Architectures for Computer Vision
from Algorithm to Chip with Verilog

Hong Jeong

©2014 John Wiley & Sons Singapore Pte Ltd. Published 2014 by John Wiley & Sons Singapore Pte Ltd.

March 13, 2015
Part 1 Verilog HDL
Chapter 3 Processor, Memory, and Array
Contents

Image Processing System

Taxonomy of Algorithms and Architectures

Neighborhood Processor

BP Processor

DP Processor

Forward and Backward Processors

Frame Buffer and Image Memory

Multidimensional Array

Queue

Stack

Linear Systolic Array
Figure: The general structure of image processing system.
Contents

Image Processing System

Taxonomy of Algorithms and Architectures
  Neighborhood Processor
  BP Processor
  DP Processor
  Forward and Backward Processors
  Frame Buffer and Image Memory
  Multidimensional Array
  Queue
  Stack
  Linear Systolic Array
<table>
<thead>
<tr>
<th>Memory Processor</th>
<th>a) multi-dimensional array</th>
<th>b) local Register</th>
<th>c) global Queue</th>
<th>d) local Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Neighborhood processor</td>
<td>IN algorithm</td>
<td>SAT algorithm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2) BP processor</td>
<td>BP algorithm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3) Viterbi processor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4) Forward/Backward processor</td>
<td></td>
<td></td>
<td>HMM, DP algorithms</td>
<td></td>
</tr>
</tbody>
</table>

Table: PEs and memories for fundamental algorithms.

cf. IN: Iterative neighborhood, BP: Belief Propagation, and SAT: Sum Area Table.
Contents

Image Processing System

Taxonomy of Algorithms and Architectures

**Neighborhood Processor**

BP Processor

DP Processor

Forward and Backward Processors

Frame Buffer and Image Memory

Multidimensional Array

Queue

Stack

Linear Systolic Array
Neighborhood Operation

\[ Q(x, y) \leftarrow T\{ I(x, y), Q(x, y), Q(x + 1, y), \\
Q(x - 1, y), Q(x, y - 1), Q(x, y + 1) \}, \]

\[ O(x, y) \leftarrow H(Q(x, y)), \quad (1) \]
Figure: Processing elements for neighborhood computation.
Listing 1: The neighborhood processor: pe_neighbor.v

```verilog
module pe_neighbor #(parameter DATA_WIDTH=32) (  
    input clock, reset,  
    input signed [DATA_WIDTH-1:0] image, //pixel value  
    input signed [DATA_WIDTH-1:0] q_east, q_west,  
      q_south, q_north //neighbor states  
    output reg signed [DATA_WIDTH-1:0] q, result //state and result  
  );  
  //Moore machine  
  parameter STATE1 = 1’b0, STATE2 = 1’b1; //assign states  
  //initialize state  
  reg [1:0] state = STATE1;  
  always @(posedge clock) begin  
    if (reset) begin //reset  
      state <= STATE2; //next state  
      q <= 0;  
    end  
    else case (state)  
      STATE1 : begin //idle state  
        state <= STATE1;  
        q <= 0;  
      end
  endcase  
endmodule
```
STATE2 : begin //main operation
    q <= T(image, q, q_east, q_west,
         q_south, q_north);   //state transition
    result <= H(q);       //observation
end
endcase
end //always
endmodule
Contents

Image Processing System
Taxonomy of Algorithms and Architectures
Neighborhood Processor
BP Processor
DP Processor
Forward and Backward Processors
Frame Buffer and Image Memory
Multidimensional Array
Queue
Stack
Linear Systolic Array
BP Operations

\[ Q_e(x, y) \leftarrow T_e\{I(x, y), Q_s(x, y + 1), Q_w(x - 1, y), Q_n(x, y - 1)\}, \]
\[ Q_w(x, y) \leftarrow T_w\{I(x, y), Q_e(x + 1, y), Q_s(x, y + 1), Q_n(x, y - 1)\}, \]
\[ Q_s(x, y) \leftarrow T_s\{I(x, y), Q_e(x + 1, y), Q_w(x - 1, y), Q_n(x, y - 1)\}, \]
\[ Q_n(x, y) \leftarrow T_n\{I(x, y), Q_e(x + 1, y), Q_s(x, y + 1), Q_w(x - 1, y)\}, \]
\[ O(x, y) \leftarrow H\{I(x, y), Q_w(x - 1, y), Q_e(x + 1, y), Q_s(x, y + 1), Q_n(x, y - 1)\}. \]
Figure: A processing element with four states.
Listing 2: The BP processor: pe.v

```verilog
module pe #(DATA_WIDTH=32) (  
    input clock, reset,  
    input signed [DATA_WIDTH-1:0] image, //pixel image  
    input signed [DATA_WIDTH-1:0] i_east, i_west,  
    i_south, i_north //neighbor states  
    output reg signed [DATA_WIDTH-1:0] o_east, o_west,  
    o_south, o_north, //new states  
    output reg signed [DATA_WIDTH-1:0] result, //result  
);  
//define states  
parameter STATE1 = 1'b0, STATE2 = 1'b1; //assign states  
//initialize state  
reg [1:0] state = STATE1;  
always@(posedge clock) begin  
    if (reset) begin //reset  
        state <= STATE2; //next state  
        result <= 0;  
    end  
    else case (state)  
        STATE1 : begin //idle state  
            state <= STATE1;  
        end  
    endcase;  
end
```

