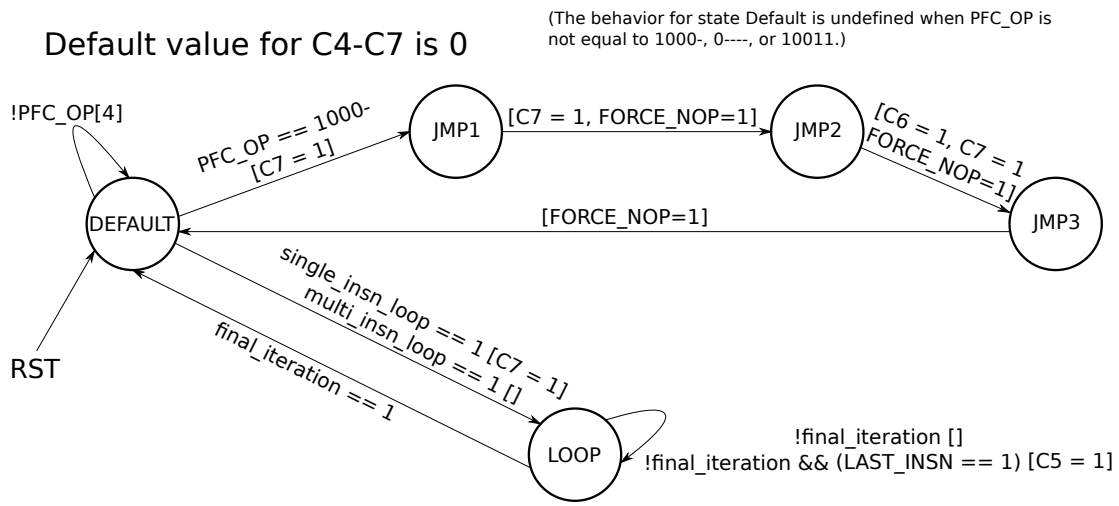
Hardware for loop counter




```
// Control signal generation:

always @* begin
  C1 = 0; C2 = 0; C3 = 0;
  if (PFC_OP[4:0] == 5'b10011) begin
    C1 = 1; C2 = 1; C3 = 2;
  end else if(last_insn) begin
    C1 = 2; C3 = 1;
  end
end
```




```
single_insn_loop = (PFC_OP == 5'b10011) && (last_insn == 1);
multi_insn_loop = (PFC_OP == 5'b10011) && (last_insn != 1);
final_iteration = (last_iter == 1) && (last_insn == 1);
```



Note that K is not defined here. The contents is fairly trivial and left as an exercise for the reader. (The convention used in this FSM is that outputs are specified in square brackets.)

Assembly Program 1

```
set r2,10
nop
jump.lt r0,r2,skip
```

```
        add     r0,r0,55
        jump.eq r14,r14,endprog     ; No need for uncond. jump
skip:
        set     r3,1
        add     r0,r0,80 ; 48 + 32
        jump.eq r1,r3,endprog  ; We negate the condition here to
        add     r0,r0,-32     ; avoid having to implement jump.neq
endprog:
```

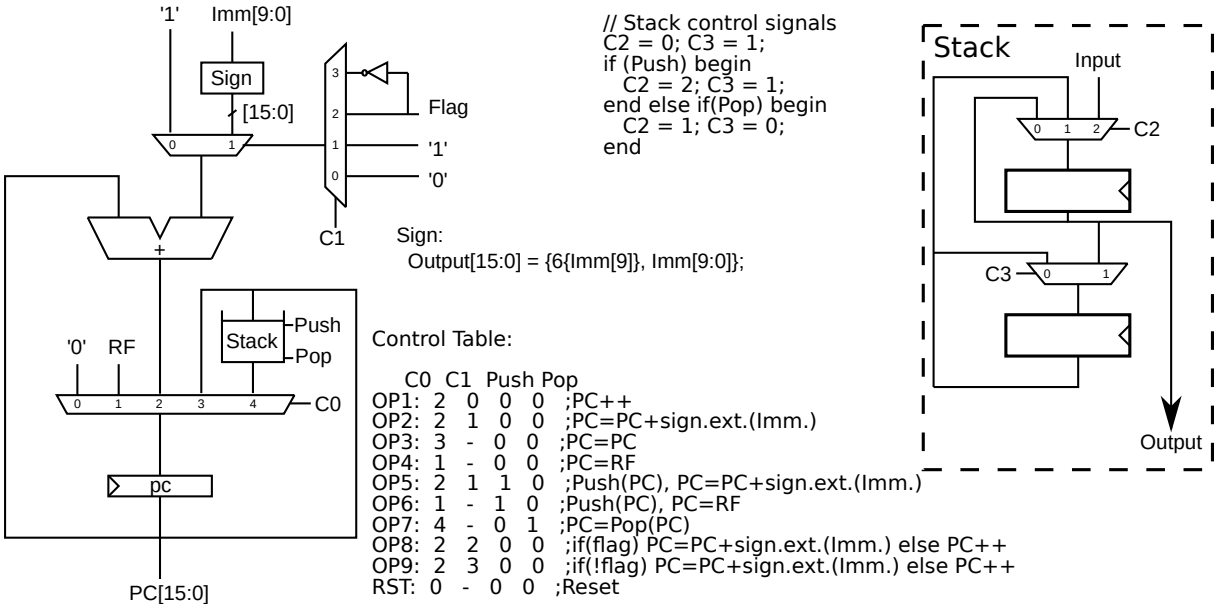
Assembly Program 2

```
        repeat 4, 36
        ld r2,[r0]
        add r0,r0,1
        st [r1],r2
        add r1,r1,1
```

Conclusions

This solution is certainly much easier on the assembler programmer (especially for the second program), but as you can see, the hardware solution is far more complex. This is a trade-off you can make as an ASIP designer. If you do not envision that a large amount of program code needs to be written for your ASIP you may want to simplify the hardware to reduce the verification cost. If, on the other hand, you expect that a huge amount of code will be written for your processor it is probably worthwhile to spend extra time on hardware development to reduce the cost of software development. (Although a compiler will help with the software development effort, this still holds true, especially if you envision that the critical parts of a program will still need to be written as optimized assembler code.)

Solution proposal for 4.3



Solution proposal for 4.4

There are many different ways to solve this exercise by trading off hardware off hardware complexity against software complexity. See the notes at the end of this solution proposal for some other other ideas about how this exercise can be solved.

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 it is assumed 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.

