차세대 반도체 디지털 혁신공유대학 사업단

POLARIS

# On-chip buffer, block RAM

2022.02.21 (Mon)

# Road map

Review

Verilog HDL

On-chip Buffers

# Format of the binary file for weights

- What is the format for the binary file?
- Answers:
  - Format
    - Header:
    - Parameter for the convolutional layers saved layer by layer
    - Each CNN layer
      - Biases
      - For batch normalization
        - Scale, mean, variance
      - Filters
  - The code for loading a weight file is in load_weights_upto_cpu ([additionally.c])

```
3665        network net = parse_network_cfg(cfgfile, 1, quantized);
3666        //parse_network_cfg_custom(cfgfile, 1);    // set batch=1
3667        if (weightfile) {
3668            load_weights_upto_cpu(&net, weightfile, net.n);
3669        }
3670        //set_batch_network(&net, 1);
3671        yolov2_fuse_conv_batchnorm(net);
```

# load_weights_upto_cpu (additionally.c)

```
2638    // parser.c
2639    void load_weights_upto_cpu(network *net, char *filename, int cutoff)
2640    {
2641        fprintf(stderr, "Loading weights from %s...", filename);
2642        fflush(stdout);
2643        FILE *fp = fopen(filename, "rb");
2644        if (!fp) file_error(filename);
2645
2646        int major;
2647        int minor;
2648        int revision;
2649        fread(&major, sizeof(int), 1, fp);
2650        fread(&minor, sizeof(int), 1, fp);
2651        fread(&revision, sizeof(int), 1, fp);
2652        if ((major * 10 + minor) >= 2) {
2653            fread(net->seen, sizeof(uint64_t), 1, fp);
2654        }
2655        else {
2656            int iseen = 0;
2657            fread(&iseen, sizeof(int), 1, fp);
2658            *net->seen = iseen;
2659        }
2660        //int transpose = (major > 1000) || (minor > 1000);
2661
2662        int i;
2663        for (i = 0; i < net->n && i < cutoff; ++i) {
2664            layer l = net->layers[i];
2665            if (l.dontload) continue;
2666            if (l.type == CONVOLUTIONAL) {
2667                load_convolutional_weights_cpu(l, fp);
2668            }
2669        }
2670        fprintf(stderr, "Done!\n");
2671        fclose(fp);
2672    }
```

Parse the header

Parameters for layers

There are 20 bytes for the header

yolov3-tiny-aix2022.weights

| Address | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00000000 | 00 | 00 | 00 | 00 | 02 | 00 | 00 | 00 | 05 | 00 | 00 | 00 | 00 | 06 | 31 | 00 |
| 00000010 | 00 | 00 | 00 | 00 | a1 | aa | 46 | bf | 63 | ce | 1b | 3f | 14 | ad | 53 | 3c |
| 00000020 | ca | eb | 9d | 3e | 59 | 3c | 96 | 3e | dd | 7c | be | 3e | b8 | f8 | 44 | bf |
| 00000030 | b7 | c5 | 09 | 3f | ea | cc | b7 | 3e | 9b | 32 | 7b | 3e | ce | a1 | 4a | 3f |
| 00000040 | 31 | 32 | 13 | bf | c1 | 56 | 21 | 3f | 1f | 15 | be | 3e | 53 | 40 | 79 | bc |
| 00000050 | 7c | ba | 8b | 3e | 07 | 63 | 42 | 3f | 8f | cd | 27 | 3f | ed | 73 | ab | 3e |
| 00000060 | 3d | 2a | 30 | 3f | bb | 39 | 23 | 3f | 96 | da | a3 | 3f | 29 | 7d | 4e | 3f |
| 00000070 | 7d | 98 | be | 3f | bc | f4 | 7f | 3f | b5 | 20 | 38 | 3f | b2 | f6 | 36 | 3f |
| 00000080 | a9 | 6d | 16 | 3f | 52 | 09 | 29 | 3f | 25 | c8 | a8 | 3f | 78 | 9c | 9f | 3f |
| 00000090 | d5 | 4b | 8c | 3f | 8b | c5 | ad | be | fe | 28 | 8f | 3d | 8e | 5b | 39 | 3e |
| 000000a0 | 2e | 7f | 3b | bc | 24 | 93 | d7 | bd | af | c9 | a2 | bc | b8 | f3 | 68 | 3e |

# load_convolutional_weights_cpu

- Parameters for convolutional layers
- Biases[num_filters], where num_filters is the number of filter sets (e.g., 16 for layer 0)
- Example:
  - biases[0] = bf46_aaa1 or -0.776041090
  - biases[1] = 3f1b_ce63 or 0.608617961
- If batch normalization
  - scales[num_filters]
  - rolling_mean[num_filters]
  - rolling_variance[num_filters]
- Weights[n*c*size*size]
  - n: Number of filter sets, e.g., 16 for Layer 0
  - c: Number of input channels, e.g., 3 for Layer 0
  - size: filter size, e.g., 3 for Layer 0

```
2625    // parser.c
2626    void load_convolutional_weights_cpu(layer l, FILE *fp)
2627    {
2628        int num = l.n*l.c*l.size*l.size;
2629        fread(l.biases, sizeof(float), l.n, fp);
2630        if (l.batch_normalize && (!l.dontloadscales)) {
2631            fread(l.scales, sizeof(float), l.n, fp);
2632            fread(l.rolling_mean, sizeof(float), l.n, fp);
2633            fread(l.rolling_variance, sizeof(float), l.n, fp);
2634        }
2635        fread(l.weights, sizeof(float), num, fp);
2636    }
```

