Examination
Design of Embedded DSP Processors, TSEA26

Date	2018-10-24
Room	G36
Time	8–12
Course code	TSEA26
Exam code	TEN1
Course name	Design of Embedded DSP Processors
Department	ISY
Number of questions	5
Number of pages (including this page)	16
Course responsible	Oscar Gustafsson
Teacher visiting the exam room *Phone number during the exam time*	Oscar Gustafsson 013-284059
Visiting the exam room	About 9 and 11
Course administrator	Oscar Gustafsson
Permitted equipment	None, besides an English dictionary
Grading	**Points** **Swedish grade** 41–50 5 31–40 4 21–30 3 0–20 U

Important information:

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

Good luck!

Question 1: Arithmetic and Logic Unit (8 p)

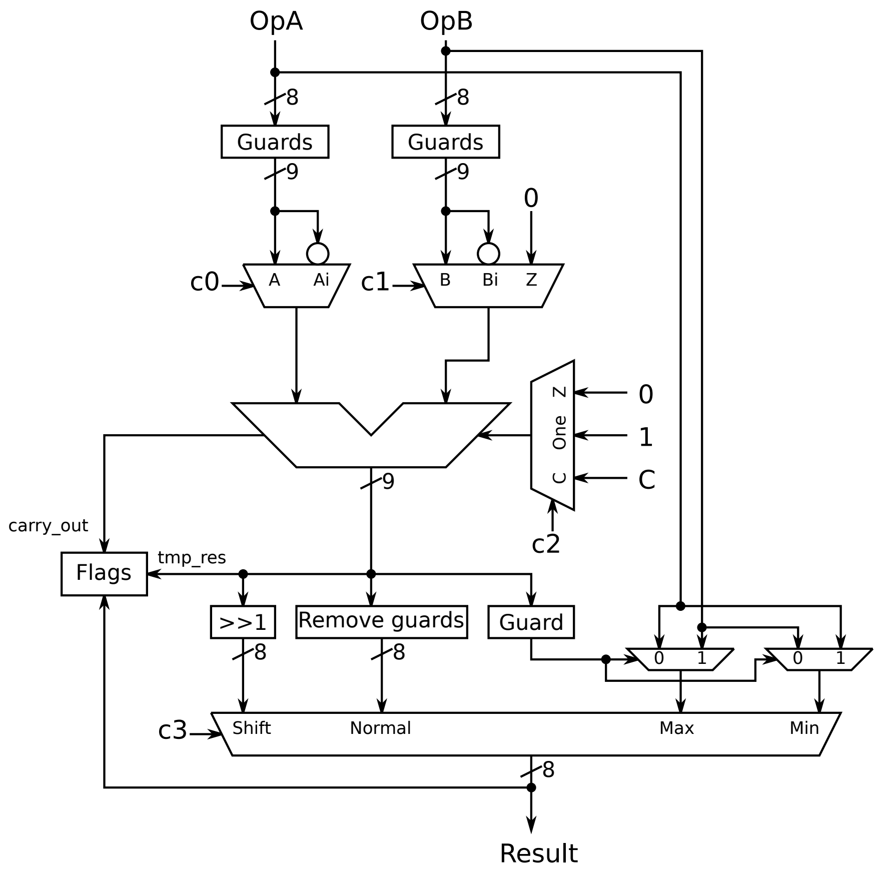
An 8-bit Arithmetic and Logic Unit should be implemented. The following functions should be supported.

Op	**Function**	**Description**
0	Nop	No operation
1	Result $\leftarrow$ OpA + OpB	Addition
2	Result $\leftarrow$ OpA − OpB	Subtraction
3	Result $\leftarrow$ OpA + OpB + C	Addition with carry flag
4	Result $\leftarrow$ OpA − OpB + C	Subtraction with carry flag
5	Result $\leftarrow \frac{\text{OpA+OpB}}{2}$	Average value
6	Result $\leftarrow$ $\|$OpA$\|$	Absolute value
7	Result $\leftarrow \max\{\text{OpA}, \text{OpB}\}$	Max value
8	Result $\leftarrow \min\{\text{OpA}, \text{OpB}\}$	Min value

In addition, there should be support for carry flag, overflow flag and sign flag, so this must also be implemented. Note that saturation is not required, but that the average value should always produce a correct result, even if the corresponding 8-bit addition happens to.

