Assignment 3 Solution

This document is an illustration of how one could arrive at the solution. The focus is on the process of design, rather than the design itself. The approach followed here is to first look at the computational aspect of the system to be designed by describing the specified functionality in a program like manner. Then this program is restructured to make it closer to hardware implementation. The next step is to construct an ASM chart for this program. Finally that ASM chart is converted into a VHDL program in a straight forward manner.

Image filtering operation, as defined for this assignment, can be written in the form of a program as shown below. Here the arrays X (size = m x n), Y (size = m-2 x n-2) and C (size = 3 x 3) denote unfiltered image, filtered image and filter coefficients, respectively.

```
for (I = 1; I < m - 1; I++)
   for (J = 1; J < n - 1; J++) {
      sum = 0;
      for (i = –1;  i < 2; i++)
         for (j = –1; j < 2; j++)
            sum += X[I + i, J + j] * C [i, j];
      Y [I, J] = sum;
   };
```

In order to move towards hardware implementation, let us get rid of the loops.

```
I = 1;
Loop1: J = 1;
   Loop2: sum = 0; i = –1;
      Loop3: j = –1;
         Loop4:
            sum += X[I + i, J + j] * C [i, j];
            if (j < 1) {j++; goto Loop4};
         if (i < 1) {i++; goto Loop3};
      Y [I, J] = sum;
      if (J < n – 2) {J++; goto Loop2};
   if (I < m – 2) {I++; goto Loop1};
```

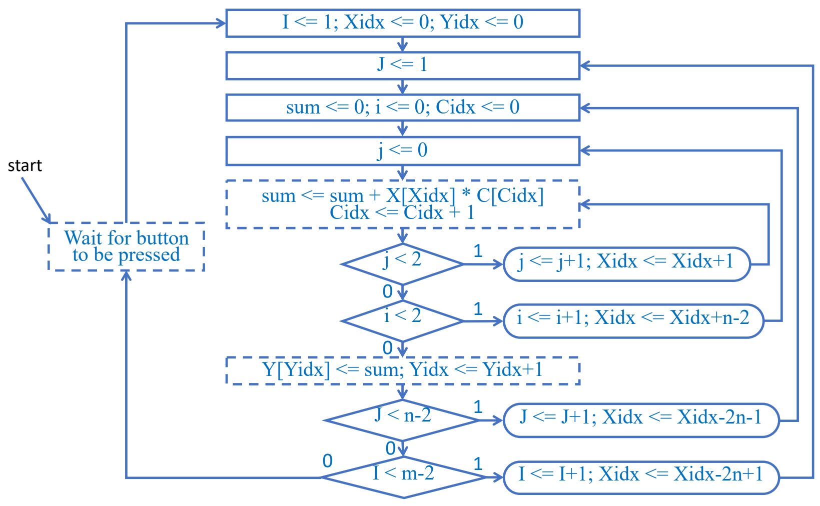
Next we replace 2-d matrices with 1-d matrices assuming row-wise storage. New variables, Xidx, Yidx and Cidx are used as indices for X, Y and C, respectively in place of index pairs (I+i, J+j), (I, J) and (i, j). Computing index values from I, J, i and j would involve multiplication. However, if we compute the next index value from the previous one, multiplication is avoided. Whereas Yidx and Cidx need to be simply incremented in order to get the next value (by virtue of row-wise storage coupled with row-wise scan), case of Xidx is more complex. The amount of increment/decrement is different for different loops.

- In Loop4, Xidx is incremented by 1 in order to move to the next element in the same row.
- In Loop3, Xidx is incremented by n – 2 in order to move to the first element in the next row within the same 3 x 3 block.
- In Loop2, Xidx is decremented by 2n + 1 in order to move to the first element in the first row of the next 3 x 3 block.
- In Loop1, Xidx is decremented by 2n – 1 in order to move to the first element in the first row of the first 3 x 3 block of the next scan.