// Note: 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 does not 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
```

Alternative Solutions

Efforts have been made to keep the solution above fairly "mainstream" with few instructions that would not 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).

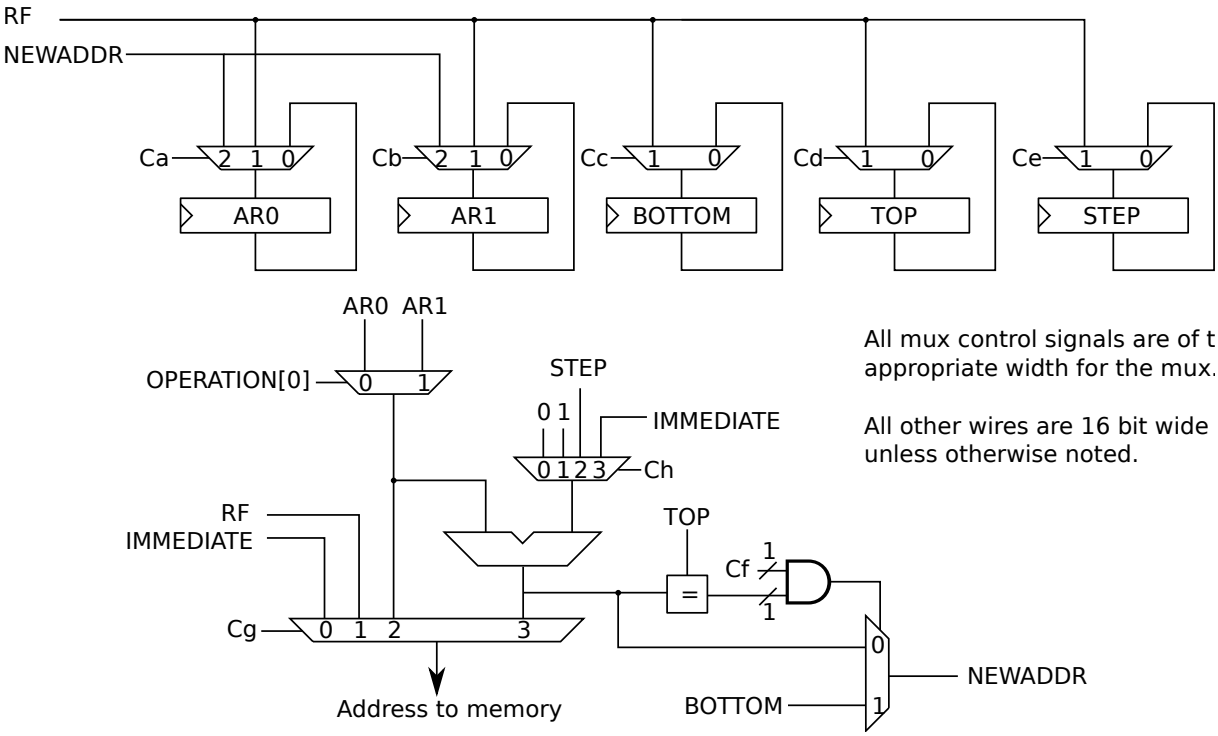
Another way that could be used to simplify the assembly code a lot is to use a conditional call instruction. This is not commonly found in real processors, but it could have simplified `main()` and `worker()` quite a lot[2].

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.)

[2]In most processors there is little to gain from conditional calls however, as such a call would need to setup the parameters for the subroutine call anyway. In such a situation you would have no need for the conditional call, as the condition would be evaluated before setting up the parameters for the subroutine call.

12 AGU Solution Proposals

Solution proposal for 5.1

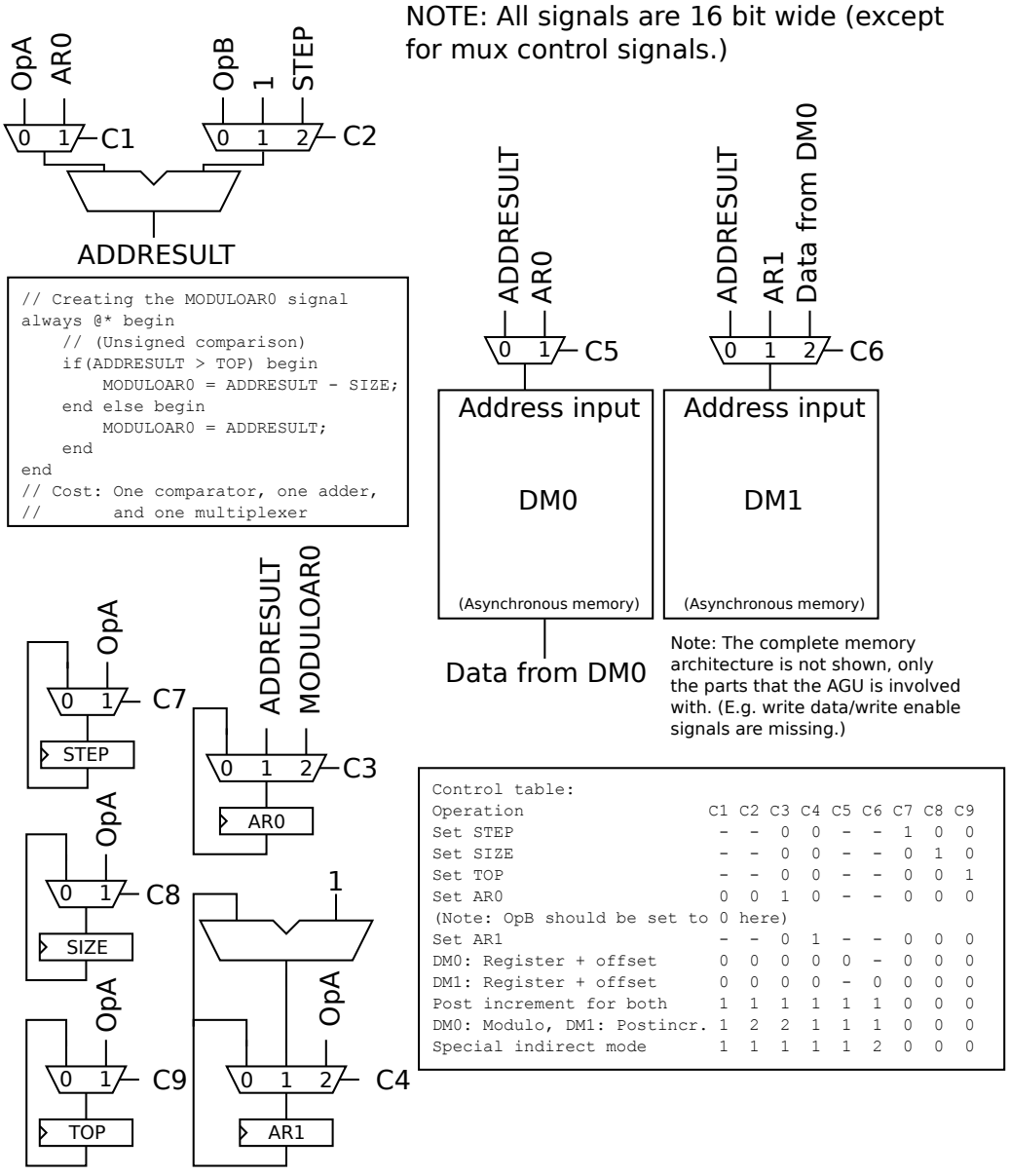


Operation	Ca	Cb	Cc	Cd	Ce	Cf	Cg	Ch
0000	1	0	0	0	0	x	x	x
0100	0	0	0	0	0	x	0	x
0110	2	0	0	0	0	0	2	2
1000	0	0	0	0	0	0	x	3
1011	0	2	0	0	0	1	2	1

Solution proposal for 5.2

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

Part B: AGU Schematic and Control Table



Control table:

Operation	C1	C2	C3	C4	C5	C6	C7	C8	C9
Set STEP	-	-	0	0	-	-	1	0	0
Set SIZE	-	-	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 SIZE, r2  ; buffersize in r2
        set END, r3   ; last in r3
        set STEP, r4  ; step in r4
        CLR ACR
        repeat 128
        conv ACR,DM0[AR0%++], DM1[AR1++]
        ret

interleaver:
        load    r1, DM0[r0+31]
        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
```

Solution proposal for 5.3

Required addressing modes for DM0:

- Modulo addressing, post-increment, step-size 1
- Address register + offset

Required addressing modes for DM1:

- Post increment with step size 1

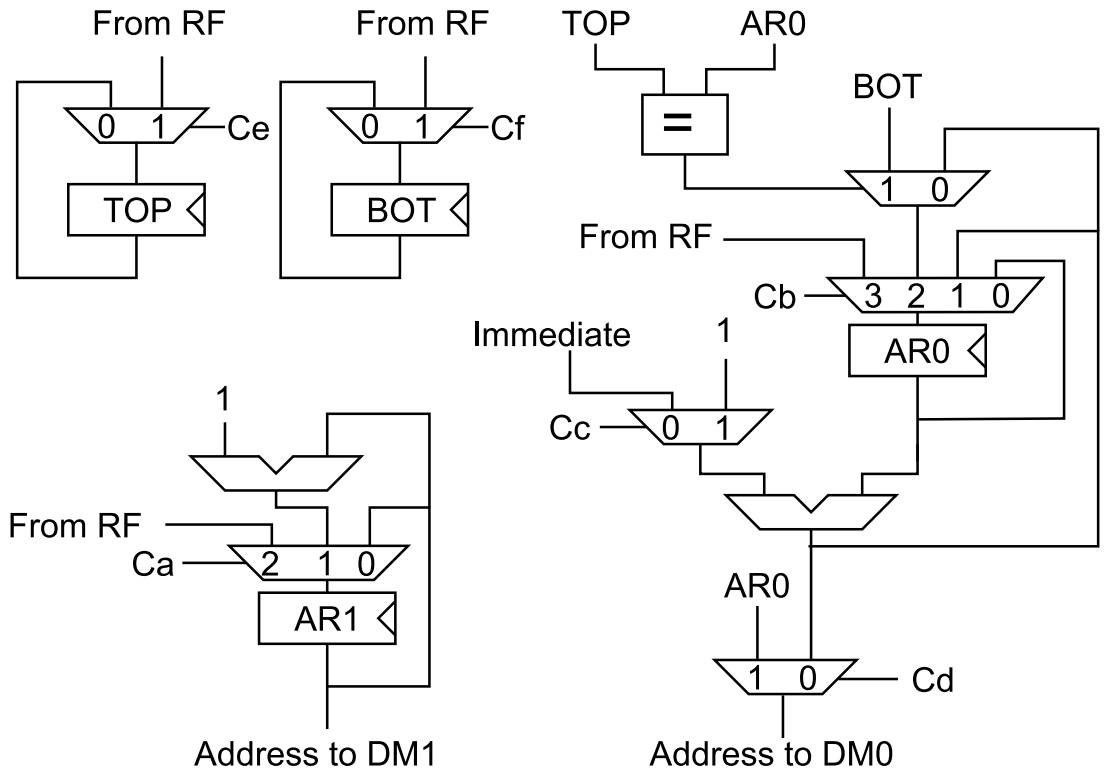
Note that the hardware schematic on the next page is checking for equality before AR0 has been increased by 1 whereas the desired behavior in the given pseudo code is that this check should happen after the `samplesptr` has been increased. This can be fixed by decreasing the value written into the `TOP` 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]

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

Assembly 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

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
```

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.

Solution proposal for 5.4

Exercise 5.2 was made relatively easy as 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 ASIC, synchronous memories should be used as it is hard to create a large and efficient asynchronous memory.)

An alternative would be to simply extend the pipeline when using memory accesses with indirect addressing mode. However, in that case 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). (Such an instruction would also be useful when copying data from one memory to another.)