Hong Jeong (©2014 John Wiley & Sons Singapore Pte Ltd.)
result <= 0;
end
STATE2 : begin //state transition
    o_east <= T_e(image, i_south, i_west, i_north); //east out
    o_west <= T_w(image, i_east, i_south, i_north); //west out
    o_south <= T_s(image, i_east, i_west, i_north); //south out
    o_north <= T_n(image, i_east, i_west, i_south); //north out
    result <= H(i_east, i_west, i_south, i_north); //result
end
endcase
end //always
endmodule
Contents

Image Processing System
Taxonomy of Algorithms and Architectures
Neighborhood Processor
BP Processor

DP Processor
Forward and Backward Processors
Frame Buffer and Image Memory
Multidimensional Array
Queue
Stack
Linear Systolic Array
Forward Operations

\[
\begin{align*}
\phi(i, j) &= \min_{k \in [0, N_q - 1]} \phi(i - 1, k) + \mu(k, j) + \rho(i, j), \\
\eta(i, j) &= \arg \min_{k \in [0, N_q - 1]} \phi(i - 1, k) + \mu(k, j), \quad j \in [0, N_q - 1].
\end{align*}
\tag{3}
\]
Figure: A linear array and the PE in forward processing.
Listing 3: Viterbi forward processor: pe.v

module forward_processor #(parameter DATA_WIDTH = 16,
                      STACK_DEPTH = 100)(
    input clock, reset,
    input signed [DATA_WIDTH-1:0] rho, phiu, phid, //up down and queue
    output reg [1:0] eta, pushpop                  //pointer to queue
);
//variables
parameter mu = 1, muu = 2, mud = 2;
reg [DATA_WIDTH-1:0] phi;                   //cost
integer epsilon = 1;                       //threshold

always @ (posedge clock) begin
    pushpop = 0;                            //no operation
    if (phiu + muu - phid - mud > epsilon) begin //choose smaller
        phi <= phid + mud;                   //new cost
        eta <= 2’b11;                       //new pointer
    end
    else if (phiu + muu - phid - mud > epsilon)begin
        phi <= phiu + muu;
        eta <= 2’b01;
    end
else begin //choose itself
    phi <= phi + mu; //new cost
    eta <= 2'b00; //new pointer
end
pushpop = 2'b01; //push
end //always
endmodule
(a) A linear array

(b) a PE

Figure: A linear array and the PE in the backward processing.
Backward Operations

\[ b(j + \eta(j)) \leftarrow 1, \text{if } b(j) \cdot \eta(j) = 1, \]

(4)
Listing 4: Viterbi Backward processor: backward_processor.v

module backward_processor (  
  input clock, reset,  
  input [1:0] fu, fd, //flag input  
  output [1:0] qu, qd, //flag output  
  output [1:0] pushdown, //pop  
  input [DATA_WIDTH-1:0] q //popped data  
);

reg flag = 0; //current flag

always @ (posedge clock) begin
  qu = 0; qd = 0; //initialize output
  pushpop = 2’b11; //issue pop
  if (flag & q == 2'b01) qu <= 1; //activate upward
  else if (flag & q == 2'b11) qd <= 1; //activate downward
  else if (flag) flag <=0; //keep unchanged
  else flag <= 0; //turn off the flag
  end //always
endmodule
Contents

Image Processing System

Taxonomy of Algorithms and Architectures

Neighborhood Processor

BP Processor

DP Processor

Forward and Backward Processors

Frame Buffer and Image Memory

Multidimensional Array

Queue

Stack

Linear Systolic Array
Forward Backward Probabilities

\[
\begin{align*}
\alpha(i, j) &= \sum_{k \in [0, N_q - 1]} \alpha(i - 1, k) + \mu(k, j) + \rho(i, j), \\
\beta(i, j) &= \sum_{k \in [0, N_q - 1]} \beta(i + 1, k) + \mu(j, k) + \rho(i + 1, k),
\end{align*}
\]
\[ \alpha(i-1, N_q - 1) \rightarrow \ldots \rightarrow \alpha(i-1, 0) \rightarrow \text{PE} \rightarrow \alpha(i, j) \]