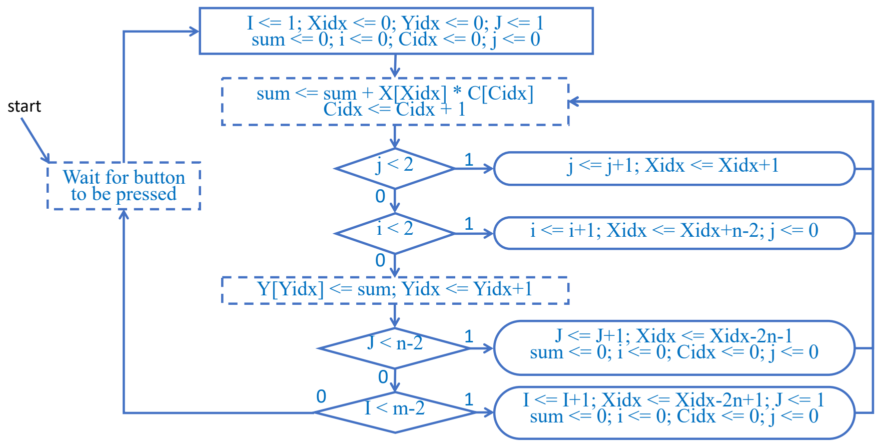
The code with 1-d arrays is shown below. In this code, ranges of i and j are changed from [–1, 1] to [0, 2] for convenience, because these two variables are only being used to count the loop iterations and not for indexing.

```
I = 1; Xidx = 0; Yidx = 0;
Loop1: J = 1;
   Loop2: sum = 0; i = 0; Cidx = 0;
      Loop3: j = 0;
         Loop4:
            sum += X[Xidx] * C [Cidx]; Cidx++;
            if (j < 2) {j++; Xidx++; goto Loop4};
         if (i < 2) {i++; Xidx = Xidx + n – 2; goto Loop3};
      Y [Yidx] = sum; Yidx++;
      if (J < n – 2) {J++; Xidx = Xidx – 2n – 1; goto Loop2};
   if (I < m – 2) {I++; Xidx = Xidx – 2n + 1; goto Loop1};
```

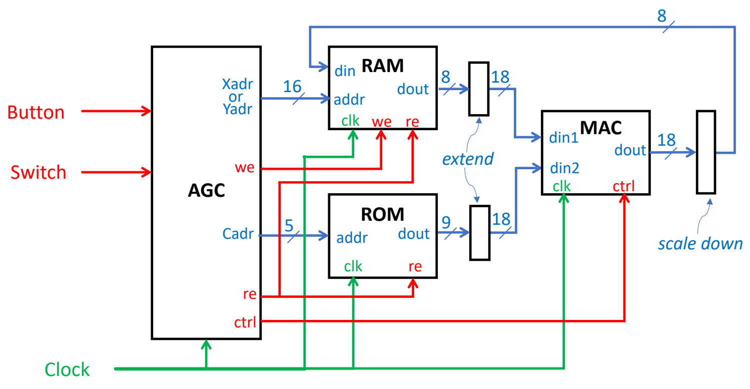
Based on this code, we can now construct the ASM chart shown below. The boxes with dashed outline need further elaboration that is taken up later.


I <= 1; Xidx <= 0; Yidx <= 0
J <= 1
sum <= 0; i <= 0; Cidx <= 0
j <= 0
start
sum <= sum + X[Xidx] * C[Cidx]
Cidx <= Cidx + 1
Wait for button to be pressed
j < 2
1
j <= j+1; Xidx <= Xidx+1
0
i < 2
1
i <= i+1; Xidx <= Xidx+n-2
0
Y[Yidx] <= sum; Yidx <= Yidx+1
J < n-2
1
J <= J+1; Xidx <= Xidx-2n-1
0
0
I < m-2
1
I <= I+1; Xidx <= Xidx-2n+1

