Belegarbeit

Hardwareentwurf

Exponential-Moving-Average Filter in Verilog

David Lohner, 81471

Leipzig, den 27. August 2023

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Aufgabenstellung

Ziel dieses Belegs ist es einen gleitenden Mittwelwertfilter (Exponential-Moving-Average (EMA)) in Verilog zu implementieren und mittels der Software „ModelSim“ zu testen. Das Filter gilt es als endlichen Zustandaustomaten zu entwerfen. Die Berechnungszeit pro Eingabewert soll maximal fünf Takte betragen. Zudem soll der Filterparameter α (siehe Gleichung 1) von externen Modulen frei einstellbar sein.

Die Dokumentation der Belegarbeit untergliedert sich in drei Abschnitte. Zu Beginn wird der Algorithmus des EMA-Filters in Abschnitt 2 vorgestellt und dessen Eigenschaften im Zeit- und Frequenzbereich beschrieben. Abschnitt 3 geht anschließend auf die Verarbeitungslogik und deren Funktionsweise ein. Abschließend werden in Abschnitt 4 die Simulation sowie die daraus resultierenden Ergebnisse vorgestellt und mit den Ausgangsdaten der Python-Implementierung verglichen.

2 Exponential-Moving-Average

Das in Verilog zu implementierende diskrete IIR Tiefpassfilter wird mit folgender Differenzengleichung mathematisch beschrieben.

$$y[k] = \alpha \cdot x[k] + (1 - \alpha) \cdot y[k - 1] \tag{1}$$

Durch den Faktor α wird die Grenzfrequenz sowie die Flankensteilheit des Filters beeinflusst. Die Grenzfrequenz wird mit der folgenden Gleichung berechnet.

$$f_c = \frac{f_s}{2\pi} \arccos\left[1 - \frac{\alpha^2}{2(1-\alpha)}\right] \tag{2}$$

Zur besseren Veranschaulichung zeigt Abbildung 1 die Impulsantwort und den Amplitudengang des EMA-Filters für $\alpha = 0.3$. Die Grenzfrequenz im Verhältnis zur Abtastfrequenz ist dabei als roter Punkt im Graphen markiert. Tabelle 1 zeigt zudem beispielhaft wie sich die Grenzfrequenz des Tiefpassfilters mit dem Parameter α beeinflussen lässt.

α	0.001	0.01	0.1	0.2	0.4	0.8
$\frac{f_c}{f_s}$	0.000159	0.001600	0.016784	0.035663	0.083129	0.352416