**Figure:** A forward processor.
Listing 5: Forward processor: pe.v

module forward_processor #(parameter DATA_WIDTH = 16)(
  input clock, reset,
  input signed [DATA_WIDTH-1:0] rho, alpha, //input
  output reg signed [DATA_WIDTH-1:0] q,       //output
);
//variables
parameter mu = 1;

always @ (posedge clock) begin
  if (reset) q <= 0;
  else q <= q + alpha + mu + rho;
end //always
endmodule
Contents

Image Processing System

Taxonomy of Algorithms and Architectures

Neighborhood Processor

BP Processor

DP Processor

Forward and Backward Processors

Frame Buffer and Image Memory

Multidimensional Array

Queue

Stack

Linear Systolic Array
Listing 6: Frame buffer: fbuffer.v

```
module fbuffer
  #(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=10)
  (input we, clock,
  input [(DATA_WIDTH-1):0] data,     //input data
  input [(ADDR_WIDTH-1):0] read_addr, write_addr, //address
  output reg [(DATA_WIDTH-1):0] q     //output data
);

//declare the RAM variable
reg [DATA_WIDTH-1:0] ram[0: 2 ** ADDR_WIDTH-1];

always @(posedge clock) begin
  //write and read
  if (we) ram[write_addr] <= data;    //write data
  q <= ram[read_addr];               //read data
end //always

endmodule
```
Listing 7: 2D array: imem.v

module imem #(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=10)(
    input we_a, we_b, clock,
    input [(DATA_WIDTH-1):0] data_a, data_b, //input data ports
    input [(ADDR_WIDTH-1):0] addr_a, addr_b, //addresses
    output reg [(DATA_WIDTH-1):0] q_a, q_b //output data ports
);
    //declare the RAM variable
    reg [DATA_WIDTH-1:0] ram[0: 2 ** ADDR_WIDTH-1];
    //port A
    always @ (posedge clock)
    begin
        if (we_a)
            begin
                ram[addr_a] <= data_a; //write data
                q_a <= data_a; //store the address
            end
        else q_a <= ram[addr_a]; //output data
    end //always
    //port B
    always @ (posedge clock)
    begin
    ```
if (we_b)
begin
    ram[addr_b] <= data_b; //write data
    q_b <= data_b; //store the address
end
else q_b <= ram[addr_b]; //output the data
end //always
endmodule
Contents

Image Processing System
Taxonomy of Algorithms and Architectures
Neighborhood Processor
BP Processor
DP Processor
Forward and Backward Processors
Frame Buffer and Image Memory
Multidimensional Array
Queue
Stack
Linear Systolic Array
A 2D memory, $Q$: $(x,y)$ for address, data for input, $q$ for output, read, write, and clock for control and clock signals.
Listing 8: Module: ram2d.v

module ram #(parameter DATA_WIDTH=32, ADDR_WIDTH=10) (  
    input we, clock,  
    input [DATA_WIDTH-1:0] data, //input data  
    input [ADDR_WIDTH-1:0] x,y, //address  
    output [DATA_WIDTH-1:0] q //output data  
);  
//declare the variables  
reg [DATA_WIDTH-1:0] ram[0:2 ** (2 * ADDR_WIDTH)-1]; //RAM  
reg [ADDR_WIDTH-1:0] x_reg, y_reg; //hold address  
//write data  
always @ (posedge clock) begin  
    if (we) ram[0:2 ** (2 * ADDR_WIDTH)-1] //write  
        x_reg <= x; y_reg <= y; //store address  
end //always  
//read data  
assign q = ram[0:2 ** (2 * ADDR_WIDTH)-1]; //output  
endmodule
Contents

Image Processing System
Taxonomy of Algorithms and Architectures
  Neighborhood Processor
  BP Processor
  DP Processor
Forward and Backward Processors
Frame Buffer and Image Memory
Multidimensional Array
Queue
Stack
Linear Systolic Array

Hong Jeong (©2014 John Wiley & Sons Singapore Pte Ltd.)