Draw a hardware architecture with control table for an ALU supporting the functionality discussed above. Denote the word length of all data signals (no need for control signals). (8 p)

Solution:



Op	Condition	c0	c1	c2	c3	cflag
0	-	-	-	-	-	0
1	-	A	B	Z	Normal	1
2	-	A	Bi	One	Normal	1
3	-	A	B	C	Normal	1
4	-	A	Bi	C	Normal	1
5	-	A	B	Z	Shift	1
6	sign(A) = +/0	A	0	Z	Normal	1
6	sign(A) = −/1	Ai	0	One	Normal	1
7	-	A	Bi	One	Max	1
8	-	A	Bi	One	Min	1

Guards

```
output(8 downto 0) <= input(7) & input(7 downto 0)
```

>> 1

```
output(7 downto 0) <= input(8 downto 1)
```

Remove guards

```
output(7 downto 0) <= input(7 downto 0)
```

Guard

```
output <= input(8)
```

Flags

```
if cflag
   carry, C <= carry_out
   overflow, V <= tmp_res(8) xor tmp_res(7)
   sign, S <= result(7)
end if
```

One may argue if the overflow flag really should be set in the case of operation 5 since the shift avoids the overflow which the flag may indicate. Note that the sign flag must be decided based on the actual result.

Question 2: Address Generation Unit (10 p)

An address generation unit (AGU) shall be implemented that can handle image data in an efficient way. For simplicity, we only consider the case of 256 × 256 pixels, i.e., 16 bits are required to address the whole memory. However, the data is either stored in a linear way or in a block way as illustrated in Fig. 1 and it should be possible to access the data in a linear or block way independent of how the data is stored. Hence, the AGU must possibly perform translations between the different formats. There is a register, `FORMAT`, which is 0 if the data is stored in a linear way in the memory and 1 if it is stored in a block way, and an input, `MODE` which determines if the 16-bit immediate or register file value shall be interpreted as linear `ROW & COLUMN`, where `ROW` and `COLUMN` are 8 bits each, or block `BLOCK & POSITION`, where `BLOCK` and `POSITION` are 8 bits each.

The following instructions shall be supported:

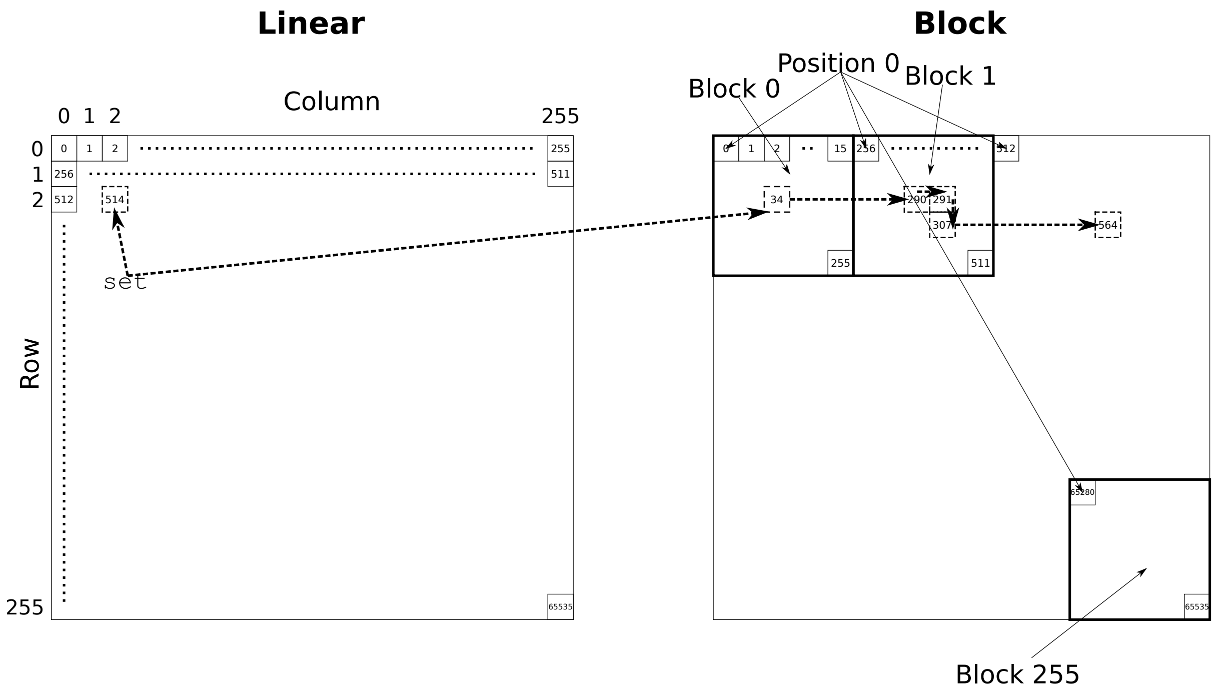


Figure 1: Two different memory mappings of 256 × 256 image in AGU problem.

