FEUP
Universidade do Porto
Faculdade de Engenharia

Departamento de Engenharia Electrotécnica e de Computadores

Digital Systems Design 2016/2017

4º year, 1º semester
Exam – 17th January 2017

Maximum duration: 3h00m, closed book.

[1.5 points]

1 - Explain how to assess the quality of a behavioral simulation process of a RTL model and describe how did you exploit that during the functional verification stage in the development of the last laboratory project (and if you did not make use of that, explain how could you have done that).

The quality of a behavioral simulation process can be assessed by the code coverage metrics that provide statistics about the percentage of statements (of the RTL Verilog code) that are simulated, the percentage of cases of conditional expressions actually simulated, the number of states and state transitions activated during the simulation of finite state machines and also the percentage of bits that did change state during the simulation (toggle coverage).

The second part of the question does not have a unique answer. Most of the groups (eventually all of them) did not worry about the code coverage during the functional simulation and just did use the testbench provided. A valid answer for those could be "we did not make use of the code coverage metrics [note that the argument: "nobody told us to do that…" cannot be accepted as an excuse!], but we should have done to know how far should the simulation run. When all the metrics referred above reach the mark of 100%, it is (usually) not worth to continue the simulation.

[2.5 points]

2 - During the design of a digital system, various verification processes must be performed at different design stages. Two important verification stages are the functional verification, by logic simulation, of the system under design before (behavioral simulation) and after (post-synthesis simulation) performing the RTL synthesis.

[1 point]

a) Explain the differences between the models that are simulated in these two stages.

The functional verification performed before the RTL synthesis process (referred to as "behavioral simulation" in the XILINX ISE design environment) simulates the logic behavior of the Verilog source code written by the designer (usually called the "RTL code"). The functional verification done after the RTL synthesis (or "post-translate simulation") simulates the logic circuit (or netlist) built by the synthesis process, that is formed by the logic elements available in the target technology (logic gates, flip-flops, memory blocks, lookup-tables, etc).

[1.5 points]

b) If the post-synthesis simulation permits to verify the correctness of the circuit under design, explain why is it fundamental to do first the behavioral simulation, in the perspective of the verification and debug procedure (do not consider the fact that the post-synthesis simulation is much slower than the behavioral simulation).

In the netlist model being simulated, obtained by the RTL synthesis process, there is no clear relationship between the signals (wires, buses, registers) and the statements in the Verilog source code. If a functional error is found only during the post-synthesis simulation stage, it is thus very difficult or even impossible to track the error back to the source code to correct it.

[2.5 points]

3 - The construction of a reset signal is essential to define a coherent initial state in all registers of a synchronous digital system. If the reset signal is used directly from an external input (see example below) and if it is activated asynchronously with the clock signal (e.g. by pressing manually a push-button), this may lead to a wrong initialization of the registers controlled by this signal.

```
module toplevel( input reset, input clock, ... );
...
always @(posedge clock or posedge reset)
if (reset)
```

```
// initialization actions
else
// non-reset behavior
```

[1 point]

a) Explain why this wrong behavior may occur.

If the reset signal is not synchronous with the clock, its transitions (0->1 and 1->0) may occur at any instant. If the de-activation of the reset (transition 1->0) happens close enough to the active clock transition, and because of the differences of the propagation time from the reset input to each flip-flop, some flip-flops may see the reset being released before the clock, others after the clock and others may also be set temporarily into a meta-stable state and then jump randomly to state 1 or 0. Note that how much time is "close enough" depends on the actual timing parameters of the flip-flops.

[1.5 points]

b) Explain how to solve this problem, assuming a clocked synchronous system with a single clock domain and an asynchronous reset signal, as shown in the example above.

The solution is to synchronize the reset signal to the system clock using a basic two-stage shift-register. This way, the transitions of the synchronized reset will always occur after the active clock edge.