yolov3-tiny-aix2022.weights

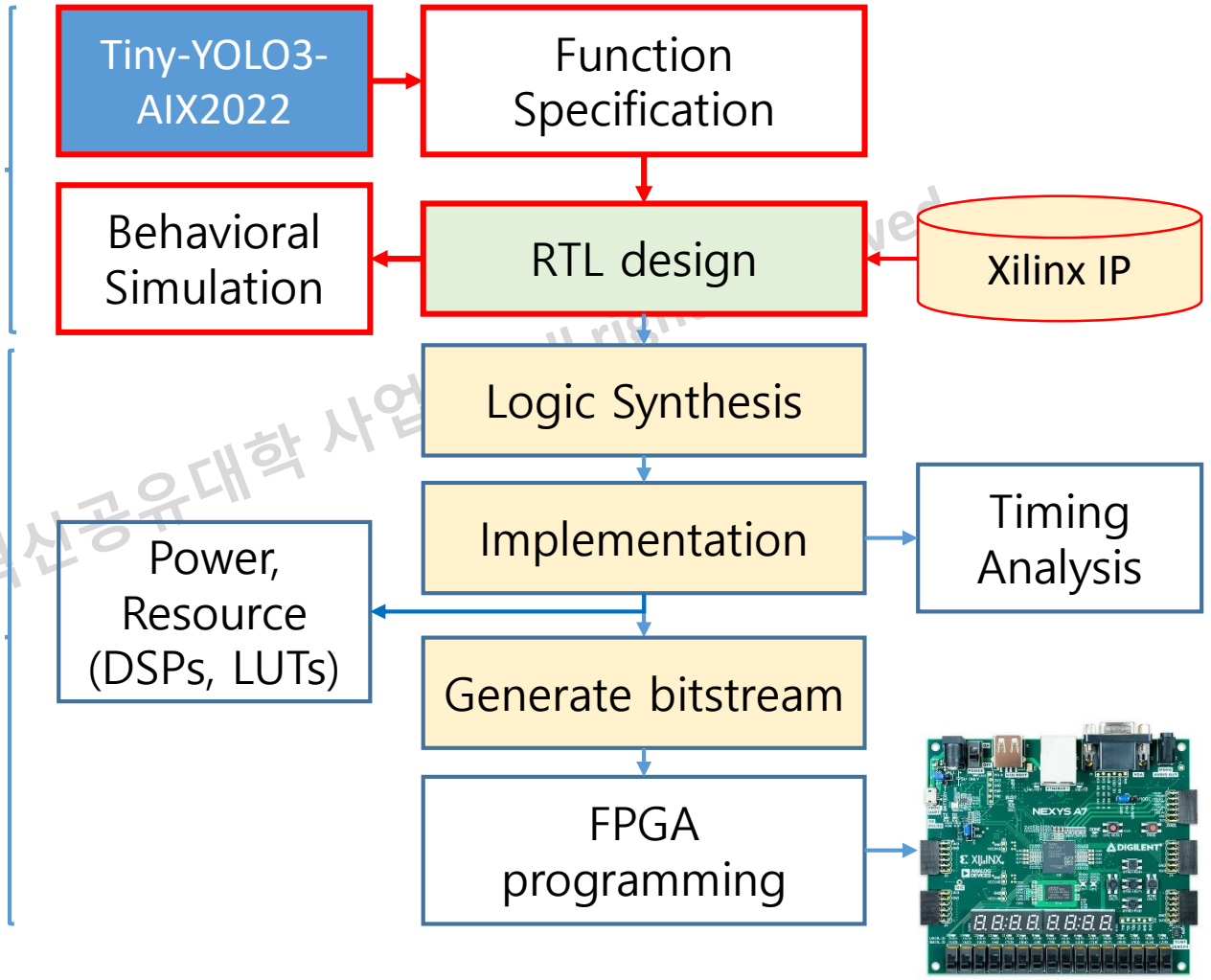
| Address | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00000000 | 00 | 00 | 00 | 00 | 02 | 00 | 00 | 00 | 05 | 00 | 00 | 00 | 00 | 06 | 31 | 00 | |
| 00000010 | 00 | 00 | 00 | 00 | a1 | aa | 46 | bf | 63 | ce | 1b | 3f | 14 | ad | 53 | 3c | |
| 00000020 | ca | eb | 9d | 3e | 59 | 3c | 96 | 3e | dd | 7c | be | 3e | b8 | f8 | 44 | bf | |
| 00000030 | b7 | c5 | 09 | 3f | ea | cc | b7 | 3e | 9b | 32 | 7b | 3e | ce | a1 | 4a | 3f | Biases |
| 00000040 | 31 | 32 | 13 | bf | c1 | 56 | 21 | 3f | 1f | 15 | be | 3e | 53 | 40 | 79 | bc | |
| 00000050 | 7c | ba | 8b | 3e | 07 | 63 | 42 | 3f | 8f | cd | 27 | 3f | ed | 73 | ab | 3e | |
| 00000060 | 3d | 2a | 30 | 3f | bb | 39 | 23 | 3f | 96 | da | a3 | 3f | 29 | 7d | 4e | 3f | Scales |
| 00000070 | 7d | 98 | be | 3f | bc | f4 | 7f | 3f | b5 | 20 | 38 | 3f | b2 | f6 | 36 | 3f | |
| 00000080 | a9 | 6d | 16 | 3f | 52 | 09 | 29 | 3f | 25 | c8 | a8 | 3f | 78 | 9c | 9f | 3f | |
| 00000090 | d5 | 4b | 8c | 3f | 8b | c5 | ad | be | fe | 28 | 8f | 3d | 8e | 5b | 39 | 3e | |
| 000000a0 | 2e | 7f | 3b | bc | 24 | 93 | d7 | bd | af | c9 | a2 | bc | b8 | f3 | 68 | 3e | Mean |
| 000000b0 | e2 | e5 | 49 | bc | b0 | ec | e0 | 3b | 00 | 5c | 40 | bc | 9e | f0 | 2b | bb | |
| 000000c0 | c6 | ad | 86 | bf | 29 | 7c | 68 | bc | 8d | 64 | 67 | bc | 78 | 94 | f9 | 3a | |
| 000000d0 | a3 | c7 | 58 | bd | 1b | 19 | d4 | 3c | 2d | d5 | 55 | 3d | ae | 6e | 48 | 3c | |
| 000000e0 | bb | 1b | 68 | 3d | 01 | cd | 11 | 3d | 66 | af | cf | 3d | 1d | 03 | 13 | 3c | Variance |
| 000000f0 | 0b | cb | 5d | 3e | 7a | db | 56 | 3d | d0 | b3 | 35 | 3d | f0 | 05 | e1 | 3b | |
| 00000100 | 96 | 00 | 4a | 3e | 37 | 14 | a8 | 3b | 59 | 3b | 0c | 3e | e5 | 8a | 28 | 3e | |
| 00000110 | 7c | 92 | 37 | 3d | 38 | 6a | a8 | 3d | e3 | 97 | 41 | 3e | f5 | 7b | 41 | be | |
| 00000120 | e5 | 03 | b9 | 3d | eb | 07 | de | bd | 61 | 96 | e9 | be | 25 | 06 | 61 | 3e | Weights |