Op	**Function**	**Description**
0	Reset	Clear all registers
1	Nop	No operation
2	Ar ← RF	Load address from register file, interpret depending on `MODE`
3	Ar ← Imm	Load address from immediate, interpret depending on `MODE`
4	Mem[Ar][Ar]	Access memory with Ar
5	Mem[Ar++][Ar]	Access memory with Ar, post-increment of `ROW` or `BLOCK` depending on `MODE`
6	Mem[Ar][Ar++]	Access memory with Ar, post-increment of `COLUMN` or `POSITION` depending on `MODE`
7	Mem[Ar++][Ar++]	Access memory with Ar, post-increment of `ROW` or `BLOCK` and `COLUMN` or `POSITION` depending on `MODE`

For the post-increment addressing, the value is expected to wrap-around. For example, when the end of the row is reached and `COLUMN` is increased, Ar should be set to the first value of the next row. If the end of a block is reached and `POSITION` is increased, Ar should be set to the first value of the next block.

Hint: let the internal address register be in one of the two formats and convert when required.

Example (the dashed arrows in the figure illustrates this sequence of operations):

```
; image stored in block mode, so format == 1

set ar.linear, 0x0202   ; row 2, column 2
                        ; points at address 34, which is the pixel at that position
                        ; with image stored in block mode (format == 1)
                        ; if format == 0, 514 would be the correct address

; load with post-increment of block => address = 34, next address = 290
load.block r0, mem(ar[++,])
```

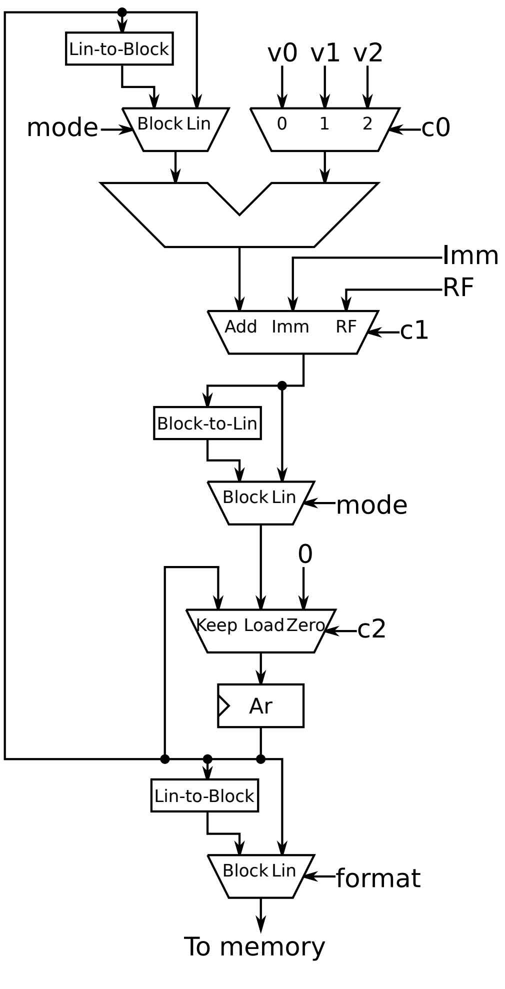
```
; load with post-increment of column => address = 290, next address = 291
load.linear r0, mem(ar[,++])

; load with post-increment of row => address = 291, next address = 317
load.linear r0, mem(ar[++,])

; load with post-increment of block and position => address = 317, next address = 564
load.block r0, mem(ar[++,++])
```

Draw a hardware architecture including control table of the Address Generation Unit meeting the functionality described above. Inputs are Op, RF, Imm, `mode`, and `format`. Output is address going to memory. (10 p)

Solution: Assume that the address is stored in linear format (arbitrarily chosen based on the hint). The architecture can then be drawn as below (in general without knowing how the conversion is performed). The constants v0 to v2 corresponds to adding one row/block, one column/position, or both, respectively.