Tabelle 1: Grenzfrequenz des EMA-Filters in Abhängigkeit von α

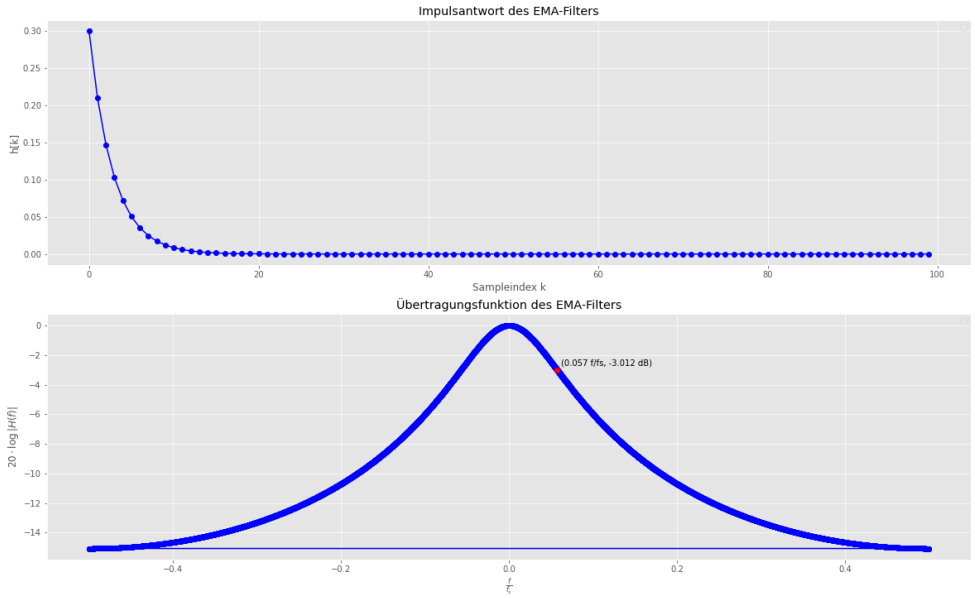

Abbildung 1: Impulsantwort und Amplitudengang des EMA-Filters für $\alpha = 0.3$

3 Verilog Implementierung

3.1 ema.v

Die Top-Level Konfiguration (siehe Abbildung 2) des Moduls „ema.v“ ist in folgender Abbildung (Abbildung 2) dargestellt. Die Portliste ist der Tabelle 2 zu entnehmen.

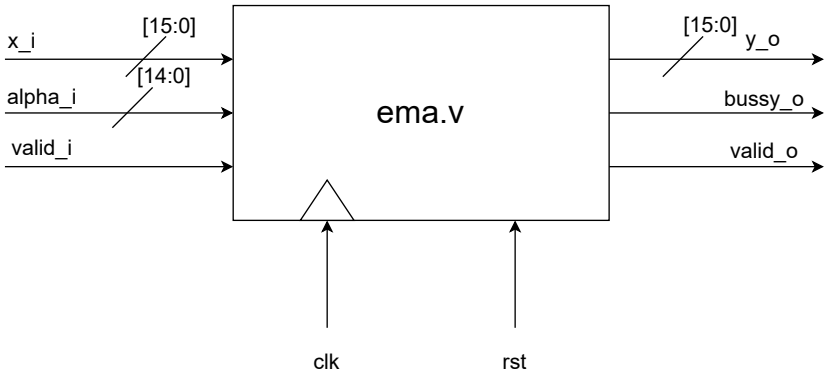

Abbildung 2: Top-Level IO Port-Definition des Moduls „ema.v“

Das Modul wird mit der Taktleitung „clk“ betrieben. Über ein „Valid“-Eingangssignal lässt sich die Verarbeitung eines Eingangswertes starten. Das „Valid“-Ausgangssignal hingegen signalisiert die erfolgreiche Verarbeitung des letzten Eingangssamples. Das

„Bussy“-Ausgangssignal dient dazu, anderen Modulen mitzuteilen, dass ein Sample gerade verarbeitet wird. Das Resetsignal setzt „high-aktiv“ die internen Zustands-, Zwischen- und Speicherregister zurück.

Port	Semantik	Format	I/O
clk	Taktsignal	1-Bit	I
rst	Resetsignal	1-Bit	I
x_i	Filtereingang	signed 16-Bit	I
alpha_i	Filterparameter α	unsigned 15-Bit	I
valid_i	Startsignal	1-Bit	I
y_o	Filterausgang	signed 16-Bit	O
bussy_o	Beschäftigungsstatus der FSM	1-Bit	O
valid_o	signalisiert validen Ausgang	1-Bit	O

Tabelle 2: Portliste des Moduls „ema.v“

Die Verilog Implementierung des EMA-Filters beruht auf einer Ablaufsteuerung mittels eines endlichen Zustandsautomaten (FSM). Aus der Differenzengleichung (siehe Gleichung 1) geht hervor, dass grundsätzlich zwei Multiplikationen, eine Addition und eine Subtraktion für die Berechnung des Filterausgangs benötigt werden. Die Subtraktion ergibt sich daraus, dass der Parameter α dem Modul übergeben wird (siehe Abbildung 2) und somit auch der Faktor $(1 - \alpha)$ nicht in einer LUT gespeichert werden kann. Mit einer von-Neumann ähnlichen Ablaufsteuerung ergeben sich zwei Zustände („FETCH“ und „EXECUTE“) pro Berechnungschritt. Ohne „IDLE“- und „EVAL“-Zustand werden so bereits acht Takte für die Berechnungen benötigt. Eine Optimierungsmöglichkeit ergibt sich jedoch bereits aus der richtigen Wahl des Zahlenformats. Die Ein- und Ausgangswerte werden als vorzeichenbehaftete 16-Bit Integerzahlen den Modulen übergeben (siehe Tabelle 2). Somit lassen sich Werte zwischen -2^{15} bis $(2^{15} - 1)$ mit dem Modul verarbeiten. Da der Parameter α den Wertebereich zwischen 0 und 1 abdeckt, muss für ihn ein Festkomma-Zahlenformat definiert werden. Hier wurde das Format $Q0.15$ gewählt. Dies hat folgenden Hintergrund: wird beispielsweise der Wert $0,4$ in diesem Zahlenformat dargestellt, ergibt sich die 15-Bit Festkommazahl 011001100110011 (13107). Das inverse dieser Binärzahl ist 100110011001100 (19660) und somit genau der Faktor $(1 - \alpha)$ im gewählten Festkommaformat. Dies lässt sich auch für weitere Werte von α verifizieren. Somit kann durch eine einfache Negierung des Eingabeparameters die Subtraktion $(1 - \alpha)$ eingespart werden. Da die ALU mit 16-Bit signed Integerwerten rechnet, wird das Festkommaformat noch um ein positives Vorzeichenbit bei dem Datentransfer zur ALU erweitert[1]. Auf den Faktor $\alpha = 1$ wurde im Festkommaformat verzichtet, da dieser das Filter brücken würde. Zudem lassen sich durch das Verwenden einer zweiten ALU