[4 points]

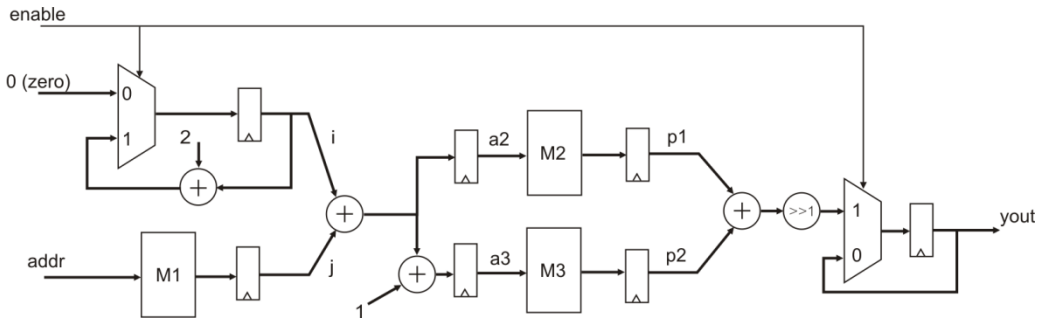
4 - In the model represented by the Verilog code below, the memories M2 e M3 contain the same set of data:

```
// signals 'addr[5:0]' and 'enable' are inputs
reg [ 7:0] M1[0:63];
reg [19:0] M2[0:255], M3[0:255];
reg [19:0] p1, p2;
reg [ 7:0] i, j, a2, a3;
reg [19:0] yout;

always @(posedge clock)
if (reset)
  // initialize all register with zeros
else
begin
  yout <= enable ? ( p1 + p2 ) >> 1 : yout;
  a2   <= j + i;
  a3   <= j + i + 1;
  p1   <= M2[ a2 ];
  p2   <= M3[ a3 ];
  i    <= enable ? (i + 2) : 8'd0;
  j    <= M1[ addr ];
end
```

[2 points]

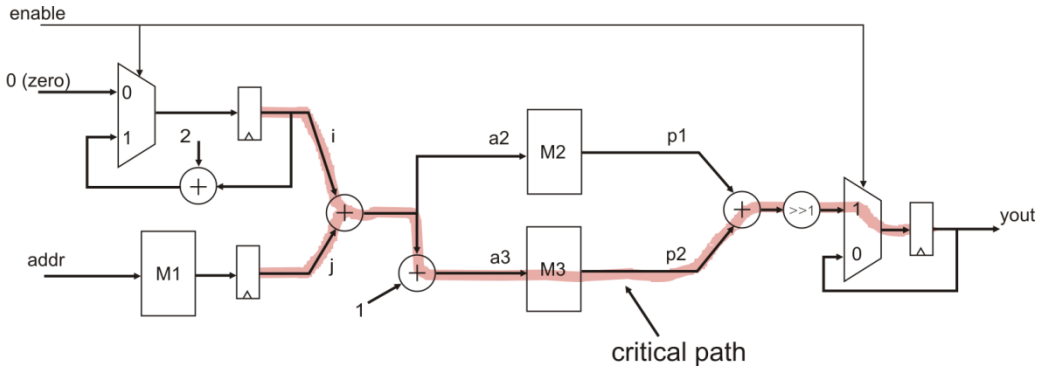
a) Draw a block diagram representing the circuit described by the Verilog code above. Assume that M1[ ], M2[ ] and M3[ ] are implemented by read-only memories with asynchronous (combinational) reads. To simplify your diagram do not represent the structures that implement the reset action of the registers.