For this particular case it is possible to implement the block to linear conversion and vice versa by simply exchanging the LSB half of `ROW/BLOCK` with the MSB half of `COLUMN/POSITION`. In VHDL:

```
output <= i(15 downto 12) & i(7 downto 4) & i(11 downto 8) & i(3 downto 0);
```

The control table looks as:

Op	**Function**	**c0**	**c1**	**c2**
0	Reset	-	-	Zero
1	Nop	-	-	Keep
2	Ar ← RF	-	RF	Load
3	Ar ← Imm	-	Imm	Load
4	Mem[Ar][Ar]	-	-	Keep
5	Mem[Ar++][Ar]	0	Add	Load
6	Mem[Ar][Ar++]	1	Add	Load
7	Mem[Ar++][Ar++]	2	Add	Load

The constants here are the same for both modes, where v0 = `0x0100`, v1 = `0x0001`, and v2 = `0x0101`.

Question 3: Program Flow Control (14 p)

Two versions of the same processor should be evaluated. The first version has a cycle time of 10 ns ($f_{\text{clk}} = 100$ MHz) and has three delay slots for both conditional and unconditional jumps and a one cycle latency for ALU operations, i.e., the result from one instruction can be used directly by the proceeding one. The second version has a cycle time of 8 ns ($f_{\text{clk}} = 125$ MHz) due to more pipelining stages. This leads to five delay slots for conditional and unconditional jumps and two cycles of latency for the ALU, i.e., the result from an instruction can not be used by the proceeding instruction, but only for the one following that. Neither of the processors introduce any NOPs, so this must be done by the programmer. The processors does not support immediates for ALU instructions, only for `set`. `set` uses the ALU to avoid structural hazards, so for the first processor the value can be used in the next instruction, while for the second processor, the next instruction can not use the value. The same holds for stack `pop`.

Assume that the structure of the first processor is as follows:

Pipeline stage	Description
P0	Next program counter value, PC++
P1	Program counter
P2	Fetched instruction
P3	Decoded instruction
P4	Register file
P5	ALU and and conditional checking
P6	Write back

For the second processor one pipeline stage is added in register file and one in the ALU.

(a) Consider the following piece of code, written for the first processor:

```
   ; Input data in r3
0:   set r0, #1
1:   set r1, #1
2:   set r2, #5
  loop:
3:   cmp r3, r1
4:   jmp.leq destination
5:   add r4, r3, r0
```

```
 6:   nop
 7:   nop
 8:   add r4, r4, r1
    destination:
 9:   add r1, r1, r1
10:   sub r2, r2, r0
11:   jmp.nz loop
12:   nop
13:   nop
14:   nop
```

Rewrite the code for the second processor and draw pipeline diagrams for the second processor when the input is `r3=1` and `r3=2`. It is enough to include the first 10 instructions that are not NOPs. (7 p)

Solution: Code for second processor (primarily five delay slots)

```
   ; Input data in r3
 0:   set r0, #1
 1:   set r1, #1
 2:   set r2, #5
    loop:
 3:   cmp r3, r1
 4:   jmp.leq destination
 5:   add r4, r3, r0
 6:   nop
 7:   nop
 8:   nop
 9:   nop
10:   add r4, r4, r1
    destination:
11:   add r1, r1, r1
10:   sub r2, r2, r0
11:   jmp.nz loop
12:   nop
13:   nop
14:   nop
15:   nop
16:   nop
```

(b) Consider the following piece of code

```
int f1(int[] a, int[] b, int[] c, int[] d, int e) {
   int sum;
   for(int i = 0; i <= 8; i++) {
      sum = a[i] + b[i] + c[i] + d[i];
      if (sum >= e) {
         return sum;
      }
   }
   return -1;
}
```

The values are on the stack and the return value should be `push`:ed to the stack, where the first argument, `a`, (the address of the array), is the first value to be `pop`:ed and so on. It can be assumed that you have enough registers both in the general register file and number of address registers to not limit the way you write your assembly

code. It is possible to pop directly to address registers. The arrays a, b, c, and d are all stored in memory which can be accessed through a load instruction with post-increment addressing. For the first processor, the result can be used in the next instruction, while for the second processor, the result can not be used for the proceeding instruction. Considering that the code is written as a sub-routine, and, hence, ret ends the program.
What is the worst case execution time of the program for the two different processors? It is enough to count the number of cycles for the instructions, including ret, to determine the time. However, there is no need to consider the delay slots for the ret, nor the pipeline latency for filling and emptying the pipeline. You should try to minimize the amount of NOPs required for both codes to get a fair comparison. Although it should be possible to easily change the loop counter if required (so do not use loop unrolling). (7 p)

Solution:

Processor 1 – initial implementation

```
   pop ar0  ; a
   pop ar1  ; b
   pop ar2  ; c
   pop ar3  ; d
   pop r0   ; e
   set r1, #1
   set r2, #8  ; as geq is used
loop:
   load r4, (ar0++)
   load r5, (ar1++)
   load r6, (ar2++)
   load r7, (ar3++)
   add r4, r4, r5
   add r4, r4, r6
   add r4, r4, r7
   cmp r4, r0
   jmp.geq end
   nop  ; delay slot 1
   nop  ; delay slot 2
   nop  ; delay slot 3
   sub r2, r2, r1
   jmp.nz loop
   nop  ; delay slot 1
   nop  ; delay slot 2
   nop  ; delay slot 3
   set r4, #-1 ; return value
end:
   push r4
   ret
```

$7 + 9 \times 17 + 3 = 163$ cycles $\Rightarrow 1.63\ \mu$s.