[1] siehe auch im Codelisting 1, Zeilen 143 und 148

die Multiplikationen $y[k-1] \cdot (1-\alpha)$ und $x[k] \cdot \alpha$ parallel ausführen. Wird die Addition der beiden Multiplikationsergebnisse im Evaluierungszustand ausgeführt,[2] können zwei weitere Takte ausgelassen werden. Folglich werden vier Zustände für die Implementierung des EMA-Filters benötigt. Diese werden in folgender Tabelle (siehe Tabelle 3) näher beschrieben.

Zustand	Kommentar	Folgezustand	Übergangsbedingung
IDLE	Ist sensitiv auf high-aktives valid_i um den Berechnungsvorgang zu starten	MULT_FETCH	valid_i == 1'b1
MULT_FETCH	Parallele Multiplikation von (x[k], α) und (y[k], (1-α))	MULT_EXEC	/
MULT_EXEC	Speichern der ALU Ergebnisse in Zwischenregistern	EVAL	(alu_result_valid & alu2_result_valid) == 1'b1
EVAL	Addition der beiden Zwischenregister und Belegung des Ausgangswires; Setzen des Ausgangsvalids valid_o	IDLE	/

Tabelle 3: Zustandssemantik des endlichen Zustandsautomaten

3.2 alu.v

Die Konfiguration des Moduls „alu.v“ ist in der nachstehenden Abbildung (siehe Abbildung 3) veranschaulicht. Dessen Portliste ist in der Tabelle 4 aufgeführt. Da die benötigten Multiplikationen parallel ausgeführt werden (siehe Abschnitt 3.1), sind zwei Instanzen des Moduls im Top-Modul „ema.v“ instantiiert. Der vollständige Code ist dem Beleg im Anhang unter dem Listing 2 zu entnehmen.

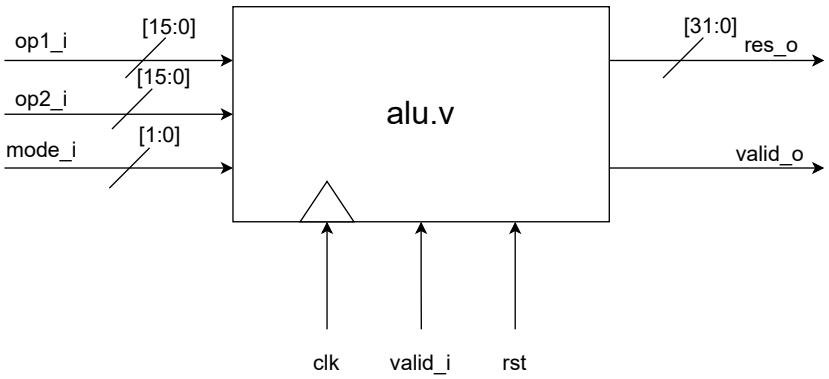

Abbildung 3: Top-Level IO Port-Definition des Moduls „alu.v“

Die ALU wird ebenfalls mit der Leitung „clk“ getaktet. Sie ist so konzipiert, dass die Addition und Multiplikation der Operanden „op1_i“ und „op2_i“ parallel ausgeführt wird.