```
// 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  r1,DM1[AR1++]
        store  r2,DM1[AR1++]
        store  r3,DM1[AR1++]

        ret
```

13 Accelerated Instructions Solution Proposals

Solution proposal for 6.1

There are a few different ways to solve this exercise, especially if you do not care about a critical path from the memory and through the multiplier. However, the key to solving the exercise at all is to realize that you will need read from dm0 100 times and write to dm1 100 times during the repeat loop.

Scheduling table:

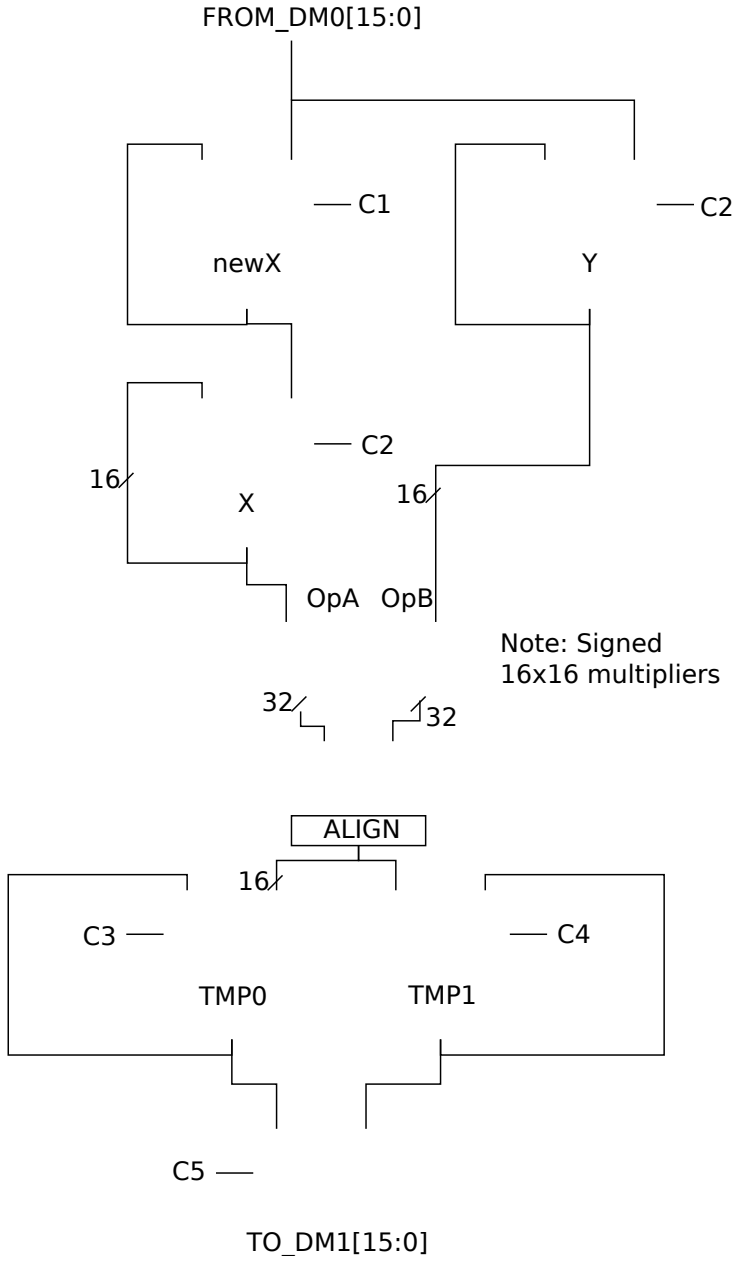
nr	DM0	ROT	DM1
1	`newX = DM0`		
2	`Y = DM0, X = newX`		
3	`newX = DM0`	`tmp0 = A*X + B*Y`	
4	`Y = DM0, X = newX`	`tmp1 = C*X + D*Y`	`DM1 = tmp0`
5	`newX = DM0`	`tmp0 = A*X + B*Y`	`DM1 = tmp1`
6		`tmp1 = C*X+D*Y`	`DM1 = tmp0`
7			`DM1 = tmp1`

Operations 1, 2, and 3 constitute the prologue. Operations 4 and 5 are repeated 49 times. The epilogue is operations 6 and 7.

- **rot1:** `newX = dm0[ar0++];`
- **rot2:** `Y = dm0[ar0++]; X = newX;`
- **rot3:** `tmp0 = OpA*X+OpB*Y; newX = dm0[ar0++];`
- **rot4:** `dm1[ar1++] = tmp0; tmp1 = OpA*X+OpB*Y; Y = dm0[ar0++]; X = newX;`
- **rot5:** `dm1[ar1++] = tmp1; tmp0 = OpA*X+OpB*Y; newX = dm0[ar0++];`
- **rot6:** `dm1[ar1++] = tmp0; tmp1 = OpA*X+OpB*Y;`
- **rot7:** `dm1[ar1++] = tmp1;`

The following assembly program will work:

```
rotate_vector:
    ld   r4,dm0[r0]
    ld   r5,dm0[r0+1]
    ld   r6,dm0[r0+2]
    ld   r7,dm0[r0+3]
    move ar0,r1
    move ar1,r2

    rot1   ; Prologue
    rot2
    rot3   r4,r5

    repeat 49, endloop
    rot4   r6,r7
    rot5   r4,r5
endloop:

    rot6   r6,r7 ; Epilogue
    rot7

    ret
```



Content of ALIGN: `assign out[15:0] = in[30:15];`

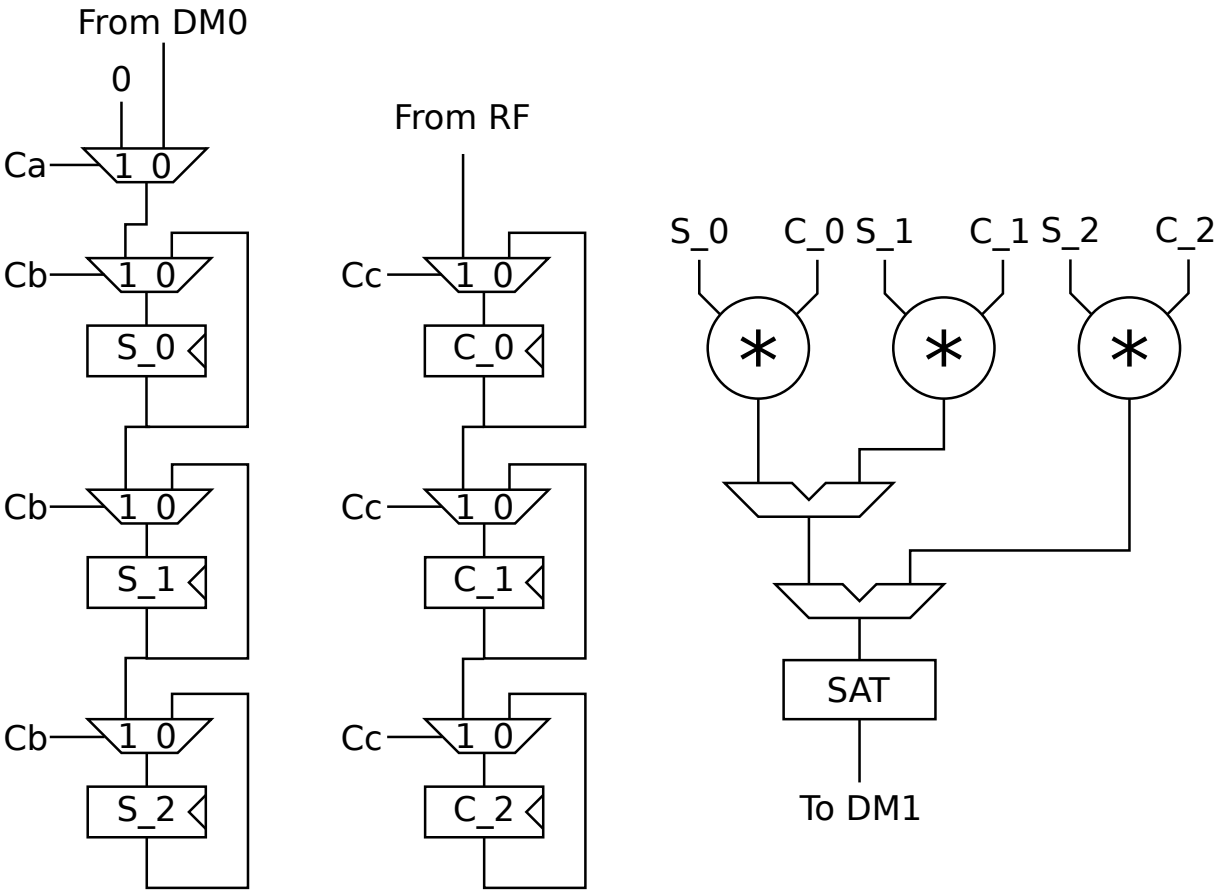
Insn	C1	C2	C3	C4	C5
rot1	1	0	0	0	-
rot2	0	1	0	0	-
rot3	1	0	1	0	-
rot4	0	1	0	1	0
rot5	1	0	1	0	1
rot6	0	0	0	1	0
rot7	0	0	0	0	1

Solution proposal for 6.2

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
```