# FPGA design flow

Function simulation
Text editor: *vim, notepad++, VS code*
RTL simulator: *ModelSim, ISE, Vivado*

FPGA Implementation (*ISE, Vivado*)
- Synthesize the design
- Implementation: mapping, placement and routing

Optimization
- Analyze timing, power, resources

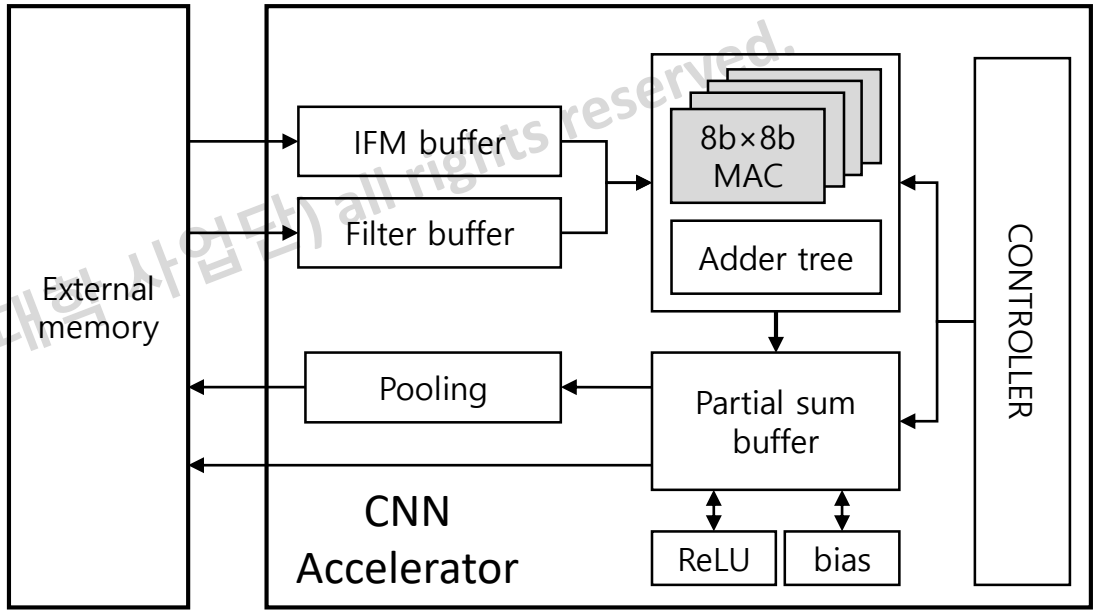


# Previous labs

- Lab 1: DSP
  - How to use Vivado IP integrator to generate a DSP?
  - How to test a DSP unit?
- Lab 2: Multiplication and accumulation (MAC)
  - Multipliers and an adder tree
  - MAC
- Lab 3: Computing units
  - Use MAC to do a convolutional layer

# Inference engine

- Processing Element (PE): MAC, adder tree
- Buffers
  - Input feature map (IFM)
  - Filters
  - Intermediate results or partial sum
- Pooling
- Controller



# Convolutional layer (cnv_tb.v)

- There are four MAC instances
- Each module:
  - Uses one set of convolutional filters, e.g., win
  - Outputs a specific acc_o

```
20  mac u_mac_00(
21  ./*input          */clk(clk),
22  ./*input          */rstn(rstn),
23  ./*input          */vld_i(vld_i),
24  ./*input [127:0]  */win(win[0]),
25  ./*input [127:0]  */din(din),
26  ./*output[ 19:0]  */acc_o(acc_o[0]),
27  ./*output         */vld_o(vld_o[0])
28  );
29  mac u_mac_01(
30  ./*input          */clk(clk),
31  ./*input          */rstn(rstn),
32  ./*input          */vld_i(vld_i),
33  ./*input [127:0]  */win(win[1]),
34  ./*input [127:0]  */din(din),
35  ./*output[ 19:0]  */acc_o(acc_o[1]),
36  ./*output         */vld_o(vld_o[1])
37  );
```

```
38  mac u_mac_02(
39  ./*input          */clk(clk),
40  ./*input          */rstn(rstn),
41  ./*input          */vld_i(vld_i),
42  ./*input [127:0]  */win(win[2]),
43  ./*input [127:0]  */din(din),
44  ./*output[ 19:0]  */acc_o(acc_o[2]),
45  ./*output         */vld_o(vld_o[2])
46  );
47  mac u_mac_03(
48  ./*input          */clk(clk),
49  ./*input          */rstn(rstn),
50  ./*input          */vld_i(vld_i),
51  ./*input [127:0]  */win(win[3]),
52  ./*input [127:0]  */din(din),
53  ./*output[ 19:0]  */acc_o(acc_o[3]),
54  ./*output         */vld_o(vld_o[3])
55  );
```

# Convolutional layer (cnv_tb.v)

Read the hex file into a buffer

```
68  // Read the input file to memory
69  initial begin
70      $readmemh(INFILE, in_img ,0,FRAME_SIZE-1);
71  end
72  initial begin
73      rstn = 1'b0;            // Reset, low active
74      vld_i= 0;
75      din = 0;
76      i = 0;
77
```

```
1  2a
2  45
3  5b
4  63
5  6a
6  6c
7  6f
8  6d
```