Processor 1 – better implementation

```
   pop ar0  ; a
   pop ar1  ; b
   pop ar2  ; c
   pop ar3  ; d
   pop r0   ; e
   set r1, #1
   set r2, #8  ; as geq is used
loop:
   load r4, (ar0++)
   load r5, (ar1++)
   load r6, (ar2++)
   load r7, (ar3++)
   add r4, r4, r5
   add r4, r4, r6
   add r4, r4, r7
   cmp r4, r0
   jmp.geq end
   nop  ; delay slot 1
   nop  ; delay slot 2
   sub r2, r2, r1 ; OK to decrease counter in delay slot
   jmp.nz loop
   nop  ; delay slot 1
   nop  ; delay slot 2
   nop  ; delay slot 3
   set r4, #-1 ; return value
end:
   push r4
   ret
```

$7 + 9 \times 16 + 3 = 154$ cycles $\Rightarrow 1.54\ \mu$s.

Processor 1 – even better implementation using software pipelining

```
   pop ar0  ; a
   pop ar1  ; b
   pop ar2  ; c
   pop ar3  ; d
   pop r0   ; e
   set r1, #1
   set r2, #7  ; as geq is used
   load r4, (ar0++) ; First iteration
   load r5, (ar1++)
   load r6, (ar2++)
   load r7, (ar3++)
   add r4, r4, r5
   add r4, r4, r6
   add r4, r4, r7
loop:
   cmp r4, r0
   jmp.geq end
   load r7, (ar3++) ; delay slot 1
   load r5, (ar1++) ; delay slot 2
   load r6, (ar2++) ; delay slot 3
   load r4, (ar0++) ; if loaded in delay slot, the return value would change
   sub r2, r2, r1
   jmp.nz loop
   add r4, r4, r5  ; delay slot 1
   add r4, r4, r6  ; delay slot 2
   add r4, r4, r7  ; delay slot 3
   set r4, #-1 ; return value
end:
   push r4
   ret
```

$14 + 8 \times 11 + 3 = 105$ cycles $\Rightarrow 1.05\ \mu$s.

Processor 2 – initial implementation

```
   pop ar0  ; a
   pop ar1  ; b
   pop ar2  ; c
   pop ar3  ; d
   pop r0   ; e
   set r1, #1
   set r2, #8  ; as geq is used
   set r3, #-1 ; return value
loop:
   load r4, (ar0++)
   load r5, (ar1++)
   load r6, (ar2++)
   load r7, (ar3++)
   add r4, r4, r5
   nop
   add r4, r4, r6
   nop
   add r4, r4, r7
   nop
   cmp r4, r0
   jmp.geq end
   nop  ; delay slot 1
   nop  ; delay slot 2
   nop  ; delay slot 3
   nop  ; delay slot 4
   nop  ; delay slot 5
   sub r2, r2, r1
   jmp.nz loop
   nop  ; delay slot 1
   nop  ; delay slot 2
   nop  ; delay slot 3
   nop  ; delay slot 4
   nop  ; delay slot 5
   push r3
   ret
end:
   push r4
   ret
```

$8 + 9 \times 24 + 2 = 226$ cycles $\Rightarrow 1.808\ \mu$s.

Processor 2 – better implementation

```
   pop ar0  ; a
   pop ar1  ; b
   pop ar2  ; c
   pop ar3  ; d
   pop r0   ; e
   set r1, #1
   set r2, #8  ; as geq is used
   set r3, #-1 ; return value, if done as in previous code
               ; an additional cycle is required for worst case
```

```
loop:
   load r4, (ar0++)
   load r5, (ar1++)
   load r6, (ar2++)
   add r4, r4, r5   ; assuming that load also requires an extra cycle
   load r7, (ar3++) ; not clearly stated in problem
   add r4, r4, r6
   nop
   add r4, r4, r7
   nop
   cmp r4, r0
   jmp.geq end
   nop  ; delay slot 1
   nop  ; delay slot 2
   nop  ; delay slot 3
   nop  ; delay slot 4
   sub r2, r2, r1 ; delay slot 5
   jmp.nz loop
   nop  ; delay slot 1
   nop  ; delay slot 2
   nop  ; delay slot 3
   nop  ; delay slot 4
   nop  ; delay slot 5
   push r3
   ret
end:
   push r4
   ret
```

$8 + 9 \times 18 + 2 = 172$ cycles $\Rightarrow 1.376\ \mu$s.