[2]Möglich, da Multiplikationsergebnisse nach dem letzten „EXEC“ Zustand Wertstabil sind

Über den Eingang „mode_i“ lässt sich dann steuern, welches Rechenergebnis am Ausgang „res_o“ verfügbar ist. Die Leitung „valid_i“ wird im Modul getaktet gespeichert und wieder an den Ausgang „valid_o“ gelegt. Hat dieser den Wert Eins, wird das Verstreichen eines Taktes signalisiert und somit auch das Anliegen eines validen Ergebnisses auf der Leitung „res_o“. Zudem dient die Leitung „valid_o“ als Zustandsübergangsbedingung zwischen den Zuständen „MULT_EXEC“ und „EVAL“ (siehe Tabelle 3). Um die interne Bitbreite nicht zu vergrößern, werden die Ausgangswerte wieder auf ein 16-Bit signed Integerformat gebracht. [3]

Port	Semantik	Format	I/O
clk	Taktsignal	1-Bit	I
rst	Resetsignal	1-Bit	I
op1_i	Operand 2	signed 16-Bit	I
op2_i	Operand 2	signed 16-Bit	I
valid_i	Validsignal	1-Bit	I
mode_i	ALU Modi: IDLE, ADD, MULT	signed 16-Bit	I
res_o	Ergebnis der ALU Operation	signed 32-Bit	O
valid_o	Ausgangsvalid	1-Bit	O

Tabelle 4: Portliste des Moduls als.v

4 Simulation und Evaluierung

4.1 Testbench

Um die korrekte Funktionsweise des EMA-Filters mithilfe der Software „ModelSim“ zu überprüfen, ist die Testbench „ema_tb.v“ wie folgt konfiguriert worden. In jedem Takt wird ein Eingangssample aus der Datei „input.txt“ eingelesen. Nach einer Verzögerung von einem Takt wird dann das Startsignal „valid_i“ gesetzt und nach drei weiteren Takten zurückgesetzt. Somit ergibt sich insgesamt eine Verzögerung von vier Takten, welche genau der Verarbeitungszeit des Filters pro Eingangssample entspricht. Die Ausgangswerte werden in der Datei „output.txt“ gespeichert. Der Eingangsparameter α wird zu Beginn als Konstante im Festkommaformat hinterlegt. Die Testbench ist im Anhang unter dem Listing 3 ersichtlich.

[3] siehe Zeilen 159 und 160 des Listing 1 im Anhang

4.2 Timing

Die nachstehende Abbildung (siehe Abbildung 4) zeigt das Timing Diagramm des Top-Moduls sowie den beiden Instanzen „ALU“ und „ALU2“. Als Eingang dient ein weißes Rauschsignal. Der Parameter α beträgt 0.3. In der Abbildung 4 befindet sich der gelbe Cursor im letzten Zustand (current_state == 3) eines Bearbeitungszyklus. In diesem ist ein neues Ausgangssample verfügbar und das Validausgangssignal „valid_o“ wird gesetzt. Im nächsten Takt liegt das „valid_i“ Signal am Moduleingang „ema.v“ an, wodurch die FSM in den nächsten Zustand (next_state == 1) wechselt. Nun werden die ALU-Validsignale gesetzt und die Operanden an die Eingänge der beiden ALUs gelegt. So führt die Instanz „ALU“ die Multiplikation von $x[k] = -1168$ und $\alpha = 9830$ (0.3 in Q0.15) durch. „ALU2“ berechnet das Produkt aus $y[k-1] = -464$ und $(1-\alpha) = 22937$ (0.6 im FKF). Im nächsten Zustand (current_state == 2) werden die Ergebnisse, nach einer Abfrage des ALU-Validausgangssignals in Zwischenregistern gespeichert. Im letzten Zustand wird die Summe der Multiplikationsergebnisse auf die Ausgangsleitung gelegt und in einem Zwischenregister („y_last_r“) für den nächsten Durchlauf gesichert. Daraufhin wiederholt sich die Bearbeitungsschleife bis die Datei „input.txt“ vollständig durchlaufen wurde.