Initialize four set of convolutional filters

```
78      // CNN filters of four output channels
79      win[0][  7:  0] = 8'd142; win[1][  7:  0] = 8'd69 ; win[2][  7:  0] = 8'd13 ; win[3][  7:  0] = 8'd69 ;
80      win[0][ 15:  8] = 8'd151; win[1][ 15:  8] = 8'd181; win[2][ 15:  8] = 8'd244; win[3][ 15:  8] = 8'd135;
81      win[0][ 23: 16] = 8'd215; win[1][ 23: 16] = 8'd209; win[2][ 23: 16] = 8'd255; win[3][ 23: 16] = 8'd235;
82      win[0][ 31: 24] = 8'd127; win[1][ 31: 24] = 8'd19 ; win[2][ 31: 24] = 8'd241; win[3][ 31: 24] = 8'd128;
83      win[0][ 39: 32] = 8'd163; win[1][ 39: 32] = 8'd128; win[2][ 39: 32] = 8'd127; win[3][ 39: 32] = 8'd32 ;
84      win[0][ 47: 40] = 8'd205; win[1][ 47: 40] = 8'd95 ; win[2][ 47: 40] = 8'd240; win[3][ 47: 40] = 8'd90 ;
85      win[0][ 55: 48] = 8'd229; win[1][ 55: 48] = 8'd221; win[2][ 55: 48] = 8'd252; win[3][ 55: 48] = 8'd48 ;
86      win[0][ 63: 56] = 8'd255; win[1][ 63: 56] = 8'd121; win[2][ 63: 56] = 8'd237; win[3][ 63: 56] = 8'd52 ;
87      win[0][ 71: 64] = 8'd113; win[1][ 71: 64] = 8'd8  ; win[2][ 71: 64] = 8'd1  ; win[3][ 71: 64] = 8'd211;
88      win[0][ 79: 72] = 8'd0  ; win[1][ 79: 72] = 8'd0  ; win[2][ 79: 72] = 8'd0  ; win[3][ 79: 72] = 8'd0  ;
89      win[0][ 87: 80] = 8'd0  ; win[1][ 87: 80] = 8'd0  ; win[2][ 87: 80] = 8'd0  ; win[3][ 87: 80] = 8'd0  ;
90      win[0][ 95: 88] = 8'd0  ; win[1][ 95: 88] = 8'd0  ; win[2][ 95: 88] = 8'd0  ; win[3][ 95: 88] = 8'd0  ;
91      win[0][103: 96] = 8'd0  ; win[1][103: 96] = 8'd0  ; win[2][103: 96] = 8'd0  ; win[3][103: 96] = 8'd0  ;
92      win[0][111:104] = 8'd0  ; win[1][111:104] = 8'd0  ; win[2][111:104] = 8'd0  ; win[3][111:104] = 8'd0  ;
93      win[0][119:112] = 8'd0  ; win[1][119:112] = 8'd0  ; win[2][119:112] = 8'd0  ; win[3][119:112] = 8'd0  ;
94      win[0][127:120] = 8'd0  ; win[1][127:120] = 8'd0  ; win[2][127:120] = 8'd0  ; win[3][127:120] = 8'd0  ;
```

⇒ Come from a quantized model

# Objectives

- Verilog
  - Dataflow modelling
  - Assignment
  - Data types
  - Operations
- Labs
  - Memory
    - Single-port and dual-port RAM
    - Use Vivado IP generator to generate a block ram
      - Simulation
      - FPGA: COE format and ram initialization

# Road map

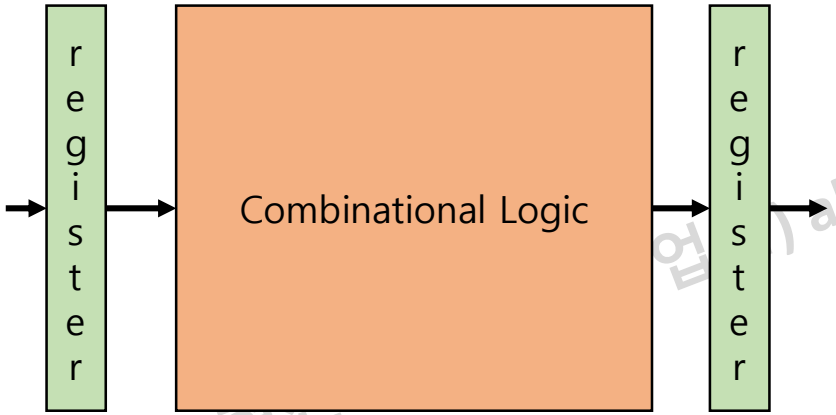
Review

Verilog HDL

On-chip buffers

# Dataflow modelling

- Any digital system can be constructed by interconnecting registers and a combinational logic put between them for performing the necessary functions.




  - Dataflow provides a powerful way to implement a design.
  - Logic synthesis tools can be used to create a gate-level circuit from a dataflow design description.
  - RTL (register transfer level) is a combination of dataflow and behavioral modeling.

# Assignments

- Two basic forms of assignments
  - Continuous assignment: assign values to **nets**
  - Procedural assignment: assign values to **variables**
- **Example**
  - Continuous assignment: the most basic statement of dataflow modeling.
  - A continuous assignment begins with the keyword assign.

```
assign net_lvalue = expression;

assign net1 = expr1,
       net2 = expr2,
       ...,
       netn = exprn;
```

# Continuous Assignments

- An implicit continuous assignment
  - is the shortcut of declaring a net first and then writing a continuous assignment on the net.
  - is always active.
  - can only have one implicit declaration assignment per net.

```
wire out;                    // net declaration
assign out = in1 & in2; // regular continuous assignment

wire  out = in1 & in2;    // implicit continuous assignment
```

# Data types

- Two classes of data types:
  - nets: Nets mean any hardware connection points.
  - variables: Variables represent any data storage elements.