Architectures for Computer Vision
Queue Operation

\[ Q = \{ q(1), q(2), \ldots, q(N_q) \}, \]
A queue $Q$: $q$ for input, $A$ for fixed addresses, $Q(A)$ for output, shift and clock for control and clock signals.
Push Operation

\[ Q \leftarrow \text{push} \ x. \quad (7) \]
Listing 9: Queue: queue.v

module queue #(parameter DATA_WIDTH=8, DATA_NUM = 2, QUEUE_DEPTH=10, ADDRO = 0, ADDR1 = 10)(
  clock, push, data, q0, q1);
input clock, push;
input signed [DATA_WIDTH * DATA_NUM - 1:0] data; //input data
output reg signed [DATA_WIDTH-1:0] q0, q1; //output data

//define variables
reg [DATA_WIDTH-1:0] ram [0:QUEUE_DEPTH * DATA_NUM -1]; //ram
integer i,j;

//push the data
always @(posedge clock) begin
  if (push) begin
    for (j = DATA_NUM; j > 0; j = j - 1) begin
      for(i = QUEUE_DEPTH * DATA_NUM - 1; i > 0; i = i - 1)
        ram[i] <= ram[i-1]; //shift data
      //pop out and push the data (indexed part select)
      ram[0] <= data[(j * DATA_WIDTH - 1) -: DATA_WIDTH];
    end
    q0 <= ram[ADDRO]; q1 <= ram[ADDR1]; //read data
  end
end
end
end //always
endmodule
Contents

Image Processing System

Taxonomy of Algorithms and Architectures

Neighborhood Processor

BP Processor

DP Processor

Forward and Backward Processors

Frame Buffer and Image Memory

Multidimensional Array

Queue

Stack

Linear Systolic Array
A stack $Q$: data for input, $q$ for output, push, pop, and clock for control and clock signals.

**Figure:** A stack $Q$: data for input, $q$ for output, push, pop, and clock for control and clock signals.
Listing 10: Stack: stack.v

module Stack #(parameter DATA_WIDTH = 8, STACK_DEPTH = 100)(
    input clock, reset, // clock and reset
    input [1:0] state, // state
    input [DATA_WIDTH-1:0] data, // input data
    output reg [DATA_WIDTH-1:0] q // output data
);

// declaration memory
reg [DATA_WIDTH-1:0] ram[0:STACK_DEPTH-1]; // stack
// integer variables
integer stack_address, i;
// assign states
parameter STATE_IDLE = 0, STATE_PUSH = 1, STATE_POP = 2; // states
// always block
always @(posedge clock) begin
    if (reset) begin // reset
        for (i=0; i < STACK_DEPTH; i=i+1) ram[i] <= 0;
        stack_address <= 0;
    end
    else case (state)
        STATE_IDLE : q <= 8’hZZ; // idle state
        STATE_PUSH : begin // push state
if (stack_address == 0) begin
    ram[0] <= data;
    stack_address <= stack_address + 1;
end else if (stack_address < STACK_DEPTH - 1) begin
    ram[stack_address] <= data;
    stack_address <= stack_address + 1;
end else begin
    ram[stack_address] <= data;
    stack_address <= stack_address;
end

STATE_POP : begin //pop state
    if (stack_address == 0) begin
        q <= ram[stack_address];
        ram[stack_address] <= 0;
        stack_address <= stack_address;
    end else begin
        q <= ram[stack_address];
        ram[stack_address] <= 0;
        stack_address <= stack_address - 1;
    end
end

default : begin //fault recovery
    q <= 8’hZZ;
end
endcase
end //always
endmodule
Contents

Image Processing System
Taxonomy of Algorithms and Architectures
Neighborhood Processor
BP Processor
DP Processor
Forward and Backward Processors
Frame Buffer and Image Memory
Multidimensional Array
Queue
Stack

Linear Systolic Array
FIR Operation

\[ y(t) = \sum_{k=0}^{K-1} w_k x(t - k). \]  

(8)
Algorithm (Serial algorithm)

Compute the following.

- input: $x(t)$.
- output: $y(t)$.
- parameters: $\{w_0, w_1, w_2\}$.

1. for $t = 0, 1, \ldots$
   1.1 $y \leftarrow w_0 x(t) + w_1 x(t - 1) + w_2 x(t - 2) + \ldots + w_{K-1}(t - k)$,
   1.2 output: $y$. 
Figure: Linear systolic arrays for convolution. (Numbers represent the number of registers. PEs are on the right.)
Algorithm (PE(k))
For a PE(k), do the following.

- **input:** \(\{x_i(k), y_i(k)\}\).
- **memory:** \(w(k), x_o(k), \{y(k), y_o(k)\}\).
- **output:** \(\{x_o(k), y_o(k)\}\).