[2 points]

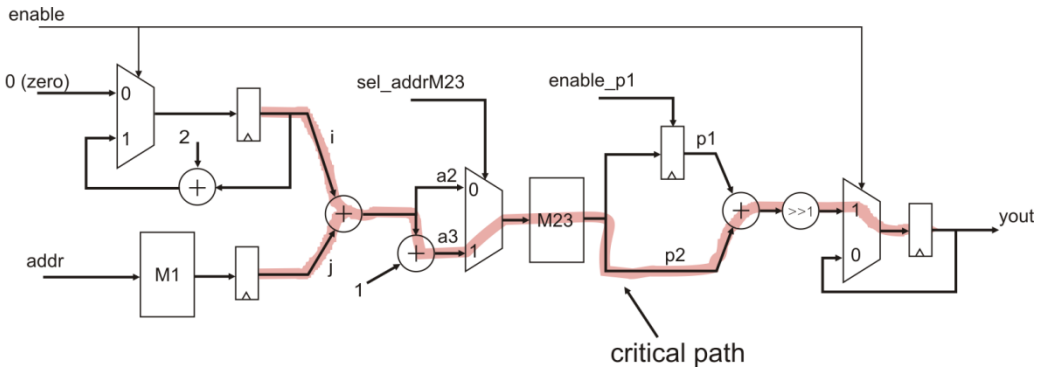
b) During a meeting of the design team, Mr. X said that it would be possible to redraw this function using less area and performing faster because the clock period to apply to the circuit is approximately 10X longer than the propagation delay of a 32 bit adder and 8X longer than the read propagation time of the memories M1, M2 and M3. Explain how to exploit this optimization and which implications this may have in the rest of the system that will use this block.

*First, as one of the students said, the original circuit is "too pipelined" for a clock with a period roughly 8 times longer than the propagation delay of the memories, which will be the most constraining block for the clock period. Thus, we can remove registers **a2**, **a3**, **p1** and **p2** and make the path from registers **i** and **j** to register **yout** fully combinational:*



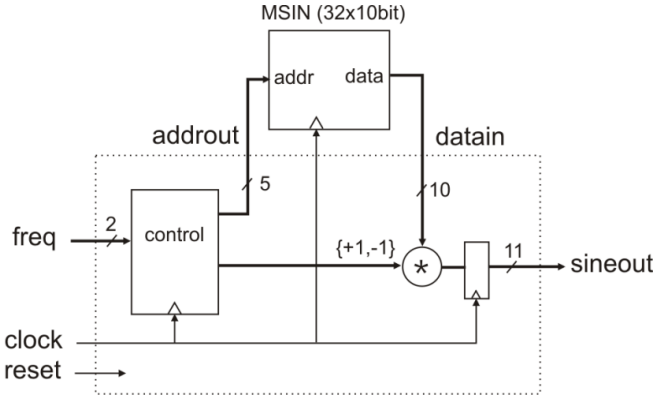
*Note that this combinational circuit will have in its timing critical path three adders (two 8-bit wide and one 20-bit wide) and one memory and will thus fit the clock period. This will make the circuit smaller (less registers) but the output **yout** will be updated 2 clock cycles before the original circuit, what must be taken into account by the rest of the system where this block is used.*

*Although the answer above is considered as correct, an additional optimization (also considered as correct) can be done by exploiting the fact that the memories **M2** and **M3** have the same data. A possible solution is shown in the diagram below, using a single memory called **M23**. This requires using a clock two times faster than the original, to allow performing two reads from memory **M23** in the same time period used before. In the first clock the address mux is set to **a2** and register **p1** is enabled and in the second clock cycle the address mux is set to **a3** and the output register is enabled.*

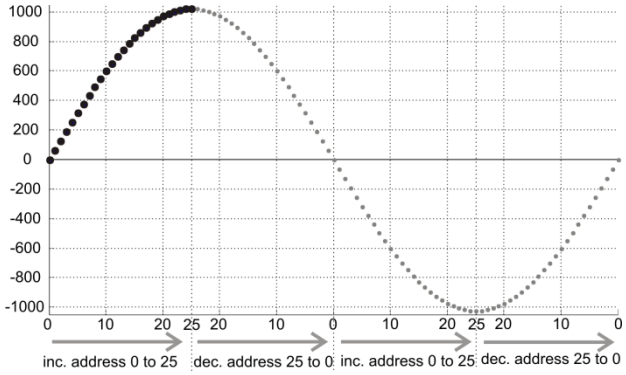


[4 points]

5 - The diagram below represents a digital sine wave generator. The output is a sequence of 11 bit signed values (two's complement) representing samples of a sine wave with frequencies selectable among 100 kHz, 50 kHz, 25 kHz and 12.5 kHz and sampling frequency equal to the main clock frequency (10 MHz).