- Variable data types
  - reg
  - integer
  - time
  - real
  - realtime

# Variable data types

- A reg variable
  - holds a value between assignments.
  - may be used to model hardware registers.
  - need not actually represent a hardware storage element.

```
reg a, b;              // reg a, and b are 1-bit reg
reg [7:0] data_a;      // an 8-bit reg, the msb is bit 7
reg [0:7] data_b;      // an 8-bit reg, the msb is bit 0
reg signed [7:0] d; // d is an 8-bit signed reg
```

- The integer variable
  - contains integer values.
  - has at least 32 bits.
  - is treated as a signed reg variable with the least-significant bit (LSB) being bit 0.

```
integer i,j;           // declare two integer variables
integer data[7:0]; // array of integer
```

# The time, real, and realtime variables

- The time variable
  - is used for storing and manipulating simulation time quantities.
  - is typically used in conjunction with the $time system task.
  - holds only unsigned value and is at least 64 bits, with the LSB being bit 0.

```
time events;         // hold one time value
time current_time; // hold one time value
```

- The real and realtime variables
  - cannot use range declaration and their initial values are defaulted to zero (0.0).

```
real events;              // declare a real variable
realtime current_time; // hold current time as real
```

# Vectors

- A vector (multiple bit width) describes a bundle of signals as a basic unit.
  - [high:low] or [low:high]
  - The leftmost bit is the MSB.
  - Both nets and reg data types can be declared as vectors.
- The default is 1-bit vector or called scalar.

- Bit-Select and Part-Select
  - integer and time can also be accessed by bit-select or part-select.
  - real and realtime are not allowed to be accessed by bit-select or part-select.
  - Constant part select: data_bus[3:0], bus[3]
  - Variable part select:

    [<starting_bit>+:width]: data_bus[8+:8]

    [<starting_bit>-:width]: data_bus[15-:8]

# Array and memory elements

- Array and Memory Elements
  - all net and variable data types are allowed to be declared as **multi-dimensional** arrays.
  - an array element can be a scalar or a vector if the element is a net or reg data type.

```
wire a[3:0];            // a scalar wire array of 4 elements
reg  d[7:0];            // a scalar reg array of 8 elements
wire [7:0]  x[3:0];     // an 8-bit wire array of 4 elements
reg  [31:0] y[15:0];    // a 32-bit reg array of 16 elements
integer states [3:0];   // an integer array of 4 elements
time    current[5:0];   // a time array of 6 elements
```

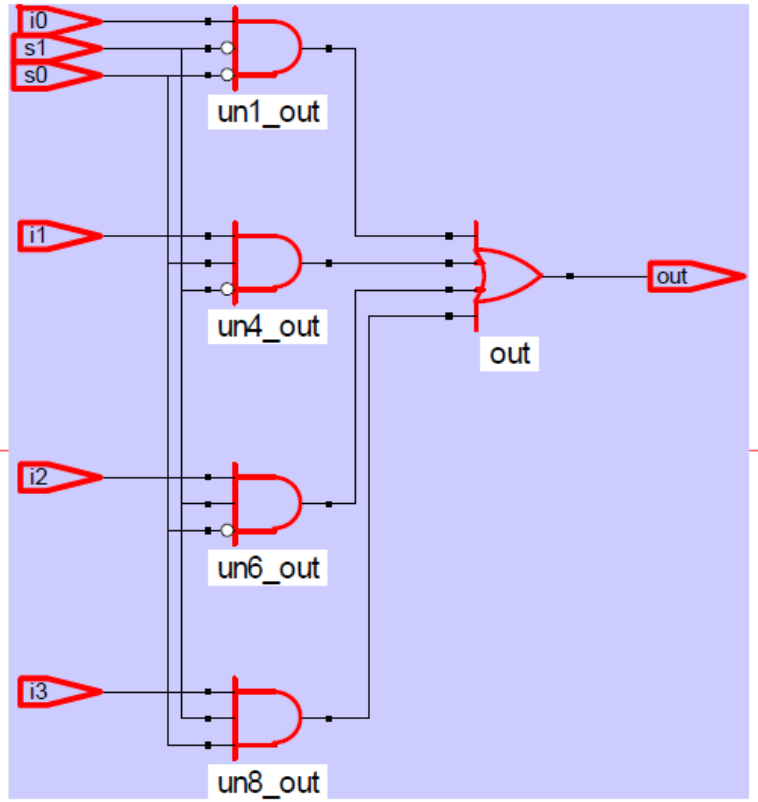
# Bitwise operations

- Bitwise operators
  - They perform a bit-by-bit operation on two operands.
  - A z is treated as x in bit-wise operation.
  - The shorter operand is zero-extended to match the length of the longer operand.

| Symbol | Operation |
|---|---|
| ~ | Bitwise negation |
| & | Bitwise and |
| \| | Bitwise or |
| ^ | Bitwise exclusive or |
| ~^, ^~ | Bitwise exclusive nor |

# Example: 4-to-1 MUX

```
module mux41_dataflow(i0, i1, i2, i3, s1, s0, out);
// Port declarations
input i0, i1, i2, i3;
input s1, s0;
output out;
// Using basic and, or , not logic operators.
assign out =    (~s1 & ~s0 & i0) |
                (~s1 & s0 & i1)  |
                (s1 & ~s0 & i2)  |
                (s1 & s0 & i3) ;
endmodule
```



# Arithmetic Operators

- Arithmetic operators
  - If any operand bit has a value x, then the result is x.
  - The operators + and –can also used as unary operators to represent signed numbers.
  - Modulus operators produce the remainder from the division of two numbers.
  - In Verilog HDL, 2's complement is used to represent negative numbers.

| Symbol | Operation |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| ** | Exponent (power) |
| % | Modulus |