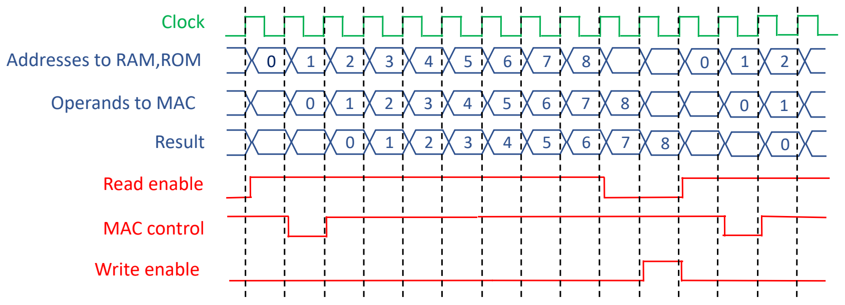
We can eliminate the states that are used for initializing loop indices J, i and j, by replicating initialization of these indices at the loop exits. To be more specific, initialization of j is done on exit paths of Loop4, Loop3 and Loop2, initialization of i is done on exit paths of Loop3 and Loop2, and initialization of J is done on the exit path of Loop2. Initialization of sum and Cidx are coupled with initialization of i. The reduced ASM chart is shown below.


I <= 1; Xidx <= 0; Yidx <= 0; J <= 1
sum <= 0; i <= 0; Cidx <= 0; j <= 0
start
sum <= sum + X[Xidx] * C[Cidx]
Cidx <= Cidx + 1
Wait for button to be pressed
j < 2
1
j <= j+1; Xidx <= Xidx+1
0
i < 2
1
i <= i+1; Xidx <= Xidx+n-2; j <= 0
0
Y[Yidx] <= sum; Yidx <= Yidx+1
J < n-2
1
J <= J+1; Xidx <= Xidx-2n-1
sum <= 0; i <= 0; Cidx <= 0; j <= 0
0
0
I < m-2
1
I <= I+1; Xidx <= Xidx-2n+1; J <= 1
sum <= 0; i <= 0; Cidx <= 0; j <= 0

Now let us look at the available RAM, ROM and MAC modules and refine the dotted boxes in this ASM chart. The adjoining figure shows how these modules can be connected. RAM and ROM directly feed the operands to MAC and the result produced by MAC goes to RAM. The additional circuit that needs to be designed is required to generate addresses and control signals for these modules. Let us call this circuit AGC (address and control signals generator).

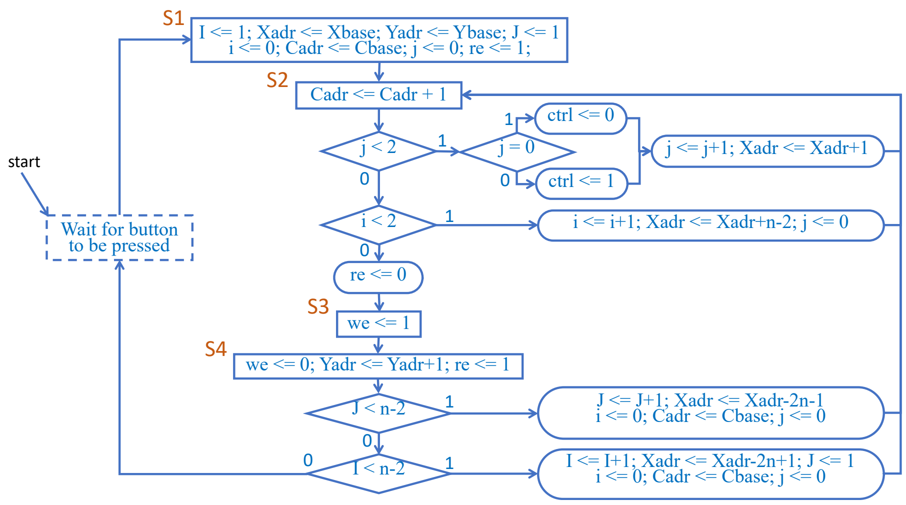


Names of the signals and port are shortened here to make the figure compact. There are three control signals required to be generated by AGC - re (read enable) going to RAM as well as ROM, we (write enable) going to RAM and ctrl (control) going to MAC. Address going to RAM is Xadr while reading X and Yadr while writing Y. Address going to ROM is Cadr. From the previous ASM chart that was representing the behaviour of entire filter, we need to derive an ASM chart representing the behaviour of just the AGC module. The essential task of the AGC module is to generate the three control signals and two addresses, while the actual reading/writing of arrays and doing multiplication/addition takes place outside this module. The AGC module needs to take care of the timings of memories and MAC module. As shown in the figure below, there is a one cycle delay between availability of RAM/ROM addresses and availability of operands for MAC. Similarly, there is a one cycle delay between availability of MAC operands and availability of result of multiplication and addition.