A read-only memory external to the circuit under design (**MSIN**) contains a table with 26 samples representing the values of *sin(x)* in the first quadrant multiplied by 1023 (**x** values in the interval $[0,\pi/2]$, dark points in the figure below). By exploiting the symmetry of the function *sin(x)* and using an appropriate addressing to the memory **MSIN**, the samples of the function *sin(x)* can be generated for its whole domain. To generate a 100 kHz sine wave with a sample per clock period, the sequence of addresses to apply to **MSIN** should be 0, 1, 2, ..., 24, 25, 24,..., 2, 1, 0, 1, 2, ..., changing the sign of the value read from the memory during the 3rd and 4th sweep of the address space (3rd and 4th quadrants).



Using a 10 MHz clock signal and incrementing/decrementing the memory addresses by 1, a 100 KHz sine wave will be generated; if the address is incremented/decremented with a step of 0.5 (1 unit at each 2 clock cycles) the output frequency will be 50 kHz (similarly, a frequency of 25 kHz will be obtained with a step of 0.25 and 12.5 KHz with a step of 0.125).

Design a Verilog module that implements this system, following the coding guidelines and constraints for building synthezisable systems and adopting as the optimization criteria the minimization of the circuit area. The memory **MSIN** will be external to this circuit and contain the 26 samples of the function *sin(x)*.The read access of this memory is synchronous with the clock (the data read from the memory is presented in the data bus in the active clock transition). The address to the memory is the output **addrout** and the data read from the memory is the input **datain**. The reset signal should be synchronous with the clock and active with the logic high. The input **freq** defines the frequency of the output sine wave, according to the following table:

input **freq**	2'b00	2'b01	2'b10	2'b11
sine wave frequency	100 kHz	50 kHz	25 kHz	12.5 kHz

Naturally the solution presented below is not unique. The three main behaviors to implement are:

1. *Incrementing and decrementing the memory address to output the sine(x) samples in the correct order*
2. *Incrementing/decrementing the memory address each clock cycle, each 2 clock cycles, 3 or 4, according to the value of input **freq***
3. *Correct the sign of the data output from the memory for the 3rd and 4th quadrants*