Solution proposal for 6.3

This is one possible version of pseudo assembly code for `Read1bit()`.

```
Read1bit:
    load      r0, [r15]  // CurrentAddress is in R15
    leftshift r1, 1, r14 // 1 << CurrentBit (which is in R14)
    and       r0, r0, r1 // Memval & Bitmask

    bne       bitwas1
    set       r0,1       // Set Bit = 1 in delay slot

    set       r0,0       // Z flag was set

bitwas1:
    add       r14,1,r14  // CurrentBit++
    cmp       r14,16

    bne       no_new_word
    nop

    set       r14,0      // CurrentBit = 0
    add       r15,1,r15  // CurrentAddress++

no_new_word:
    ret       // Return value is in r0
```

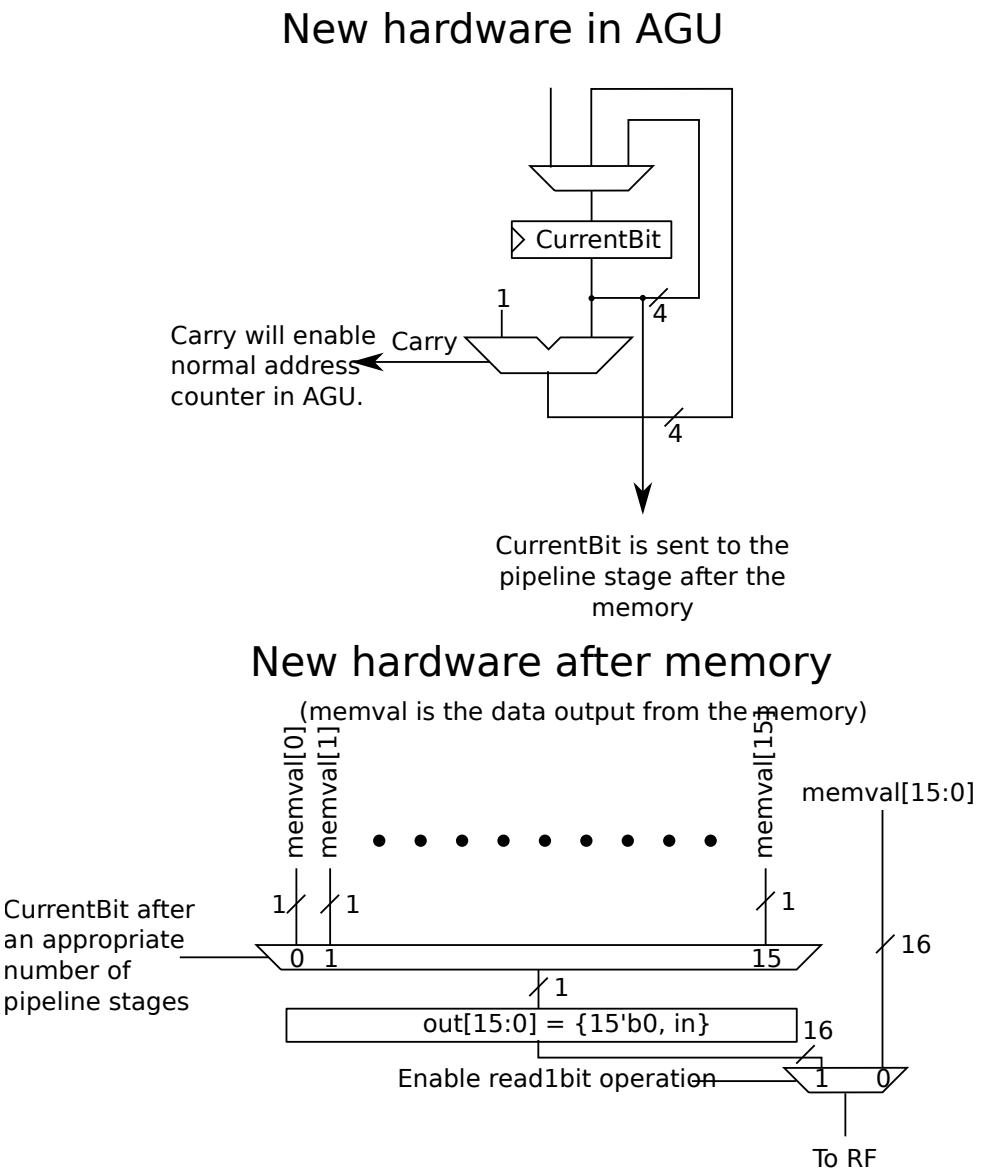
Assumptions:

- The processor has full forwarding.
- A conditional branch is predicted as not taken. If this is wrong, it costs 3 clock cycles to flush the pipeline.
- A load from memory takes 2 clock cycles to complete. The processor will stall if the program tries to use the loaded value too early.
- A return takes 3 clock cycles to complete.

Best case: The first bne is not taken. The second bne is not taken. The number of executed instructions are: 13. Plus 1 extra clock cycle for the load and 2 extra clock cycles for the return. The best case is 16 clock cycles.

Worst case: Both `bne`s are taken. The number of executed instructions are: 10. Plus 1 extra clock cycle for the load, 2 extra clock cycles for the return, and 2×3 extra clock cycles to flush the pipeline when both bnes are wrong. The worst case is therefore 19 clock cycles.

- The AGU has to be modified to support a 4-bit counter used to index bits.
- After the memory you should insert a multiplexer to select the correct bit as indicated by the `CurrentBit` signal.



Conclusion: If you use `Read1bit` quite often the extra hardware cost is negligible next to the performance increase you will get!

14 Solution Proposal for the Design Challenge

14.1 Correction checklist

Since you have probably not created exactly the same solution that I propose in this chapter I have the following checklist that should allow you to check whether your solution is correct. I also encourage you to team up with other course participants and cross check your solutions.

For all functions implemented in assembler:

- Check that the assembler code performs the correct function
- Check that the appropriate number of delay slots are used for branches
- Check that the cycle cost constraints of the function is ok
- Check that the program memory cost constraints of the function is ok

For all hardware modules:

- Check that all instructions used in the assembler programs are actually implemented in at least one hardware unit as appropriate.
- Check that it is possible to perform a NOP instruction (e.g., some hardware registers shouldn't change unless required by the instruction)
- Check that there are no combinational loops
- Check that the hardware performs as required for each instruction (by using the control tables)

14.2 A Note on the Proposed Solutions

I have tried to keep the assembly instruction set fairly general. However, for many functions I have also discussed alternative solutions. Also note that in my solution proposal I have stayed away from solutions that are likely to result in a long critical path.

One example of this is that I'll place a register in the MAC unit to immediately register the values fetched from the data memories. Another example is that I'll put a register in the ALU before sending the flags to the PC module.

However, as there are no constraints listed in the exercise, feel free to ignore these constraints for your solutions.

14.2.1 A note on delay slots

I have decided to use 3 delay slots for my conditional branches. In fact, it would be possible to reduce this based on the given pipeline, but I'm using this number to avoid a possible critical path from OPA/OPB, through the ALU, through the PC and to the address input of the program memory.

14.3 Function 1: sanitize_buffer_values

Notes

One iteration of the loop needs to execute in 6 clock cycles or less. This is hard to achieve due to data hazards effect since the pipeline has no forwarding.