Processor 2 – even better implementation using software pipelining

```
   pop ar0  ; a
   pop ar1  ; b
   pop ar2  ; c
   pop ar3  ; d
   pop r0   ; e
   set r1, #1
   set r2, #7  ; as geq is used
   set r3, #-1 ; return value
   load r4, (ar0++)
   load r5, (ar1++)
   load r6, (ar2++)
   add r4, r4, r5   ; assuming that load also requires an extra cycle
   load r7, (ar3++) ; not clearly stated in problem
   add r4, r4, r6
   nop
   add r4, r4, r7
   nop
loop:
   cmp r4, r0 ; Use r5 instead of r4 to not have to wait for the move
   jmp.geq end
   load r8, (ar0++) ; use different register to not overwrite return value
   load r5, (ar1++) ; delay slot 2
   load r6, (ar2++) ; delay slot 3
   load r7, (ar0++) ; delay slot 4
   sub r2, r2, r1   ; delay slot 5
   jmp.nz loop
   add r4, r8, r5   ; delay slot 1
   add r6, r6, r7   ; addition tree
   nop              ; delay slot 3
   add r4, r6, r6   ; delay slot 4
   nop              ; delay slot 5
   push r3
   ret
end:
   push r4
   ret
```

$17 + 8 \times 13 + 2 = 123$ cycles $\Rightarrow 0.984$ μs.

Question 4: Miscellaneous Questions (5 p)

(a) Discuss one advantage and one disadvantage of using memory hierarchies. For example, both a larger external memory and a smaller internal memory. (2 p)

Solution: Smaller memories are faster and consume less power. However, using a memory hierarchy requires moving data between the memories making the implementation more complicated.

(b) When implementing a `set` instruction, it is possible to write the immediate value to the register file directly after the instruction is decoded. However, it is common that the immediate value is passed through the ALU. What is the potential benefit of this? (1 p)

Solution: In this way the structural hazards are decreased as the write otherwise would happen in a different pipeline stage. This may for example lead to that an ALU instruction cannot be performed directly after a set. Explicitly using the ALU, compared to have a dedicated set path with the same pipeline depth, reduces the hardware complexity as signals do not have to be routed to/from the additional unit. In addition, writing in an earlier pipeline stage may lead to write-before-read data hazards.

(c) Why do we typically need two different versions of the simulator? Which are those two versions and what are their respective advantage. (2 p)

Solution: A behavioral simulator focus on high-level aspects such as the operation and result of instructions. Primarily used for software development and should be fast.

A micro architecture provides a more detailed description of the pipeline and building blocks and will therefore be slower. Primarily used for verification of HDL code.

Question 5: Multiply-Accumulate Unit (13 p)

A multiply-accumulate (MAC) unit shall be implemented using a single multiplier. To reduce the switching activity, it is possible to interleave two computations of an FIR filter. This also allows one memory to be shared between data and coefficients. The following pseudo-code illustrates this:

```
void executetwofilters(int16 dataptr, int16 coeffptr, int16 filterlength) {
  int40 acc0 = 0, acc1 = 0;
  for (i = 0; i <= filterlength-1; i++) {
    acc0 +=  mem0[dataptr++]*mem0[coeffptr];
    acc1 +=  mem0[dataptr]*mem0[coeffptr++];
  }
  int16 result0 = (int16) acc0; // Fixed-point correct casting required,
  int16 result1 = (int16) acc1; // with saturation and rounding
  return result0, result1; // Not valid C, but you get the idea
}
```

(a) Define a minimal set of suitable CONV-like instruction that can implement this function, plus required instructions to start a convolution and obtain the result. To avoid multiple reads, the instruction must get the old value from a general register, and should store the new value in the same register. In addition, we would like to keep the same data at the same multiplier input to avoid unnecessary switching. (2 p)

Solution: We need two different CONV-instructions depending on if the first or second argument is taken from memory. In addition, we need a clear instruction (or a MUL with memory access to replace the first CONV) and a MOVE instruction to get the value from the accumulator register.