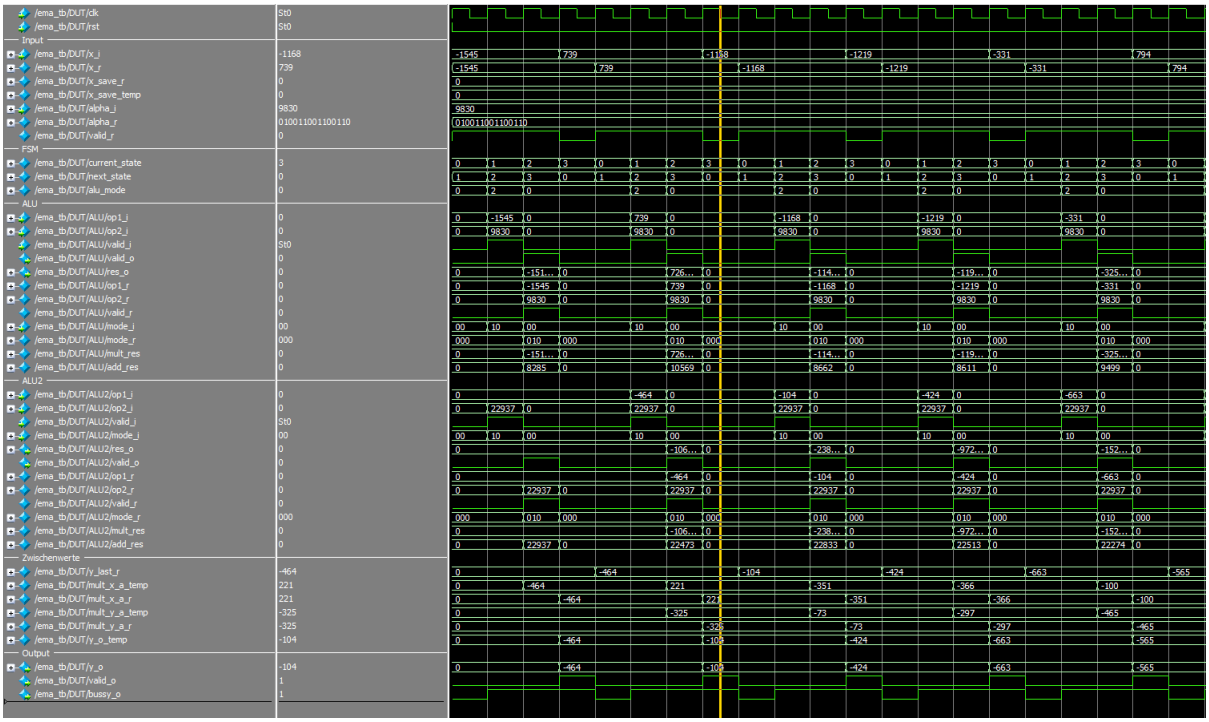

Abbildung 4: Timing Diagramm der EMA-Filterung in Verilog

4.3 Evaluierung der Ergebnisse

Wie im vorherigen Abschnitt erwähnt, dient ein Rauschsignal als Eingang des EMA-Filters. Dadurch kann über eine FFT des Ausgangssignals der Filterfrequenzgang ermittelt werden. Um einen Vergleich anführen zu können, ist das Filter ebenfalls in Python in einem Fixed[4]- sowie Floating-Point[5] Format implementiert worden. Für α wurde willkürlich der Wert 0.3 gewählt, was einer Grenzfrequenz von $\frac{f_c}{f_s} \approx 0.06$ entspricht. Das Python Programm ist dem nachstehenden Quellcode zu entnehmen.

```
1  # Exponential-Moving-Average
2  import numpy as np
3
4  x = np.genfromtxt('Noise.txt') #Eingangssignal
5  alpha = 0.3
6  y_last = 0
7  y = np.zeros(len(x))
8
9  # Int16, Fixed-Point Implementierung
10 for i in range (len(x)):
11     y[i] = np.int16((np.int32((x[i] * 9830))>>15)) + np.int16((np.int32((y_last
       ↪  * 22937))>>15)) # alpha = 0.3 im Format Q0.15
12     y_last = y[i]
13
14 # Floating-Point Implementierung
15 for i in range (len(x)):
16     y[i] = x[i] * alpha + y_last * (1-alpha)
17     y_last = y[i]
18
```

Abbildung 5 stellt die gefilterten Rauschsignale einander gegenüber. Als Vergleich wurde die Fixed-Point Python Implementierung angeführt. Die Abbildung macht deutlich, dass die Ausgangssignale identisch sind. Auch ein logischer Vergleich der beiden Ausgangsarrays bestätigt dies. Zudem ist auch das Tiefpassverhalten des EMA-Filters klar erkennbar. Wird das Ausgangssignal mit der floating-point Implementierung verglichen ergeben sich minimale Abweichungen, welche jedoch auf durch das Festkommaformat entstehende Rundungsfehler zurückzuführen sind.