Solution

Use loop unrolling to mitigate pipeline hazards. Use min/max instructions instead of conditional branches to reduce the number of instructions in the loop.

```
; Buffersize in r0
; Technique: Unroll loop 3 times
    set AR0,inbuffer
    set AR1,outbuffer
    set r14,0
loop:
    add r0,-1,r0
    ld r1, DM1[AR0++]
    ld r2, DM1[AR0++]
    ld r3, DM1[AR0++]
    max r1,-230,r1
    max r2,-230,r2
    max r3,-230,r3
    min r1,5423,r1
    min r2,5423,r2
    min r3,5423,r3
    lsl r1,2,r1     ; r1 = r1 << 2
    lsl r2,2,r2
    lsl r3,2,r3
    bne r0,r14,loop
    st DM1[AR1++], r1 ; Three delay slots
    st DM1[AR1++], r2 ;
```

```
st DM1[AR1++], r3 ;

ret
nop
```

Alternative solutions

It would be possible to combine the min and max instruction into one instruction. However, since we can only read two operands from the register file at the same time, this would mean that we would have to specify at least one of the boundaries separately. For example:

```
set highboundary 5423
set lowboundary -230
...
...
ld r2, dm0[ar0++]
minmax r0,r0
minmax r1,r1
minmax r2,r2
```

We could of course also include the left shift in this minmax instruction. Finally, we could include support for this instruction in the MAC unit, which would allow us to combine the load, min, max, and shift into one instruction. (Although this wouldn't really fit into the pipeline as discussed below.)

14.4 Function 2: butterfly

Notes

This function looks a bit bit weird as it is probably intended to do a butterfly calculation inside an FFT. However, the calculations are done using integer arithmetic which is quite odd. In reality I would probably like to discuss this code with the application engineers who proposed it, but for the sake of the design challenge, lets see if we can actually implement this in a relatively reasonable manner.

There are 6 loads, 4 stores, 4 multiplications and 6 add/subtracts. (Excluding address calculations.) We also need one return operation. Naively, that would mean a total of 21 instructions.

Unfortunately the indata and result address calculations are somewhat hard to perform. For example, to calculate `indata[i+j+1]` needs 2 constants and two register values:

- i: Register value
- j: Register value
- indata pointer: Constant

- 1: Constant

While we can reduce the number of constants by merging the indata pointer and the 1 into one constant, we will still need to supply 3 operands to the load instruction. This is unfortunately not possible as we can only supply two operands at the same time in this architecture.

That is, we either need to use an address register or use extra instructions to calculate for example $i + j$. Note also that the situation is even worse for store instructions since we also need to supply the value to be stored.

Solution

It is clear that we need to overlap some of the operations described above in some way. We also need to solve the address calculation problem.

At a first glance it would seem obvious that we can use the fact that the memory can be connected to the MAC unit as seen in the pipeline figure. However, a reasonable hardware design of the MAC unit will register the values fetched from the memory immediately to avoid a long critical path.

This means that it will be tricky to introduce an instruction that loads values from the memories, performs an operation on those values and writes back the result to the general purpose register file. The pipeline for such an instruction would be one register longer than the normal pipeline, opening the door for structural hazards.

So if we use this approach we would need to store values in the accumulator register and read out that value at a later point.

Something like this approach should work:

```
; We assume R0 contains i, R1 contains j, and R2 contains c
set AR0, indata + r0
set AR1, coeff + r2       ; r2 contains c

mul.sat ACR0,  DM0[AR0++ + R1] * DM1[AR1++]
mac.sat ACR0, -DM0[AR0-- + R1] * DM1[AR1]
mul.sat ACR1,  DM0[AR0++ + R1] * DM1[AR1--]
mac.sat ACR1,  DM0[AR0-- + R1] * DM1[AR1++]

; For mac.sat we need to saturate both the multiplication
; and the addition/subtraction! (bletch!)

ld r3, DM0[AR0++]
move.int r5, acr0    ; Move integer result to r5
ld r4, DM0[AR0--]
move.int r6, acr1
add.sat r7,r3,r5   ; tmp7
sub.sat r9,r3,r5   ; tmp9 ; Rescheduled to avoid data hazard for tmp8
add.sat r8,r4,r6   ; tmp8
sub.sat r10,r4,r6  ; tmp10
```

```
set AR0, outdata + r0
st DM0[AR0++], r7
st DM0[AR0--], r8
st DM0[AR0++ + R1], r9
ret
st DM0[AR0-- + R1], r10

; Exactly 20 instructions, yay!
```

14.5 Function 3: fir()

Notes

This is quite straight-forward. However, we need to have a modulo addressing mode to handle this function efficiently. It is also necessary to use a repeat instruction to handle the clock cycle and program memory constraints.

Solution

Note that we decrement r4 by one before assigning it to TOP. This means that we can shorten the critical path of the AGU by comparing AR0 with TOP instead of comparing AR0+1 with TOP.

```
add r4,-1,r4    ; decrement end_ptr by one
set AR0, r2 ; fir_ptr
set BOT, r3; start_ptr
set TOP, r4; end_ptr
set AR1, r1

st DM0[AR0%++], r0 ; newsample
; AR0%++ - Increment AR0 using circular buffer mode

clr ACR0
repeat 32
MAC ACR0, DM0[AR%++], DM1[AR1++]
ret
move.rnd.sat r0, ACR0

; 42 clock cycles
```

14.6 Function 4: getbits()

```
; r0: val, r1: numbits, r2: offset
set r3,1  ; The set and lsl could be replaced by
nop       ; a table lookup if we want to optimize
nop       ; the execution speed of this function.
```

```
    lsl r4,r1,r3        ; r0 = r3 << r1
    nop
    nop
    lsr r0,r2,r0        ; r0 = r0 >> r2
    add r4,-1,r4
    nop
    nop
    and r0,r0,r4
    ret                 ; We could move up ret, but then
    nop                 ; we might not be able to use
                        ; r0 immediately after we return
                        ; from the subroutine.

    ; A total of 13 clock cycles which fulfills the
    ; constraints placed on us.
```

14.7 Function 5: preprocess()

Notes

As we have no clock cycle or program memory constraints here, it makes sense to implement this in such a way that we minimize the amount of hardware.

This function was just included to make sure that we have a branch instruction separate from branch if equal/not equal. (Although it is possible to solve this exercise and only use branch if not equal. But that solution is very cumbersome so I didn't include it here.)

Solution

```
; We assume a is stored in r0 and b in r1
;
; We also assume we should pass the return value in r0 (although this
; is not stated in the exercise)

    bgt r0,r1, abigger    ; bgt - branch if r0 greater than r1
    set r2,56
    set r3,12
    nop

    bgt r3,r1, b_smaller
    nop
    nop
    nop

    ret                   ; return a + 12
    add r0,r0,12

b_smaller:
    ret
    add r0,r1,12          ; return b + 12
```

```
abigger:
    bgt r0,r2, abigger_56
    nop
    nop
    nop

    ret                    ; return b - 56
    add  r0,r1,-56

abigger_56:
    ret
    add r0,r0,-56          ; Return a - 56
```

14.8 Function 6: prepare_data()

Notes

As I mentioned during the lecture, there was a bug in the design challenge. The intention was that the cycle constraints for this task would be around 700 cycles instead of 1900 cycles. Even so, the solution is fairly trivial, except for the bitreversed part.

```
      set   AR0, indata
      set   AR1, 0
      set   r15, outdata
      set   BITREVSTEP, 0000001000000000b ; Configurable stepsize
      set   r14,0 ; A zero for bne comparison...

      set   r0, 32 ; Loop is unrolled 4 times
loop:
      ld    r1,DM0[AR0++]
      ld    r2,DM0[AR0++]
      ld    r3,DM0[AR0++]
      ld    r4,DM0[AR0++]
      ld    r5,DM0[AR0++]
      ld    r6,DM0[AR0++]
      ld    r7,DM0[AR0++]
      ld    r8,DM0[AR0++]
      add   r0,-1,r0       ; Decrement loop counter in advance
      avg   r1,r1,r2       ; Average operation
      avg   r3,r3,r4
      avg   r5,r5,r6
      avg   r7,r7,r8
      store DM0[bitrev(AR1+=BITREVSTEP)+r15], r1
      ; AR1+=BITREVSTEP is a post-increment operation here
      bne   r0,r14, loop
      store DM0[bitrev(AR1+=BITREVSTEP)+r15], r3   ; <-- delay slot
      store DM0[bitrev(AR1+=BITREVSTEP)+r15], r5   ; <-- delay slot
      store DM0[bitrev(AR1+=BITREVSTEP)+r15], r7   ; <-- delay slot
```

14.9 Function 7: dot16()

Notes

Nothing really tricky here except that we need to scale the accumulator at some point. Also, we need to figure out how many guard bits we'll need. Since we are using fractional values in this FIR filter, the range of the result after 16 iterations is [-16,16). Therefore we'll need 4 guard bits. Also, it makes sense to have an ACR register which is 32 bits wide, except for the guard bits. (Since we get a 32-bit result out of a 16x16 bit multiplication.)