Instruction	**Function**

`CONVMOVE0 acrX, (arY++), rZ`	tmp ← mem0[arY]
	acrX ← acrX + tmp × rZ
	rZ ← tmp
	arY ← arY + 1

`CONVMOVE1 acrX, (arY++), rZ`	tmp ← mem0[arY]
	acrX ← acrX + rZ × tmp
	rZ ← tmp
	arY ← arY + 1

`CLEAR acrX`	acrX ← 0
`MOVE rX, acrY`	rX ← sat(round(acrY))

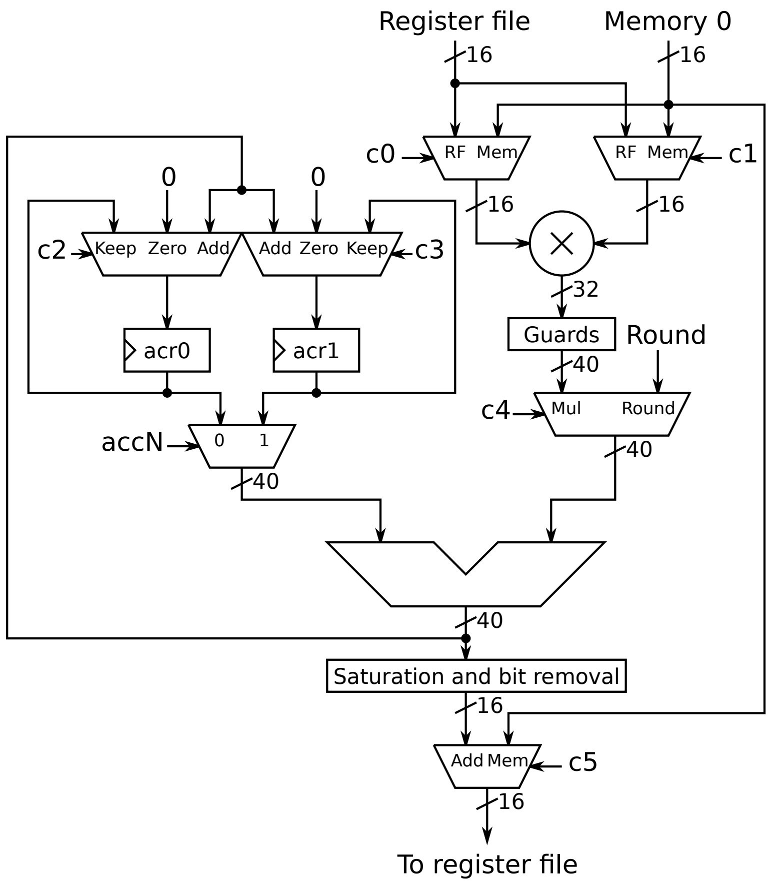
(b) Write assembly code for the program. Assume that there is a `repeat` instruction. `dataptr` and `coeffptr` are available on the stack (for the exam problem, the order does not matter as it can be easily changed). However, a `pop` require an additional cycle before the result is available. `filterlength` is a constant (so it can be used directly as a variable). The results should be put back on the stack. (3 p)

Solution:

```
   pop ar0 ; assume dataptr
   pop ar1
   clear acr0   ; clear here to allow ar1 time to be set
   load0 r0, (ar1) ; load first coefficient
   clear acr1
   repeat endoffilter, filterlength
   convmove0 acr0, (ar0++), r0
   convmove1 acr1, (ar1++), r0
endoffilter:
   move r0, acr0
   move r1, acr1
   push r0
push r1
```

(c) Draw a hardware architecture with control table (when applicable) of the MAC-unit supporting the instructions from the first part. The result should always be saturated and rounded at readout (in the correct order). Assume that fractional data is used, the word length for both data and coefficients are 16 bits and that the accumulator has 8 additional bits, apart from the 32 from the multiplier. Each input comes either from the register file, `rf`, or memory 0, `mem0`. There is a single output to the register file used both for the intermediate values and for the result. In addition, there is an input `accN` with the number of the accumulator register to use. Assume that a signed fractional representation is used. (6 p)

Solution:



Op	Condition	c0	c1	c2	c3	c4	c5
CONVMOVE0	accN = 0	Mem	RF	Add	Keep	Mul	Mem
CONVMOVE0	accN = 1	Mem	RF	Keep	Add	Mul	Mem
CONVMOVE1	accN = 0	RF	Mem	Add	Keep	Mul	Mem
CONVMOVE1	accN = 1	RF	Mem	Keep	Add	Mul	Mem
CLEAR	accN = 0	-	-	Zero	Keep	-	-
CLEAR	accN = 0	-	-	Keep	Zero	-	-
MOVE	-	-	-	Keep	Keep	Round	Add
NOP	-	-	-	Keep	Keep	-	-

The result in the accumulator has 10 integer bits and 30 fractional bits (as the result from the multiplier has two integer bits and 30 fractional bits).