***Comments in the code are necessary to help understand it** !*

```
module sinegen( input clock,
                input reset,
                input  [ 1:0] freq,
                output [ 4:0] addrout,
                input  [ 9:0] datain,
                output [10:0] sineout
              )

// Register addrfrac represents the memory address
// in fixed-point, with 3 bits for the fractional part
// and 5 bits for the integer part.
// We will increment/decrement this register with
// a step of 1 (1000 in binary), 0.5 (0100), 0.25 (0010)
// or 0.125 (0001), according to the value of input freq
//
// The address to apply to memory MSIN will be only the
// integer part of this register ( addrfrac[7:3] )
reg [7:0] addrfrac;
```

```
// First implement a decoder to translate the frequency specifier
// (input freq) into the 4-bit step (signal freqstep)
reg [3:0] freqstep;
always @*
case( freq )
   2'b00: freqstep = 4'b1000; // value 1.000
   2'b01: freqstep = 4'b0100; // value 0.500
   2'b10: freqstep = 4'b0010; // value 0.250
   2'b11: freqstep = 4'b0001; // value 0.125
endcase

// We need a four state FSM to select between each
// of the four quadrants, always incrementing
// or decrementing addrfreq by freqstep:
// state = 2'b00: increment address, positive output (Q1)
// state = 2'b01: decrement address, positive output (Q2)
// state = 2'b10: increment address, negative output (Q3)
// state = 2'b11: decrement address, negative output (Q4)
reg [1:0] state;
always @(posedge clock)
if ( reset )
begin
  state <= 2'b00;
  addrfrac <= 8'd0;
end
else
begin
  case ( state )
    2'b00: // First quadrant: inc. addr to 25
           if ( addrfrac[7:3] != 25 )
             addrfrac <= addrfrac + freqstep;
           else
           begin
             // When current address is 25,
             // next address will be 25-freqstep
             addrfrac <= addrfrac - freqstep;
             state <= 2'b01;
           end

    2'b01: // Second quadrant: dec. addr to 0
           if ( addrfrac[7:3] != 0 )
             addrfrac <= addrfrac - freqstep;
           else
           begin
             addrfrac <= addrfrac + freqstep;
             state <= 2'b10;
           end
    2'b10: // Third quadrant: inc. addr to 25
           if ( addrfrac[7:3] != 25 )
             addrfrac <= addrfrac + freqstep;
           else
           begin
             addrfrac <= addrfrac - freqstep;
             state <= 2'b11;
           end
    2'b11: // Fourth quadrant: dec. addr to 0
           if ( addrfrac[7:3] != 0 )
             addrfrac <= addrfrac - freqstep;
           else
           begin
             addrfrac <= addrfrac + freqstep;
             state <= 2'b00;
           end
  endcase
end


// Continue in next page
```

```
// the output memory address is the integer part
// of addrfrac:
assign addrout = addrfrac[7:3];

// Correct the sign of the output data
// if state is 2'b10 or 2'b11 (quadrants 3 or 4)
// change the sign of memory output:
assign sineout = ( state[1] ) ? -datain : datain;

endmodule;
```

Another solution, based on the solution presented by one student!

```
module sinegen( input clock,
                input reset,
                input  [ 1:0] freq,
                output [ 4:0] addrout,
                input  [ 9:0] datain,
                output [10:0] sineout
              )

// register 'direction' determines the direction
// of the address counter: 0: increase addr, 1 decrease addr
// register 'signal' encodes the sign of the output
reg direction, sign;

// registers 'limit' and 'count' implement the clock divider
// by 1, 2, 4 or 8 for generating the sine frequencies
// defined by the input 'freq'
reg [2:0] limit, count;


always @(posedge clock)
if ( reset )
begin
  addrout   <= d'd0;
  direction <= 1'd0;
  sign      <= 1'd0;
  limit     <= 3'd0;
  count     <= 3'd0;
end
else
begin
  case (freq)
    2'b00: limit <= 3'b000; // 0
    2'b01: limit <= 3'b001; // 1
    2'b10: limit <= 3'b011; // 3
    2'b11: limit <= 3'b111; // 7
  endcase

  // divides the clock to the required rate
  // of inc. or dec. the memory address
  if ( count < limit )
    count <= count + 3'd1;
  else // count == limit (0, 1, 3, 7)
    count <= 3'd0;

  // this condition count == 3'd0 is true for
  // each clock cycle (freq=00)
  // each two clock cycles (freq=01),
  // four clocks (freq=10) or eight clocks (freq=11)
  if ( count == 3'd0 )
  begin

    // Inc./dec. the memory address based on 'direction'
    addrout <= direction ? (addrout - 5'd1) : (addrout + 5'd1);

    // when 'addrout' equals 24, it is still inc. to 25 and
    // direction changed to 1 (to start decrementing)
```

```
    if ( addrout == 5'd24 )
      direction <= 1'd1;

    // when decrementing and address equals 1,
    // change direction to increment and toggle the output sign
    // note that 'addrout' is still dec. to zero
    if ( addrout == 5'd1 && direction == 1'd1 )
    begin
      direction <= 1'd0;
      sign <= ~sign;
    end

  end
end

// output the sine samples, change sign in
// the 3rd and 4th quadrants
assign sineout = sign ? -datain : datain;

endmodule;
```