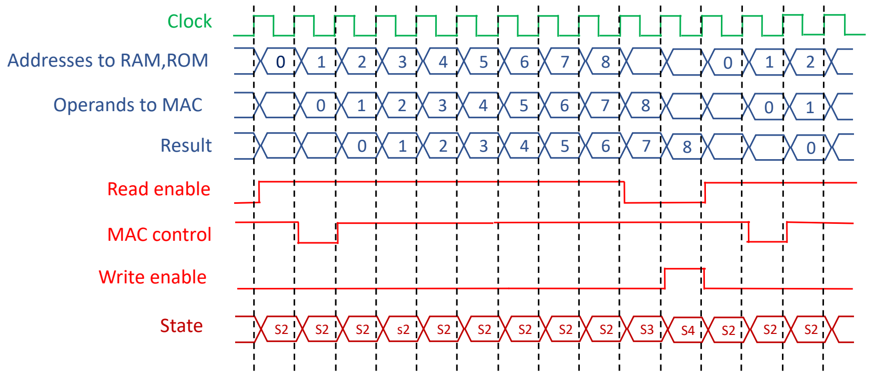
Numbers 0 .. 8 in the waveforms of addresses, operands and results indicate iteration number of the two innermost loops (3 x 3 block). Read enable signal re needs to be ‘1’ for 9 cycles. After that there is a cycle in which there is no memory access and then there is a cycle in which write enable signal we is made ‘1’ to facilitate writing of the result into RAM. MAC control signal ctrl is ‘0’ in the cycle when the first pair of operands is available to the MAC. It is kept ‘1’ otherwise.

The ASM chart for AGC is shown in the following figure. As mentioned above, AGC produces memory addresses rather than array index values. The addresses increment or decrement in the same way as array indices, but their initial values are not necessarily 0. Locations of arrays X and Y in RAM are secified as 0 and 32768 respectively. These are denoted by Xbase and Ybase in the chart. Location of array C in ROM is either 0 or 16, depending upon the filter selected by the external switch. This is denoted by Cbase. Addresses Xadr, Yadr and Cadr are initialized to the respective base addresses.


S1
I <= 1; Xadr <= Xbase; Yadr <= Ybase; J <= 1
i <= 0; Cadr <= Cbase; j <= 0; re <= 1;
S2
Cadr <= Cadr + 1
j < 2
j = 0
ctrl <= 0
ctrl <= 1
j <= j+1; Xadr <= Xadr+1
i < 2
i <= i+1; Xadr <= Xadr+n-2; j <= 0
re <= 0
S3
we <= 1
S4
we <= 0; Yadr <= Yadr+1; re <= 1
J < n-2
J <= J+1; Xadr <= Xadr-2n-1
i <= 0; Cadr <= Cbase; j <= 0
I < n-2
I <= I+1; Xadr <= Xadr-2n+1; J <= 1
i <= 0; Cadr <= Cbase; j <= 0
start
Wait for button to be pressed

In this design, re, we and ctrl are clock edge triggered signals. Signal re is required to be ‘1’ in state S2. Therefore, it is assigned ‘1’ in the states from which transition is made to S2 (i.e., S1 and S4) and assigned ‘0’ in state S2 on the exit path. Signal we is required to be ‘1’ in state S4. Therefore, it is assigned ‘1’ in the states from which transition is made to S4 (i.e., S3) and assigned ‘0’ in state S4. State S3 has been introduced to take care of one cycle gap between the last memory read cycle of of a 3 x 3 block and the memory write cycle. The first pair of operands is available to the MAC in the second iteration of state S2. Therefore, signal ctrl is assigned ‘0’ before entering S2 second time (or exiting first time) and assigned ‘1’ a cycle later.