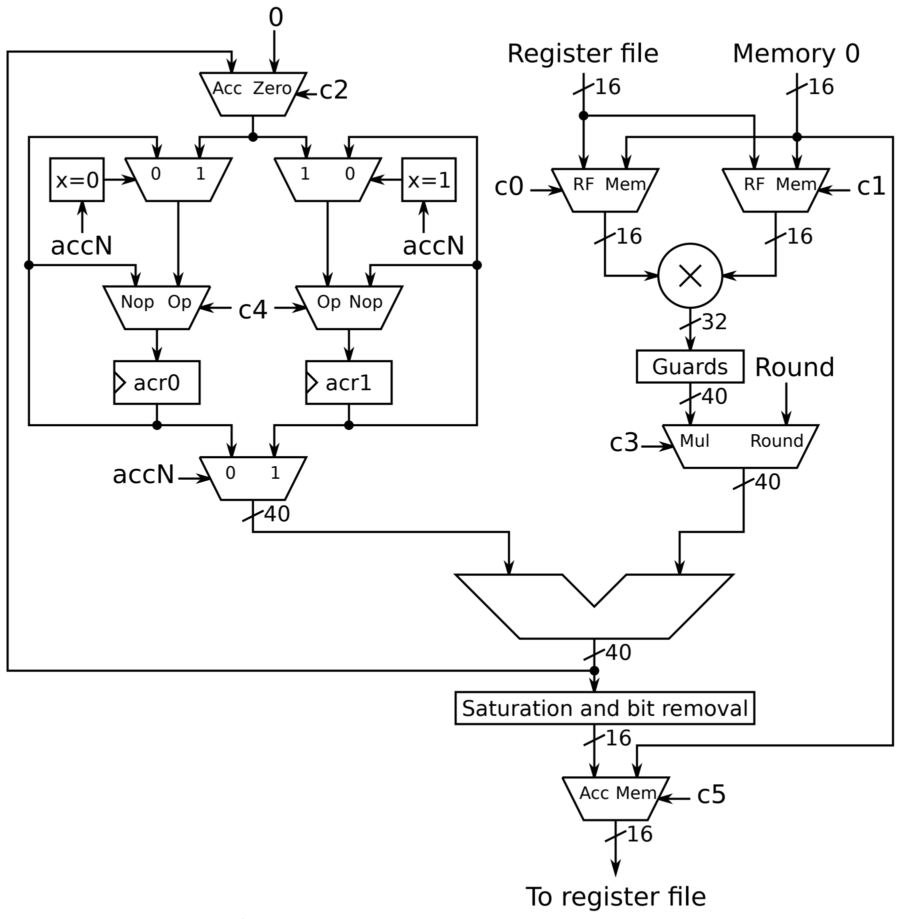
Rounding vector is a one in the first position that is quantized away from the LSB side, i.e., position 14.

```
Round = 0x0000004000
```

Saturation and bit removal

```
if input(39 downto 30) = "0000000000" or input(39 downto 30) = "1111111111"
    output <= input(30 downto 15)
else
    if input(39) = '1'
        output <= 0x8000  -- Negative saturation
    else
        output <= 0x7FFF  -- Positive saturation
    end if
end if
```

Alternative architecture (Round and Saturation and bit removal identical)



Op	c0	c1	c2	c3	c4	c5
CONVMOVE0	Mem	RF	Acc	Mul	Op	Mem
CONVMOVE1	RF	Mem	Acc	Mul	Op	Mem
CLEAR	-	-	Zero	-	Op	-
MOVE	-	-	Keep	Round	Op	Acc
NOP	-	-	Keep	-	Nop	-

(d) Discuss how modulo addressing and cyclic buffers will affect the computation using this type of approach. For example, how long will the buffer be here? (2 p)

Solution: The sample buffer should be one position longer. Also, when writing the next sample, either it is written with pre-increment or the `dataptr` is reset to the correct position, or the last iteration is outside of the repeat such that the final operation increments `dataptr` instead of `coeffptr`. The reason is that as opposed to the normal single computation approach, the last position in the data buffer is not accessed a second time and therefore not incremented. Apart from that, modulo addressing should work just as normal.