# Concatenation and Replication operations

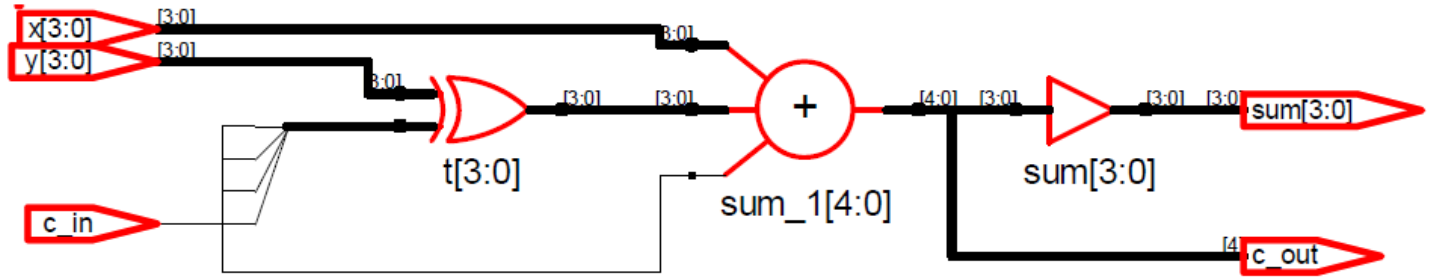
- Concatenation operators
  - The operands must be sized.
  - Operands can be scalar nets or registers, vector nets or registers, bit-select, part-select, or sized constants.
  - Example: y = {a, b[0], c[1]};

| Symbol | Operation |
|---|---|
| { , } | Concatenation |
| {const_expr{}} | Replication |

- Replication operators
  - They specify how many times to replicate the number inside the braces.
  - Example: y = {a, 4{b[0]}, c[1]};

# Example: A 4-bit two's complement adder

```
module twos_adder(x, y, c_in, sum, c_out);
// I/O port declarations
input    [3:0] x, y;    // declare as a 4-bit array
input    c_in;
output   [3:0] sum;   // declare as a 4-bit array
output   c_out;
wire     [3:0] t;       // outputs of xor gates

// Specify the function of a two's complement adder
   assign t = y ^ {4{c_in}};
   assign {c_out, sum} = x + t + c_in;
endmodule
```



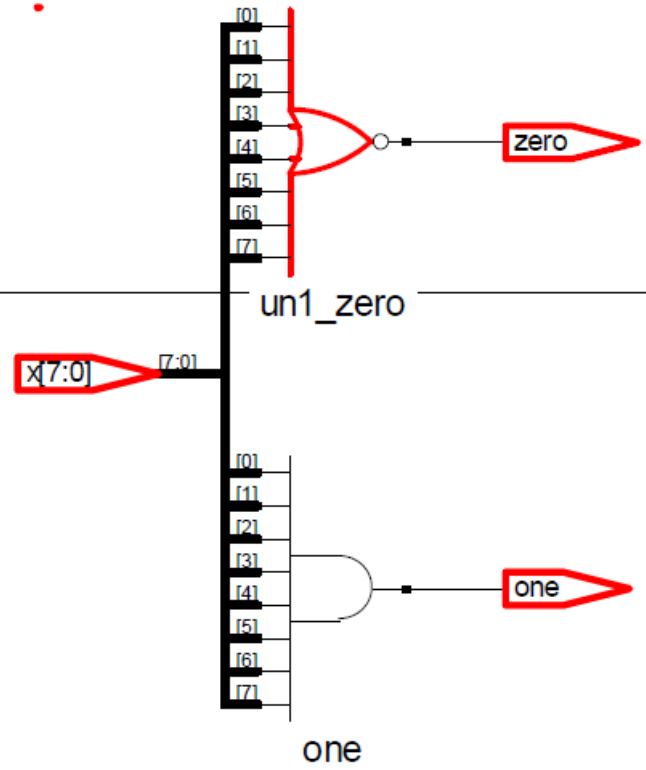
# Reduction operators

- Reduction operators
  - perform only on one vector operand.
  - carry out a bit-wise operation on a single vector operand and yield a 1-bit result.
  - work bit by bit from right to left.

| Symbol | Operation |
|---|---|
| & | Reduction and |
| ~& | Reduction nand |
| \| | Reduction or |
| ~\| | Reduction nor |
| ^ | Reduction exclusive or |
| ~^, ^~ | Reduction exclusive nor |

# Example: An All-Bit-Zero/One detector

```
module all_bit_01_detector_reduction(x, zero,one);
// I/O port declarations
input    [7:0] x;
output   zero, one;
// dataflow modeling
   assign zero = ~(|x);  // all-bit zero detector
   assign one = &x;      // all-bit one detector
endmodule
```



# Logical, relational Operators

- Logical operators
  - They always evaluate to a 1-bit value, 0, 1, or x.
  - If any operand bit is x or z, it is equivalent to x and treated as a false condition by simulators.

| Symbol | Operation |
|---|---|
| ! | Logical negation |
| && | Logical and |
| \|\| | Logical or |

- Relational operators
  - They return logical value 1 if the expression is true and 0 if the expression is false.
  - The expression takes a value x if there are any unknown (x) or z bits in the operands.

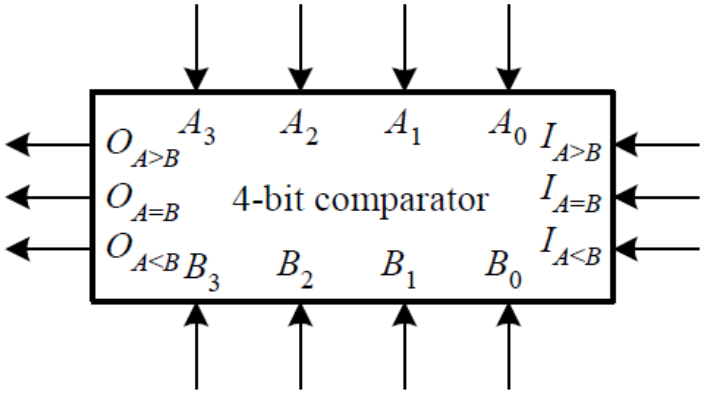
| Symbol | Operation |
|---|---|
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |

# Equality operations

- Equality operators
  - compare the two operands bit by bit, with zero filling if the operands are of unequal length.
  - return logical value 1 if the expression is true and 0 if the expression is false.
- The operators (==, !=) yield an x if either operand has x or z in its bits.
- The operators (===, !==) yield a 1 if the two operands match exactly and 0 if the two operands not match exactly.