[4.5 points]

6 - In the design of a video processing system it is required to build a block to calculate in real-time a 2D convolution (5x5 pixel window) on a digital video signal. A video frame is received line by line (starting from the top of the frame) and within each line the pixels are received from left to right at the rate of the pixel clock (112 MHz). Two additional synchronization signals, not relevant for the system under design, indicate the end of line (**HSYNC**) and end of frame (**VSYNC**). A minimum of 256 clock cycles separate the end of one frame and the reception of the first pixel of the next frame.

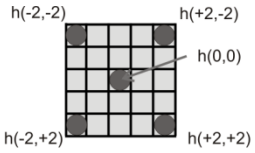
The 2D convolution between a digital image represented by the matrix *Pixelin(x,y)* and the impulse response (or *kernel*) with 5x5 coefficients, *h(i,j)*, can be expressed by the following equation, where *Pixelout(x,y)* represents the output pixel computed at coordinates *(x,y)*:

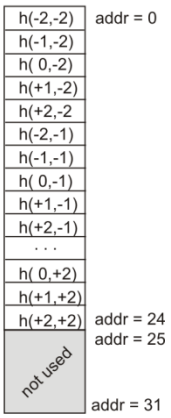
$$Pixelout(x,y) = \sum_{i=-2}^{+2} \sum_{j=-2}^{+2} Pixelin(x+i, y+j) \times h(i,j)$$

The system to implement uses a dedicated memory structure based on shift-registers that allows the parallel access to the 25 neighbor pixels *Pixelin(x+i, y+j)* of the pixel being computed, *Pixelout(x,y)*. At each pixel clock cycle a new pixel arrives and a new set of 25 neighbor pixels are presented to calculate a new output pixel. Both the pixel in and pixel out are 8 bit unsigned values

The 2D impulse response or *kernel h(i,j)* is formed by 25 fixed point signed values, with 3 bits representing the integer part and 7 bits the fractional part. These values are stored in memory **Mkernel (**32 x 10 bits, organized as shown in the figure below). This memory has an asynchronous read and can be written by an external control circuit during the 256 pixel clock interval between each two frames.

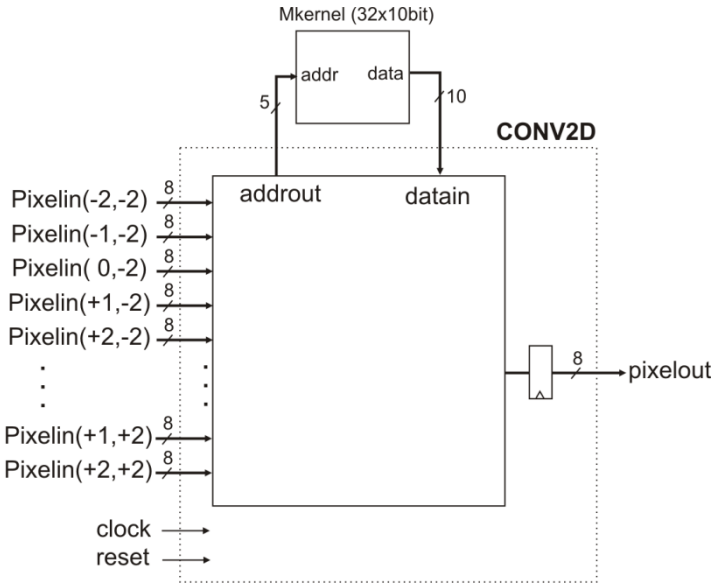
You can assume that the values *h(i,j)* guarantee that the result of each pixel will fit the unsigned 8-bit range (positive and less than 256).





It is known that, in the technology to be used for this project, a 8 x 10 bit multiplier (18 bit result) will have a propagation delay equal to 4 ns and a 20 bit adder will have a delay equal to 1.5 ns (this are approximate values).