[4]Zeilen 10 bis 12
[5]Zeilen 15 bis 17

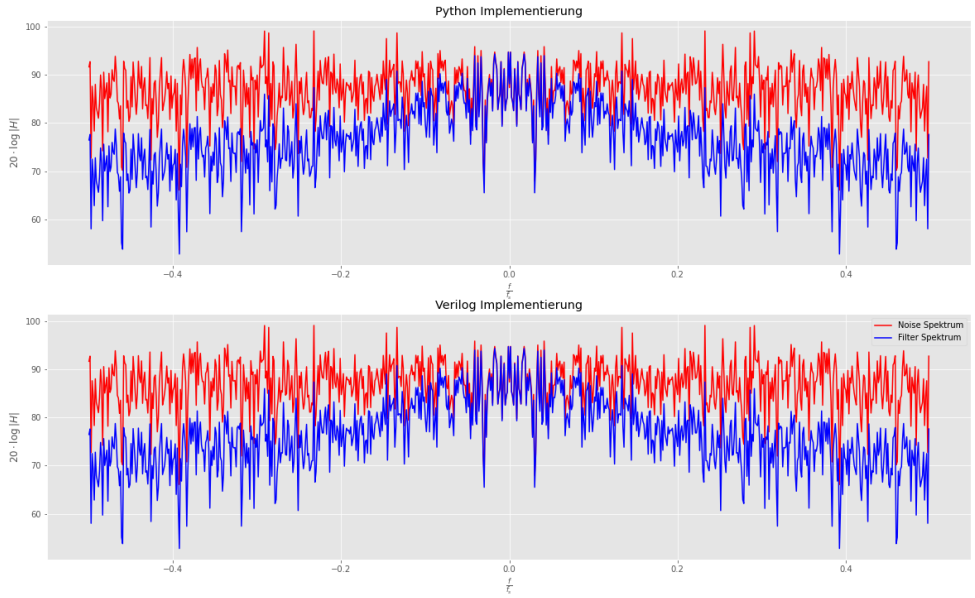

Abbildung 5: Vergleich der Filterausgangsspektren zwischen Python und Verilog

Abschließend lässt sich sagen, dass der EMA-Filter erfolgreich implementiert werden konnte. Die Verarbeitungszeit eines Samples beträgt vier Takte und liegt somit unterhalb der geforderten Obergrenze. Das Filter ist mit einer FSM implementiert worden und der Filterparameter α ist von externen Modulen innerhalb der sinnvollen Grenzen frei einstellbar. Die Verilog-Dateien, das ModelSim-Projekt sowie alle zur Entwicklung benötigten Textdateien und Python-Programme sind im GitHub-Repository verfügbar.