| Symbol | Operation |
|---|---|
| == | Logical equality |
| != | Logical inequality |
| === | Case equality |
| !== | Case inequality |

# Example: A 4-b Magnitude Comparator




```
module four_bit_comparator(Iagtb, Iaeqb, Ialtb, a, b, Oagtb, Oaeqb, Oaltb);
// I/O port declarations
input    [3:0] a, b;
input    Iagtb, Iaeqb, Ialtb;
output  Oagtb, Oaeqb, Oaltb;
// dataflow modeling using relation operators
   assign Oaeqb = (a == b) && (Iaeqb == 1);  // equality
   assign Oagtb = (a > b) || ((a == b)&& (Iagtb == 1));  // greater than
   assign Oaltb = (a < b) || ((a == b)&& (Ialtb == 1));   // less than
endmodule
```

# Shift operators

- Logical shift operators
  - >> operator: logical right shift
  - << operator: logical left shift
  - The vacant bit positions are filled with zeros.
- Arithmetic shift operators
  - >>> operator: arithmetic right shift
  - The vacant bit positions are filled with the MSBs (sign bits).
  - <<< operator: arithmetic left shift
  - The vacant bit positions are filled with zeros.

| Symbol | Operation |
|---|---|
| >> | Logical right shift |
| << | Logical left shift |
| >>> | Arithmetic right shift |
| <<< | Arithmetic left shift |

# Example: Shift operators

```
// example to illustrate logic and arithmetic shifts
module arithmetic_shift(x,y,z);
input  signed [3:0] x;
output [3:0] y;
output signed [3:0] z;
assign y = x >> 1;   // logical right shift
assign z = x >>> 1; // arithmetic right shift
endmodule
```

Note that: net variables x and z must be declared with the keyword signed. Replaced net variable with unsigned net (i.e., remove the keyword signed) and see what happens.

# The Conditional Operator

- Conditional Operator

  Usage: condition_expr ? true_expr: false_expr;

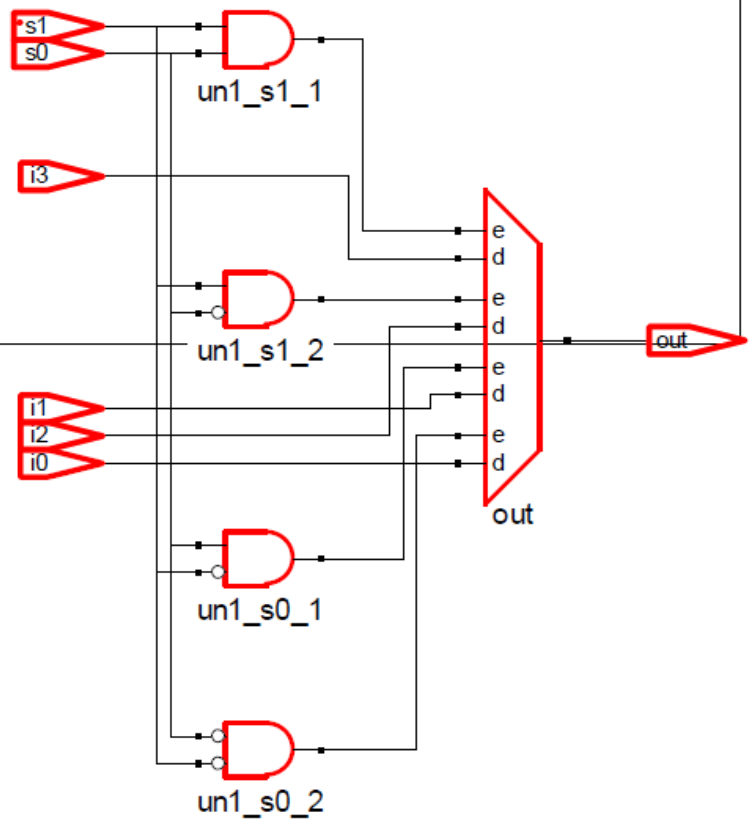
  - The condition_expr is evaluated first.
  - If the result is true then the true_expr is executed; otherwise the false_expr is evaluated.
    - if (condition_expr) true_expr;
    - else false_expr;

- a 2-to-1 multiplexer

```
assign out = selection ? in_1: in_0;
```

# Example: A 4-to-1 MUX

```
module mux4_to_1_cond (i0, i1, i2, i3, s1, s0, out);
// Port declarations from the I/O diagram
input   i0, i1, i2, i3;
input   s1, s0;
output out;
// Using conditional operator (?:)
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;
endmodule
```



# Road map

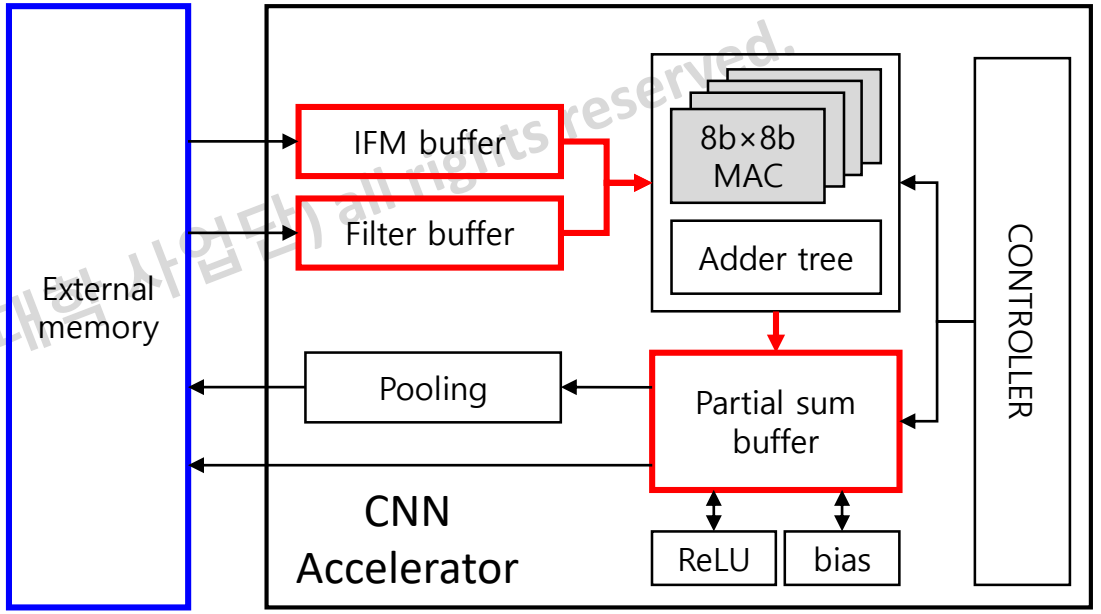
Review

Verilog HDL

On-chip Buffers

# Motivation

- On-chip buffers (memory)
  - Input feature map (IFM)
  - Filters
  - Intermediate results or partial sum
- External memory