Solution

```
    set AR0, r0 ; ptr1
    set AR1, r1 ; ptr2

    clear ACR0
    repeat 16
    mac.fractional ACR0, DM0[AR0++] * DM1[AR1++]

    scale3 acr0         ; Shift ACR0 right 3 steps (arithmetic shift)
    move.rnd.sat r0, ACR0
    ret
    nop
```

14.10 Function 8: find_val()

Notes

No constraints, so we'll create a straight forward implementation.

```
    set r1, 128    ; r1 = index
    set r2, 64     ; r2 = step
    set r15, 1     ; For while() comparison
    set AR0, data ; pointer to data array


loop:
    bgt r15,r2, finished
    ld  r3,DM0[AR0 + r1]
    nop
    nop

    bne r3, r0, continue
    nop
    nop
    nop

    and r0,r1,r1 ; Fancy way of moving r1 to r0 without having an
                 ; explicit move instruction.
```

```
    ret
    nop


continue:
    bgt  r3, r0, toosmall
    nop
    nop
    nop


    sub.sat  r1,r1, r2  ; We'll use sub.sat here to avoid the
                        ; need for a separate sub instruction
    bgt r0,r3, loop ; Well, we could have used an unconditional
                    ; jump but why add one now if we haven't
                    ; needed it yet?
    lsr r2,r2,1     ; Since step is unsigned we can shift it
    nop             ; right instead of using a division with 2
    nop             ; (if it was signed, an arithmetic shift
                    ; right will round in the wrong direction)


toosmall:
    add r1, r1, r2
    bgt r3,r0, loop
    lsr r2,r2,1
    nop
    nop



finished:
    and r0,r1,r1 ; Fancy way of moving r1 to r0 without having an
                 ; explicit move instruction.
    ret
    nop
```

14.11 Final instruction list

- `add OpW, OpA, OpB`
- `add.sat OpW, OpA, OpB`
- `and OpW, OpA, OpB`
- `avg OpW, OpA, OpB` ; OpW is set to the average value of OpA and OpB (signed)
- `bgt OpA, OpB, branchtarget` ; Branch if OpA is greater than OpB. 3 delay slots
- `bne OpA, OpB, branchtarget` ; Branch if OpA and OpB aren't equal. 3 delay slots
- `call branchtarget` ; Obviously we will need a call subroutine as well. (1 delay slot)
- `clr ACRx` ; Set accumulator to 0
- `ld OpW, DM0[ADDRMODE]`
- `ld OpW, DM1[ADDRMODE]`
- `lsl OpW, OpA, OpB` ; OpW = OpB << OpA
- `lsr OpW, OpA, OpB` ; OpW = OpB >> OpA
- `mac.fractional ACR0, DM0[ADRMODE for AR0] * DM1[AR1++]` ; Fractional multiplication
- `mac.sat ACRx, DM0[ADDRMODE for AR0] * DM1[AR1--]` ; performs sat after multiplication and after addition (bletch)
- `mac.sat ACRx, -DM0[ADDRMODE for AR0] * DM1[AR1]` ; performs sat after multiplication and after addition (bletch)
- `max OpW, OpA, OpB` ; Signed
- `min OpW, OpA, OpB` ; Signed
- `move OpW, ACRx`
- `move.rnd.sat OpW, ACRx` ; OpW = SAT(ROUND(ACRx))
- `mul.sat ACRx, DM0[ADDRMODE for AR0] * DM1[AR1++]` ; performs sat after multiplication
- `mul.sat ACRx, DM0[ADDRMODE for AR0] * DM1[AR1--]` ; performs sat after multiplication
- `repeat num_iter` ; Execute next instruction num_iter times (num_iter $< 2^6$). Must be larger than 2
- `ret` ; Return from subroutine, 1 delay slot

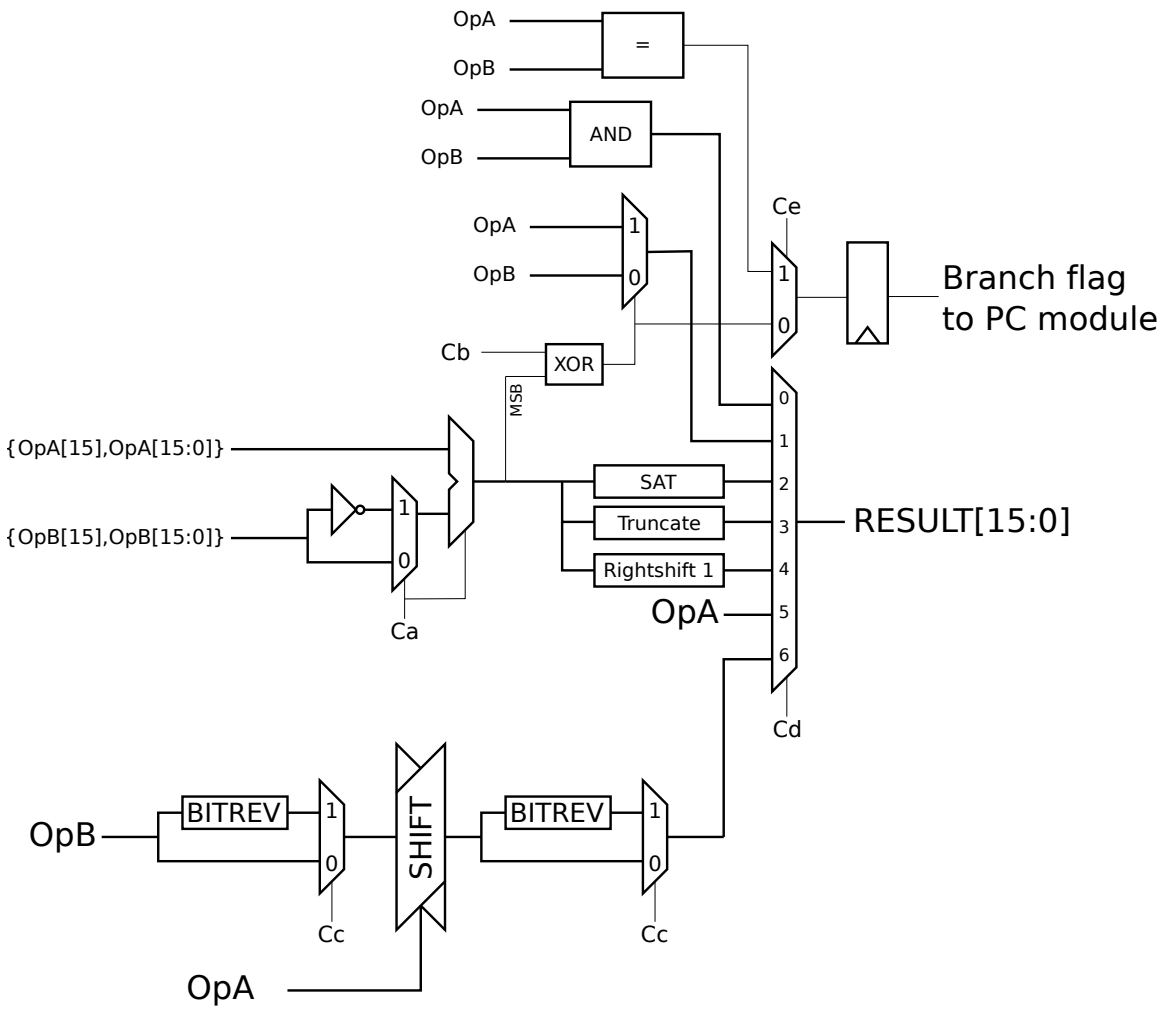
- `scale3 ACRx` ; ACRx = ACRx >> 3 (Arithmetic shift right)
- `set ARx, OpA + OpB` ; set ARx, OpB is emulated as set AR0, 0 + OpB
- `set BITREVSTEP, OpA`
- `set BOT, OpA`
- `set OpW, OpA` ; Can be used to set a register to an immediate value
- `set TOP, OpA`
- `st DM0[ADDRMODE], OpA`
- `st DM1[ADDRMODE], OpA` ; We will most likely need this as well...
- `sub.sat OpW, OpA, OpB` ; OpW = SAT(OpA - OpB)

14.12 Required addressing modes

- `ARx++`
- `ARx--`
- `ARx`
- `ARx++ + OpB`
- `ARx-- + OpB`
- `ARx%++` ; Circular addressing
- `bitrev(ARx += BITREVSTEP)+OpB` ; Bit reversed addressing + base offset

(The reason that we are limiting our addressing modes for DM1 in the MAC instruction is that we only have 7 bits to encode both the instruction and the addressing modes in. If we had a generic addressing mode for both DM0 and DM1 here we wouldn't be able to fit everything into the instruction word.) Note that the bitreversed addressing mode is post-increment!