Figure on the right shows a simplified block diagram of the system, where block **CONV2D** is the module to design. The inputs **Pixelin(,)** are updated at each pixel clock and contain the 25 neighbor pixels in the 5x5 window around the pixel to compute. These are the values that must be multiplied by the coefficients *h(i,j)* and summed to produce the output pixel.



[1.5 points]

a) Show how to build an implementation for module **CONV2D** considering a clock signal with frequency equal to the pixel clock (112 MHz) and minimizing the latency of the circuit. Justify the proposed solution using block diagrams or sections of Verilog code to help understand your solution.

*The circuit has to compute the expression above for each clock cycle, when a new set of 25 Pixelin samples are presented at the inputs. The computation to perform is an inner product between the 25 Pixelin inputs and the 25 coefficients stored in memory **Mkernel**.*

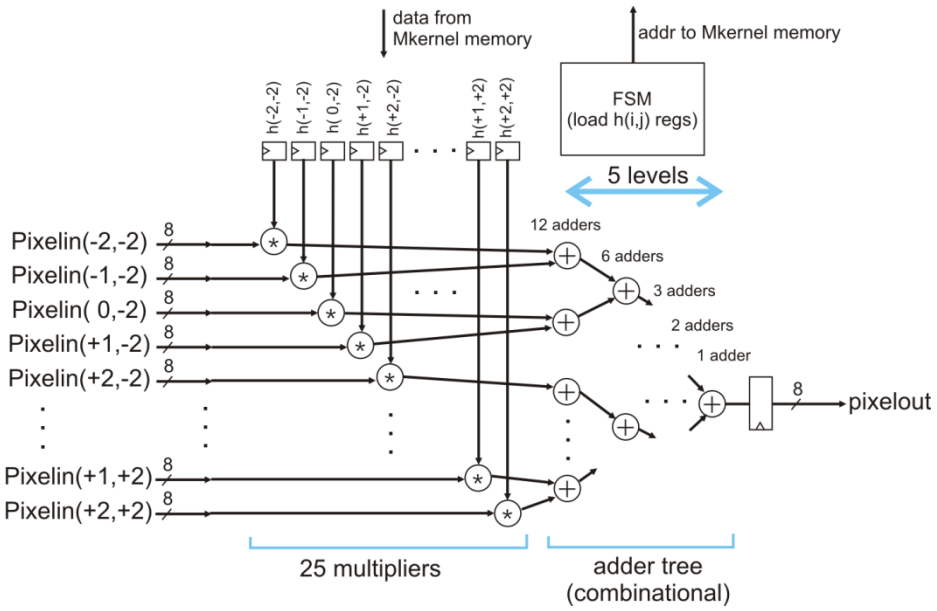
First conclusion: if we receive a new set of input data in each clock cycle (assuming the main clock is equal to the pixel clock), the circuit must be either fully combinational or pipelined to guarantee an output throughput equal to input data rate, as we have only 1 system clock per input data set (and consequently for each output result).

*Second conclusion: to be able to do that, we need to access all the 25 values stored in memory **Mkernel** in a single clock cycle. Although the memory read access is said to be asynchronous, it will be impossible to read all this data in a single clock cycle.*

*The solution is to use the interval of 256 clock cycles between the arrival of the images to pre-load the **MKernel** contents into registers local to our circuit. As it is said that this memory can only be written during this interval, and even assuming that each write cycle will consume a few clock cycles, there is plenty of time to read those 25 values into local registers. To synchronize the reading of this data, our system can make use of the signal **VSYNC** (which in fact is relevant for this solution).*

*Having said this, we have the 25 pixel data that can be accessed in parallel (inputs **Pixelin**...) , the 25 coefficients **h(i,j)** already stored into local registers and we must do 25 multiplications (in parallel) and the summation of all the results to obtain the output pixel.*