5 Anhang

```
module ema (
        clk,
        rst,
        x_i,
        alpha_i,
        valid_i,

        y_o,
        bussy_o,
        valid_o
);
parameter Win = 16;
parameter Wout = 16;
localparam Winternal = Win + Wout;

//ALU Modes
localparam ALU_IDLE = 2'd0;
localparam ADD      = 2'd1;
localparam MULT     = 2'd2;

//FSM State Definitions
localparam IDLE             = 2'd0;
localparam MULT_FETCH       = 2'd1;
localparam MULT_EXEC        = 2'd2;
localparam EVAL             = 2'd3;

// I/O Ports
input                               clk;
input                               rst;
input signed [Win-1:0]              x_i;
input  [Win-2:0]                    alpha_i;
input                               valid_i;

output wire signed  [Wout-1:0]  y_o;
output reg                          valid_o, bussy_o;

reg                                 valid_r;
reg signed [Win-1:0]                x_r;
reg        [Win-2:0]                alpha_r;

reg signed [Win-1:0]                x_save_temp, x_save_r;
reg signed [Win-1:0]                y_o_temp;
reg signed [Win-1:0]                y_last_r;

//FSM
reg [1:0]                           current_state, next_state;

//ALU
reg [1:0]                           alu_mode, alu2_mode;
reg                                 alu_valid, alu2_valid;
reg [Win-1:0]                       alu_op1, alu_op2;
reg [Win-1:0]                       alu2_op1, alu2_op2;
wire                                alu_result_valid, alu2_result_valid;
wire signed [Winternal-1:0]         alu_result, alu2_result;
reg signed [Winternal-1:0]          alu_result_r, alu2_result_r;

reg signed [Win-1:0]            mult_x_a_temp, mult_x_a_r;
reg signed [Win-1:0]            mult_y_a_temp, mult_y_a_r;

alu ALU (
.clk                (clk),
.rst                (rst),
.op1_i              (alu_op1),
.op2_i              (alu_op2),
```

```
65  .valid_i            (alu_valid),
66  .mode_i             (alu_mode),
67
68  .res_o              (alu_result),
69  .valid_o            (alu_result_valid)
70  );
71
72  alu ALU2 (
73  .clk                (clk),
74  .rst                (rst),
75  .op1_i              (alu2_op1),
76  .op2_i              (alu2_op2),
77  .valid_i            (alu2_valid),
78  .mode_i             (alu2_mode),
79
80  .res_o              (alu2_result),
81  .valid_o            (alu2_result_valid)
82  );
83
84  always @ (posedge clk) begin
85      if (rst) begin
86          valid_r         <= 1'b0;
87          x_save_r        <= 'd0;
88          x_r             <= 'd0;
89          alpha_r         <= 'd0;
90          current_state   <= IDLE;
91          alu_mode        <= ALU_IDLE;
92          alu_result_r    <= 'd0;
93          mult_x_a_r      <= 'd0;
94          mult_y_a_r      <= 'd0;
95          y_last_r        <= 'd0;
96
97      end
98
99      else begin
100         valid_r         <= valid_i;
101         x_save_r        <= x_save_temp;
102         x_r             <= x_i;
103         alpha_r         <= alpha_i;
104         mult_x_a_r      <= mult_x_a_temp;
105         mult_y_a_r      <= mult_y_a_temp;
106         alu_result_r    <= alu_result;
107         alu2_result_r   <= alu2_result;
108         current_state   <= next_state;
109         y_last_r        <= y_o_temp;
110
111     end
112 end
113
114 always @ (*) begin
115     x_save_temp         = x_save_r;
116     y_o_temp            = y_last_r;
117     mult_x_a_temp       = mult_x_a_r;
118     mult_y_a_temp       = mult_y_a_r;
119     valid_o             = 1'b0;
120     bussy_o             = 1'b1;
121     alu_mode            = ALU_IDLE;
122     alu_op1             = 'd0;
123     alu_op2             = 'd0;
124     alu_valid           = 1'b0;
125     alu2_mode           = ALU_IDLE;
126     alu2_op1            = 'd0;
127     alu2_op2            = 'd0;
128     alu2_valid          = 1'b0;
129     next_state          = current_state;
130
```

```
131 case (current_state)
132     IDLE: begin
133     bussy_o      = 1'b0;
134     alu_mode     = ALU_IDLE;
135         if (valid_r) begin
136             next_state  = MULT_FETCH;
137         end
138     end
139 
140     MULT_FETCH: begin
141         alu_mode     = MULT;
142         alu_op1      = x_r;
143         alu_op2      = {1'd0, alpha_r};
144         alu_valid    = 1'b1;
145 
146         alu2_mode    = MULT;
147         alu2_op1     = y_last_r;
148         alu2_op2     = {1'd0, ~alpha_r}; // (1-alpha) = inverted Binary
149         alu2_valid   = 1'b1;
150 
151         next_state   = MULT_EXEC;
152     end
153 
154     MULT_EXEC: begin
155         if (alu_result_valid & alu2_result_valid) begin
156         alu_mode         = ALU_IDLE;
157         alu2_mode        = ALU_IDLE;
158         next_state       = EVAL ;
159         mult_x_a_temp    = alu_result >>> (Win-1);
160         mult_y_a_temp    = alu2_result >>> (Win-1);
161         end
162     end
163 
164     EVAL: begin
165         valid_o          = 1'b1;
166         y_o_temp         = mult_x_a_r + mult_y_a_r;
167         alu_mode         = ALU_IDLE;
168         next_state       = IDLE;
169     end
170 endcase
171 
172 end
173 
174 assign y_o = y_o_temp;
175 
176 endmodule
```