14.13 ALU



SHIFT box:

```
out[15:0] = in[15:0] >> OpA;
```

SAT box:

```
if (in[16] != in[15]) begin
    if(in[16]) begin
        out[15:0] = 16'h8000;
    end else begin
        out[15:0] = 16'h7fff;
    end
end else begin
    out[15:0] = in[15:0];
end
```

Truncate box:

```
out[15:0] = in[15:0]
```

BITREV box:

```
out[15:0] = in[0:15]; // (This is cheating since this
                      // syntax doesn't work in Verilog...)
```

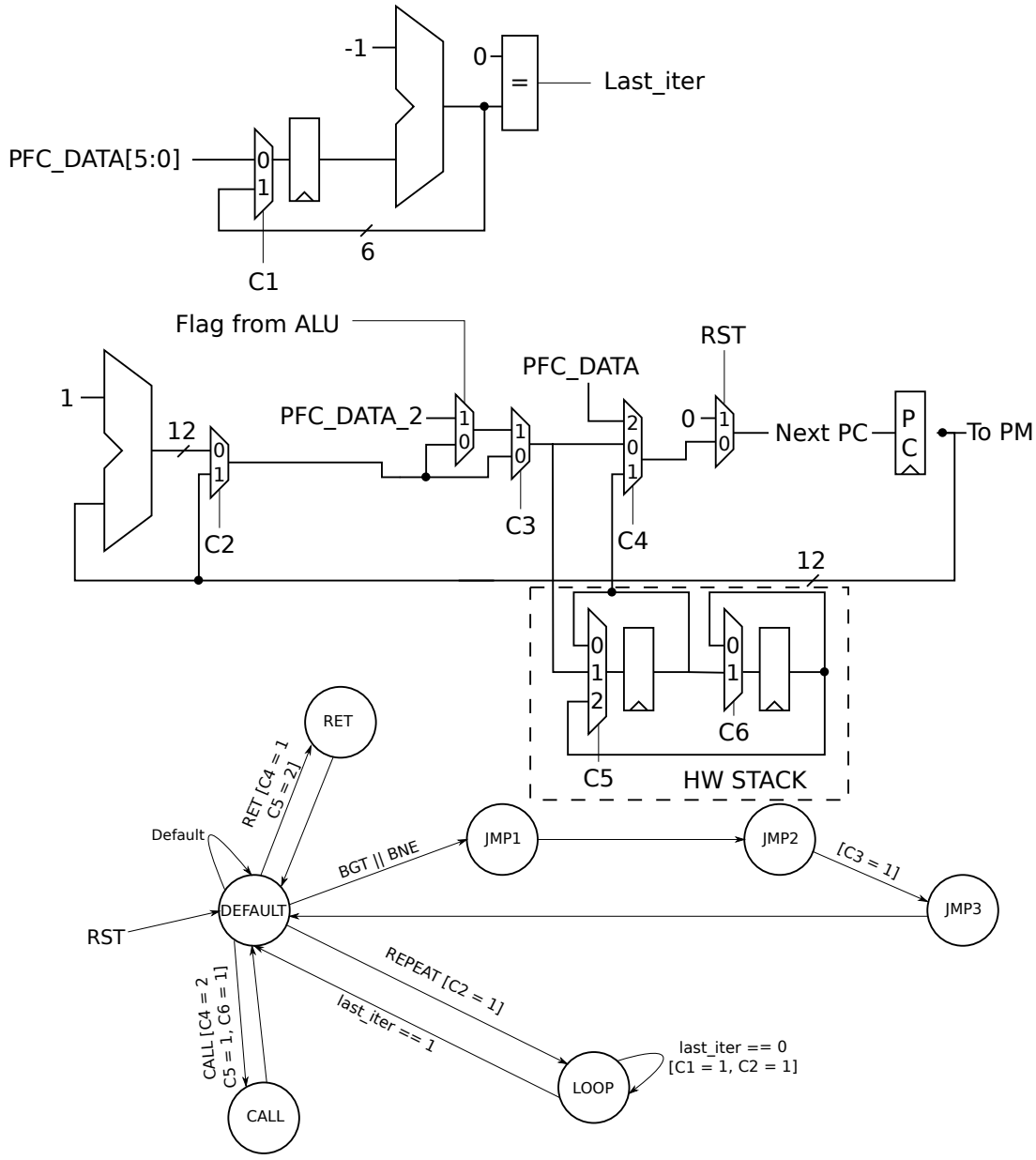
Rightshift 1:

```
out[15:0] = in[16:1];
```

Control table:

INSN	Ca	Cb	Cc	Cd	C
add	0	x	x	3	x
add.sat	0	x	x	2	x
sub.sat	1	x	x	2	x
max	1	1	x	1	x
min	1	0	x	1	x
lsl	x	x	1	6	x
lsr	x	x	0	6	x
bne	x	x	x	x	1
bgt	1	1	x	x	0
set	x	x	x	5	x
and	x	x	x	0	x
avg	0	x	x	4	x

14.14 PC



In the FSM, signals specified within brackets (such as $[C1 = 1]$), are output signals from the FSM for that transition. To minimize clutter in the FSM, if a signal is not mentioned as having a specific value, the value of that signal is assumed to be 0.

Note that the control signals from the instruction decoder reaches PC combinatorially!

Also note that the results are undefined if you put a PFC instruction into a delay slot. (Or if you try to use repeat on a PFC instruction.)

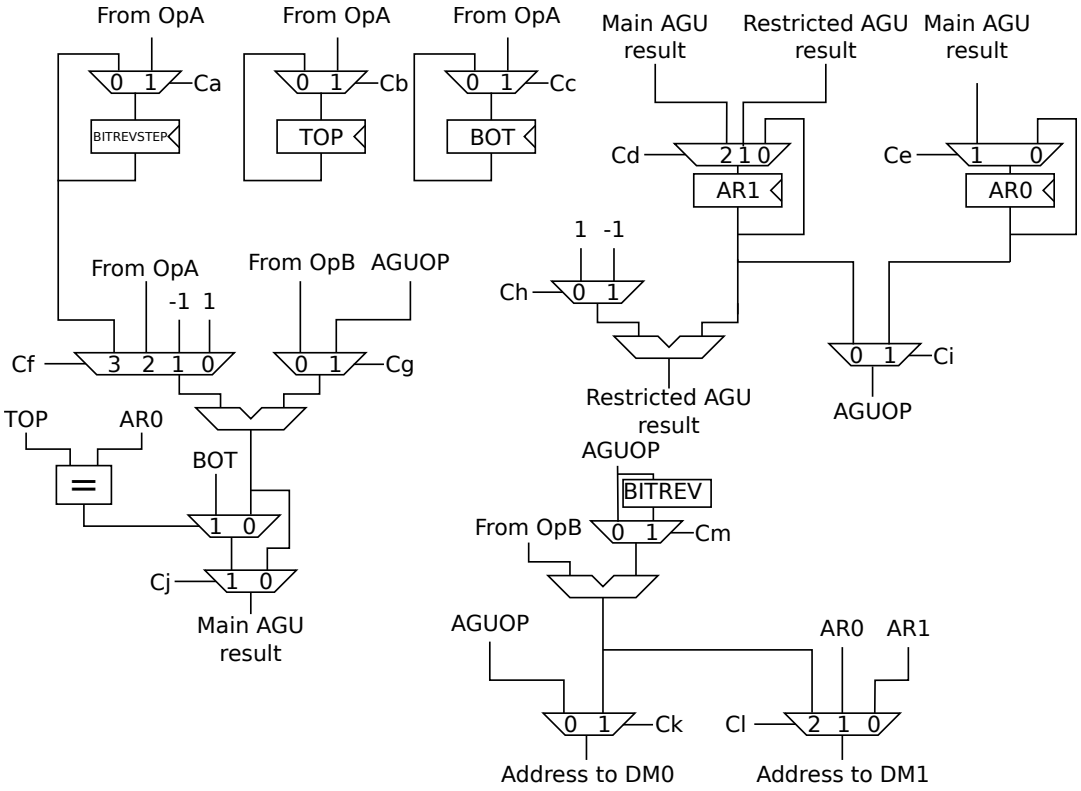
The branch flag from the ALU is delayed one clock cycle inside the ALU unit due to the register, which means that conditional branches will have 3 delay slots. (But unconditional branches have only 1 delay slot.) Also note that the ALU is responsible for handling the two different types of conditional branches that we have.