*We have a clock cycle of 8.9 ns and we know to the multiplier has a propagation delay equal to 4 ns and each 2-operand adder has a propagation delay of 1.5 ns. Thus, the array of 25 multipliers will add a propagation delay of 4 ns (as all the multipliers will work in parallel) and the adder tree required to sum the 25 products will have 5 levels of adders (see figure below). If the adders are implemented as ripple-carry adders, we know that each adder level will add a propagation delay equal to two full-adders (if the output width is increased by one bit at each level. In this case, we can estimate the total propagation time as 4 ns (the mult) + 1.5ns (the last adder) + 4 * 2 * 1.5ns / 20 (two full adders per tree level) = 6.1 ns. This number tells that it may be possible to implement the whole operation as fully combinational within the required clock cycle. However, if this solution does not meet the timing constraint, the solution is to use a pipelined version with a single register stage at the output of the multipliers or at the output of the first stage of adders.*



[1.5 points]

b) With the aim of decreasing the circuit area, it has been proposed to increase the clock frequency to 2X or 3X the pixel clock (224 MHz or 336 MHz, with periods equal to 4.46 ns and 2.98 ns, respectively). Show how to make use of a faster clock to reduce the circuit area and present an approximate number for the percentage of area reduction that may be achieved with your solution (ignore the issues related to the need of synchronization between the two clock domains).

If we have two or three clock cycles between the arrival of each new set of data we can build a smaller datapath and process roughly half of the input data in each clock cycle (for 224 MHz) or 1/3 of the input data per clock cycle.

For 224 MHz we can use only 13 multipliers (instead of 25) and perform 12 mults in one clock cycle and 13 mults in the other clock cycle. The adder three now needs to add 13 results (instead of 25) and it is necessary one accumulator to add the results of each partial summation. This solution will reduce the circuit area to approximately 50%, considering only the area used by the arithmetic operators. As the adder tree will need to be pipelined to meet the clock period constraint, the additional registers will contribute to increase the area. It is not possible to quantify the final area reduction.

Using a clock 3 times faster than the pixel clock, a similar reasoning leads to a circuit using only 1/3 of the multipliers and adders and an area reduction to roughly 33% of the original. However, as the multiplier has a propagation delay of 4ns it will not fit the clock period of the 336 MHz (2.97ns). Because of this, the multipliers for this solution must be implemented as pipelined units (as well as the adder tree) and these registers will also increase the circuit area.

[1.5 points]

c) This same system can be used to calculate 2D convolutions using smaller 3x3 kernels, instead of the original 5x5 (it is enough to set to zero, or consider as zeros, the 16 coefficients in the outside border of the 5x5 matrix – coefficients *h(-2,...), h(+2,...), h(...,-2) e h(...,+2)*. Consider that the control system that loads the **Mkernel** memory provides to your circuit one bit to select between a 5x5 kernel or a 3x3 kernel. Explain how to use that information to modify your system in order to reduce the dynamic power consumption when a 3x3 kernel is selected. Present an estimate of the decrease in the dynamic power consumption of the block **CONV2D**.

The key to reduce the dynamic power consumption in this scenario is to block the propagation of the 16 input pixel data that will be multiplied by coefficients equal to zero. This can be done either by using transparent latches in those 16 inputs that will freeze (or latch) their output when the 3x3 kernel is activated or any other structure (a mux, for example) that will feed a constant input to the 16 multipliers by zero. The reduction of the dynamic power consumption is more difficult to estimate, as it will depend on the actual organization of the adder tree. Basically, if the inputs of one circuit do not switch its dynamic power consumption is zero.

In this example we will have 16 out of 25 (64%) multipliers in such situation, whose outputs will permanently be equal to zero. If these 16 outputs are grouped together into a sub-adder tree, all the adders in that part of the tree will also have no switching activity. In such case, we can estimate that approximately 64% of the arithmetic operators will have no switching activity and thus zero dynamic power consumption.

..:: the end ::..