Listing 1: ema.v

```
module alu(
    clk,
    rst,
    op1_i,
    op2_i,
    mode_i,
    valid_i,

    res_o,
    valid_o
);

parameter Win       = 16;
parameter Wout      = 32;

input                   clk;
input                   rst;
input signed [Win-1:0]  op1_i;
input signed [Win-1:0]  op2_i; // F r alpha
input                   valid_i;
input [1:0]             mode_i;


output reg signed  [Wout-1:0] res_o;
output reg valid_o;

//ALU Modes
localparam ALU_IDLE = 2'd0;
localparam ADD      = 2'd1;
localparam MULT     = 2'd2;

reg signed [Win-1:0]    op1_r;
reg signed [Win-1:0]    op2_r;
reg                     valid_r;
reg [2:0]               mode_r;

/// MULT STAGE
wire signed [Wout-1:0]  mult_res;
assign mult_res = op1_r * op2_r;

// ADD STAGE
wire signed [Wout-1:0]  add_res;
assign add_res  = op1_r + op2_r;


/// sequential part starts here
always @(posedge clk) begin
    if (rst) begin
        op1_r   <= 'd0;
        op2_r   <= 'd0;
        mode_r  <= ALU_IDLE;
        valid_r <= 'd0;
    end
    else begin
        op1_r   <= op1_i;
        op2_r   <= op2_i;
        mode_r  <= mode_i;
        valid_r <= valid_i;
    end
end

/// combinational part starts here for writing L-values to buffer
always @(*) begin
    res_o       = 'd0;
    valid_o     = valid_r;
```

```
66      if(mode_r == ADD) begin
67          res_o       = add_res;
68      end
69      else if(mode_r == MULT) begin
70          res_o       = mult_res;
71      end
72      else if (mode_r == ALU_IDLE) begin
73          res_o       = 'd0;
74      end
75  end
76  endmodule
```

Listing 2: alu.v

```
1   module ema_tb;
2
3   reg clk, rst;
4
5
6   reg [15:0] x_i; //Q16.0
7   reg [14:0] alpha_i; //1Q.15
8   reg valid_i;
9
10  wire signed [15:0] y_o; //Q16.0
11  wire bussy_o, valid_o;
12
13  integer fd_i, fd_o, tmp;
14
15  ema DUT(
16      .rst        (rst),
17      .clk        (clk),
18      .x_i        (x_i),
19      .alpha_i    (alpha_i),
20      .valid_i    (valid_i),
21
22      .y_o        (y_o),
23      .bussy_o    (bussy_o),
24      .valid_o    (valid_o)
25  );
26
27  always
28          #1 clk = !clk;
29
30  initial begin
31      fd_i = $fopen("input.txt", "r");
32      fd_o = $fopen("output.txt", "w");
33
34      if (fd_i)       $display("File was opened successfully : %0d", fd_i);
35      else            $display("File was NOT opened successfully : %0d", fd_i);
36
37      if (fd_o)       $display("File was opened successfully : %0d", fd_o);
38      else            $display("File was NOT opened successfully : %0d", fd_o);
39
40      #50
41      clk     = 0;
42      rst     = 1;
43      x_i     = 16'd0;
44      alpha_i = 15'd9830; // > alpha = 0.3 1Q15
45      #2
46      rst         = 0;
47
48  end
49
50  always @ (posedge clk) begin
51
52      if (!($feof(fd_i))) begin
```

```
53              tmp = $fscanf(fd_i, "%d\n", x_i);
54              #2
55              valid_i = 1'b1;
56              #6
57              $fwrite(fd_o, "%d\n", y_o);
58              valid_i = 1'b0;
59      end
60
61      else begin
62          $fclose(fd_i);
63          $fclose(fd_o);
64          $finish;
65      end
66  end
67  endmodule
```

Listing 3: ema_tb.v