14.15 AGU

We need two address registers, AR0 and AR1.

For load/store instructions, any addressing modes and any memory can be used.

For convolution, only AR1++, AR1, and AR1– can be used for DM1 whereas any addressing mode can be used for DM0 as long as AR0 is used.



Control table for AGU:

INSN	Ca	Cb	Cc	Cf	Cg	Cj	Cm
set TOP	0	1	0	x	x	x	x
set BOT	0	0	1	x	x	x	x
set BITREVSTEP	1	0	0	x	x	x	x
set ARx,OpA+OpB	0	0	0	2	0	0	x
ARx++	0	0	0	0	1	0	x
ARx–	0	0	0	1	1	0	x
ARx++ + OpB	0	0	0	0	1	0	0
ARx– + OpB	0	0	0	1	1	0	0
ARxbitrev(ARx += BITREVSTEP)+OpB	0	0	0	3	1	0	1

There are also quite a few special cases in this AGU unfortunately, as specified below:

Cd:

If AR1 is not updated: Set to 0
If AR1 is updated (during non-convolution operation): Set to 2
During a convolution (i.e. mac or mul): Set to 1 if AR1 should be updated

Ce:

If AR0 should not be updated: Set to 0
If AR0 should be updated: Set to 1

Ch:

During convolution and AR1++ selected for DM1 addressing: Set to 0
During convolution and AR1– selected for DM1 addressing: Set to 1

Ci:

When AR0 is selected: Set to 1
When AR1 is selected: Set to 0
During convolution: Set to 1
If none of the above: Don't care

Ck:

When addressing DM0 using mode ARx++ + OpB: Set to 1
When addressing DM0 using mode ARx– + OpB: Set to 1
When addressing DM0 using other addressing mode: Set to 0
Otherwise: Don't care

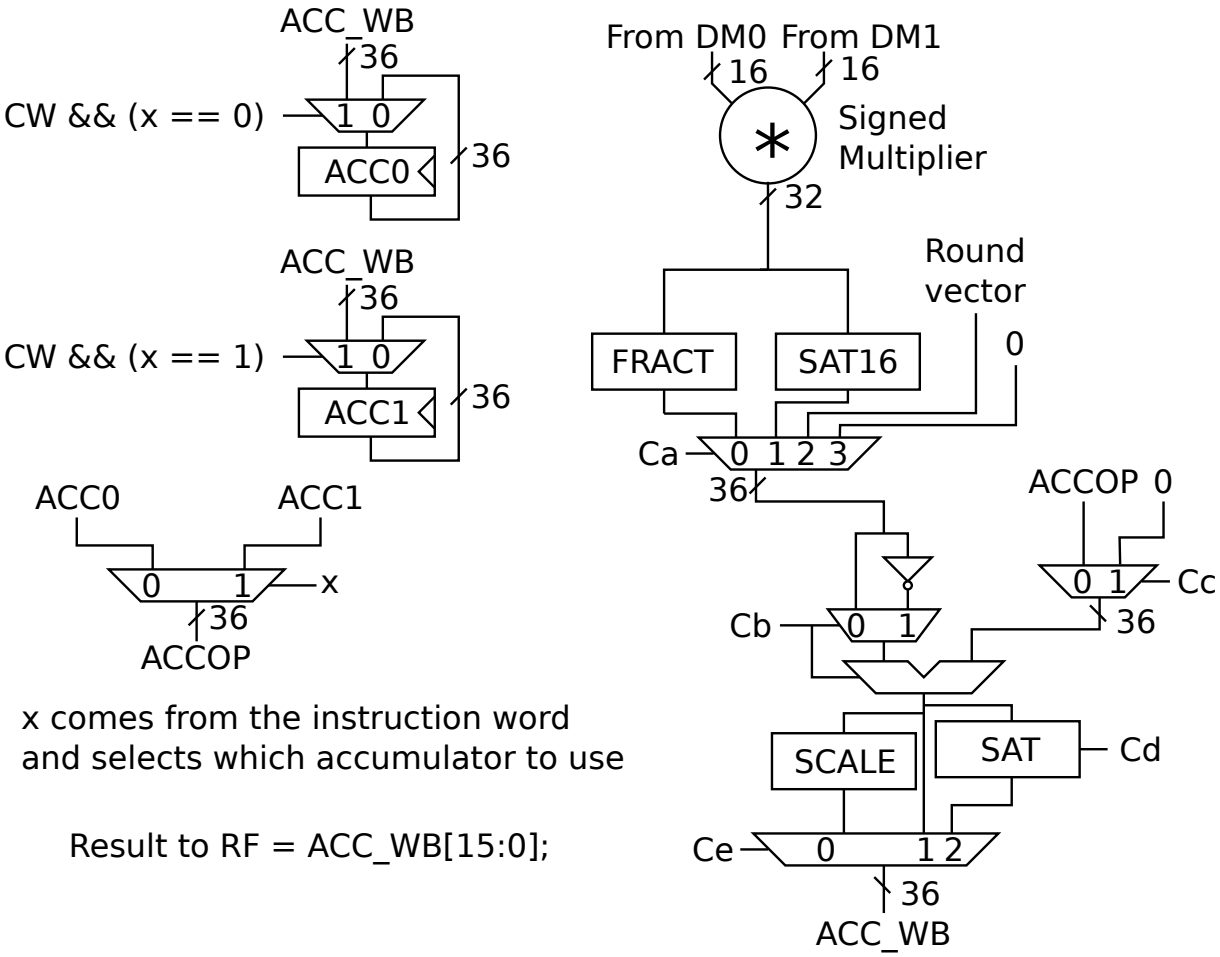
Cl:

When addressing DM1 using mode ARx++ + OpB: Set to 2
When addressing DM1 using mode ARx– + OpB: Set to 2
When addressing DM1 using register AR0: Set to 1
When addressing DM1 using register AR1: Set to 0
During convolution: Set to 0
Otherwise: Don't care

BITREV box:

```
out[15:0] = in[0:15]; // (NOTE: Incorrect Verilog syntax)
```

14.16 MAC



Control table for MAC:

INSN	Ca	Cb	Cc	Cd	Ce	CW
nop	x	x	x	x	x	0
mul.sat	1	0	1	0	3	1
mac.sat	1	tmp1	0	0	3	1
mac.fractional	0	0	0	x	2	1
clr	3	0	1	x	x	1
move.int	3	0	0	x	1	0
move.rnd.sat	2	0	0	1	2	0
scale3	3	0	0	x	0	1

Note: tmp1 is set to 1 when subtracting from the accumulator instead of adding

FRACT:

```
out[35:0] = { {4{in[31]}}, in[30:0], 1'b0};
```

SAT16:

```
if(in[31:15] != {17{in[31]}}) begin
```

```
    out[35:0] = {{21{in[31]}}, {15{~in[31]}} };
end else begin
    out[35:0] = { {4{in[31]}}, in[31:0]};
end

ROUND VECTOR = 36'h0 0000 8000;
```

SAT:

```
if (Cd) begin // Cd controls whether we saturate for
              // move.rnd.sat or for mac.sat
    if(in[35:15] != {21{in[35]}}) begin
        out[35:0] = {{21{in[31]}}, {15{~in[31]}} };
    end else begin
        out[35:0] = in[35:0];
    end
end else begin
    out[35:16] = 20'hxxxxx; // Don't care as these bits are not
                            // used when writing to RF
    if(in[35:31] != {5{in[35]}}) begin
        out[15:0] = { in[35], {15 {~in[35]}}};
    end else begin
        out[35:0] = in[31:16]; // Scale down by 16 as these
                               // bits are used when reading out to RF
    end
end
```

SCALE:

```
out[35:0] = { {3{in[35]}}, in[35:3]}; // Arithmetic shift right by 3
```

15 Version history

- v1.0 Original version.
- v1.1 Fixed wrongly numbered solutions for exercise 2.*
- v1.2 Fixed right shift by 8 in exercise 3.1. Added solution proposal for 2.2. Used the correct MSB bit in exercise 3.3 (35 instead of 36).
- v1.3 Added exercises for tutorial 1. Moved exercise 4.4 to 4.2 General cleanup of the document. Thanks to Olle Seger for help with some of the exercises for tutorial 1.
- v1.4 Added an IIR filter task (biquad) to the MAC chapter.
- v1.5.0 Clearer problem 1.2 and better solution. Improved syntax highlighting. (OG+FS)
- v1.5.1 Corrected formulation and answer of 5.2. LaTeX, grammar and consistency improvements. (OG)
- v1.5.2 Improved formatting. (OG)
- v1.5.3 Included diagrams in solutions for the ALU exercises (FS)