The figure with waveforms shown earlier is reproduced below after including state changes as well.



VHDL code based on the above ASM chart is shown next. To take care of the box labelled “Wait for the button to be pressed”, another state S0 is introduced with a role similar to state S0 of slide 15 of Lecture 20. Here it is assumed that the signal Button carries a pulse of 1 cycle duration, derived from the output of the push-button using 2 D flip-flops plus an AND gate (as shown in figure 8.53 of the text book). An alternative to using this circuit is to introduce one additional state in a manner similar to slide 16 of Lecture 20.

Some new identifiers introduced in the VHDL code are as follows. I and J are replaced with ii and jj. XYadr is the address going to RAM which takes the value of Xadr or Yadr. CadrU is the internal signal of type unsigned which is assigned to Cadr after type casting. Cbase_smooth and Cbase_sharp are the base locations of the two sets of filter coefficients.

```
– – This is the VHDL code for AGC only. Code for the overall filter is not shown.
– – The overall filter primarily has instantiation statements for RAM, ROM, MAC and AGC.


entity AGC is
   port (
      Button, Switch, Clock : in std_logic;
      re, we, ctrl : out std_logic;
      XYadr : out std_logic_vector (15 downto 0);
      Cadr : out std_logic_vector (4 downto 0)
   );


architecture simple of AGC is

   declarations

begin

   concurrent assignments

   process (Clock) begin
      if (rising_edge (Clock)) then
         case State is
            when S0 =>

               statements for S0

            when S1 =>

               statements for S1

            when S2 =>

               statements for S2

            when S3 =>

               statements for S3

            when others =>

               statements for S4

         end case;
      end if;
   end process;
end simple;
```

```
constant m : natural := 120;
constant n : natural := 160;
constant Xbase : natural := 0;
contsnat Ybase : natural := 32768;
constant Cbase_smooth : natural := 0;
constant Cbase_sharp : natural := 16;
type state_type is (S0, S1, S2, S3, S4);
signal ii : unsigned (6 downto 0);
signal jj : unsigned (7 downto 0);
signal i, j : unsigned (1 downto 0);
signal State : state_type;
signal Xadr, Yadr : unsigned (15 downto 0);
signal Cbase, CadrU : unsigned (4 downto 0);
```

```
XYadr <= std_logic_vector (Yadr) when
            (we = ‘1’) else std_logic_vector (Xadr);
Cadr <= std_logic_vector (CadrU);
Cbase <= Cbase_smooth when (Switch = ‘0’)
            else Cbase_sharp;
```

```
if (Button = ‘1’) State <= S1; end if;
```

```
ii <= 1; Xadr <= Xbase; Yadr <= Ybase; jj <= 1;
i <= 0; CadrU <= Cbase; j <= 0; re <= 1;
State <= S2;
```

```
CadrU <= CadrU + 1;
if (j < 2) then
   if (j = 0) then ctrl <= ‘1’ else ctrl <= ‘0’ end if;
   j <= j + 1; Xadr <= Xadr + 1;
elsif (i < 2) then
   i <= i + 1; Xadr <= Xadr + n – 2; j <= 0;
else
   re <= ‘0’; State <= S3;
end if;
```

```
we <= ‘1’; State <= S4;
```

```
we <= 0; Yadr <= Yadr + 1; re <= 1
if (jj < n – 2) then
   jj <= jj + 1; Xadr <= Xadr – 2 * n – 1;
   i <= 0; CadrU <= Cbase; j <= 0; State <= S2;
elsif (I < m – 2) then
   ii <= ii + 1; Xadr <= Xadr – 2 * n + 1; jj <= 1;
   i <= 0; CadrU <= Cbase; j <= 0; State <= S2;
else
   State <= S0;
end if;
```