1. **initialization:** \(w(k)\).
2. **for each clock**
   2.1 \(y = y_i(k) + w(k)x_i(k)\).
   2.2 **output:** \(x_o(k) \leftarrow x(k), y_o(k) \leftarrow y\).
Listing 11: Module: pe.v

```
//timescale 1ns / 100ps  //unit time/ precision
//define DATA_WIDTH 32  //parameter
//module pe, the signals are signed for 2’s complement arithmetic.
module pe(
    input signed ['DATA_WIDTH-1:0] xi,  //signal input
    output reg signed ['DATA_WIDTH-1:0] xo,  //signal output
    input signed ['DATA_WIDTH-1:0] yi,  //output input
    output reg signed ['DATA_WIDTH-1:0] yo,  //output output
    input clock,
    input reset
);
    //Moore machine
    parameter STATE1 = 2’b01;  //idle state
    parameter STATE2 = 2’b01;  //input weights
    parameter STATE3 = 2’b10;  //store weights
    parameter STATE4 = 2’b11;  //main operations
    reg [8:1] clock_count;  //for weight input
    reg [1:0] state = STATE1;  //initialize state
    reg signed ['DATA_WIDTH-1:0] w;  //weight
    reg signed ['DATA_WIDTH-1:0] y;  //simulate output register
```
/* FSM_ENCODING='SEQUENTIAL', SAFE_IMPLEMENTATION='YES',
SAFE_RECOVERY_STATE='<recovery_state_value>' */ //attributes
always@(posedge clock) begin //sequential circuit
    if (reset) begin //synchronous reset
        state <= STATE2; xo <= 0; yo <= 0; clock_count <= 0;
    end else
    (* PARALLEL_CASE, FULL_CASE *) case (state)//attributes
        STATE1 : begin //idle state
            state <= STATE1; xo <= 0; yo <= 0;
        end
        STATE2 : begin //input/ load weights
            if (clock_count < 4) begin
                state <= STATE2;
                clock_count <= clock_count + 1;
                xo <= xi; yo <= 0;
            end else begin
                state <= STATE3;
                w <= xi; xo = xi; yo <= 0;
            end
        end
        STATE3 : begin //main operations
            state <= STATE4;
            xo <= xi;
        end
    endcase
end

y <= yi + w * xi;
end
STATE4 : begin //for delay
    state <= STATE3;
    xo <= xi;
    yo <= y;
end
default : begin //fault Recovery
    state <= STATE1;
    $display (%0t State error occurred!, $time );//for debugging
end
endcase
end //always
endmodule
Algorithm (Network)

Connect \( \{PE(k)|k \in [0, 2]\} \).

- input: \( x \).
- output: \( yo(2) \).

1. for each clock tick, do the following:

\[
xo(-1) \leftarrow x, \quad xi(k) \leftarrow xo(k - 1), \quad yi(k) \leftarrow yo(k - 1), \quad k \in [0, 2].
\]
Listing 12: Module: network.v

```
	`timescale 1ns/100ps  //unit time/ precision
	`define DATA_WIDTH 32  //parameter

//network for connecting PEs
module network(
    input signed ['DATA_WIDTH-1:0] xi,  //input
    output signed ['DATA_WIDTH-1:0] yo,  //output
    input clock,  //clock input
    input reset  //reset input
);

//nets and variables
wire signed ['DATA_WIDTH-1:0] xo;  //actually dummy
wire signed['DATA_WIDTH*3:1] t,s;  //connection nets

//chain of instances
pe p[1:4] ({t,xi}, {xo,t}, {s,0}, {yo,s},clock,reset);
endmodule
```
Listing 13: Module: `tb.v`

```vhdl
`timescale 1 ns / 100 ps //unit time/ precision
`define DATA_WIDTH 32 //parameter

//a test bench
module tb; //test bench

//inputs
reg signed[`DATA_WIDTH-1:0] x;
reg clock, reset;

//outputs
wire signed [`DATA_WIDTH-1:0] y;

//instantiate UUT
network UUT ( //supply x and receive y
 .xi(x),
 .yo(y),
 .clock(clock),
 .reset(reset)
);

//execute once for simulation
initial begin
  x = 0; clock = 0; reset = 0; //initialize inputs
  #100 reset = 1; //generate reset
  #100 reset = 0;

```

Hong Jeong (©2014 John Wiley & Sons Singapore Pte Ltd.)

Architectures for Computer Vision
end
always #50 clock = ~clock; //clock generation
always @(posedge clock) begin //generates weights
if (reset == 1) assign x = 0; //assign-deassign
else x = $random; //random number used here
deassign x; //assign-deassign
end
endmodule
**Figure:** Simulation result.
Figure: RTL schematic diagram after synthesis.