# Example of filter, bias, scale files

- Three types of buffers
  - Filter/weight: 16 lines, each line has 128 bits.
  - Bias/scale: 16 lines, each line has 16 bits

conv_weight_L1.hex

conv_weights_L1.hex | cnn_accel.v | spram.v

```
1  0000000000000071FFE5CDA37FD7978E
2  0000000000000000879DD5F8013D1B545
3  0000000000000001EDFCF07FF1FFF40D
4  00000000000000D334305A2080EB8745
5  0000000000000033EF09D5FB8003CFF3
6  00000000000000FEEE02FC7FF6FAD2F8
7  00000000000000F880F7165702F4020D
8  00000000000000FB41CD8BB82B3480F3
9  0000000000000080AB00093CD20ED39F
10 00000000000000E2EE10AC4403C180CE
11 00000000000000FB0501FEBDF2F67F0D
12 0000000000000060EF6A77FF201FDFE
13 000000000000001F815BE67FD4F8800D
14 0000000000000193EDB928BAE1F3080
15 000000000000008BAB80F90D42F02400
16 000000000000000AF906E67FA8ED60C6
```

conv_bias_L1.hex

conv_biases_L1.hex

```
1  b744
2  1f82
3  0172
4  fe06
5  f911
6  f942
7  171d
8  f98c
9  066a
10 0199
11 f8b5
12 fd24
13 d332
14 e45d
15 0232
16 fe6b
```

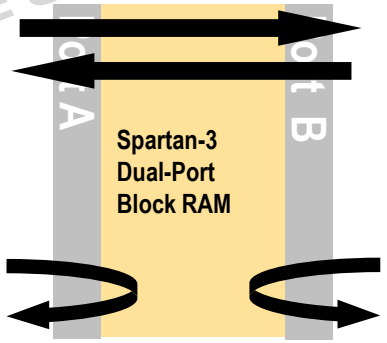
conv_scale_L1.hex

conv_scales_L1.hex

```
1  005f
2  0067
3  016c
4  00aa
5  007a
6  0136
7  00cb
8  0079
9  006b
10 00a0
11 0135
12 0107
13 004d
14 006a
15 00af
16 00ba
```

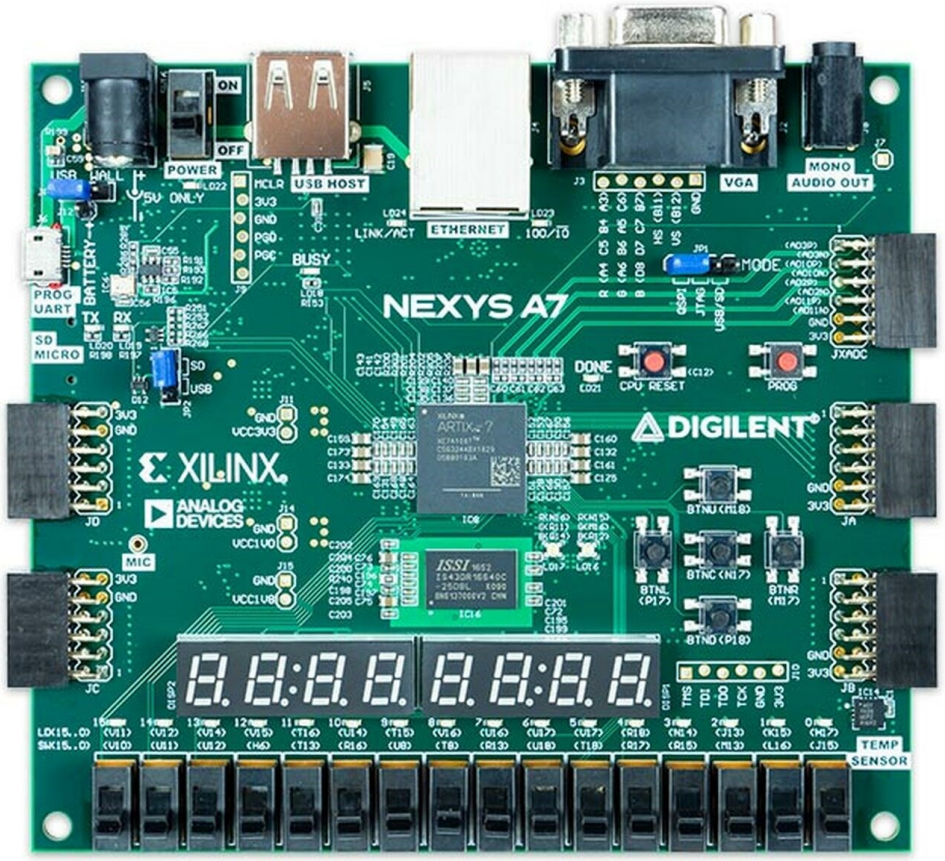
# Block RAM

- Most efficient memory implementation
  - Dedicated blocks of memory
- Ideal for most memory requirements
  - 4 to 104 memory blocks
    - **18 kbits = 18,432 bits per block (16 k without parity bits)**
  - Use multiple blocks for larger memories
- Builds both single and true dual-port RAMs
- Synchronous write and read (different from distributed RAM)



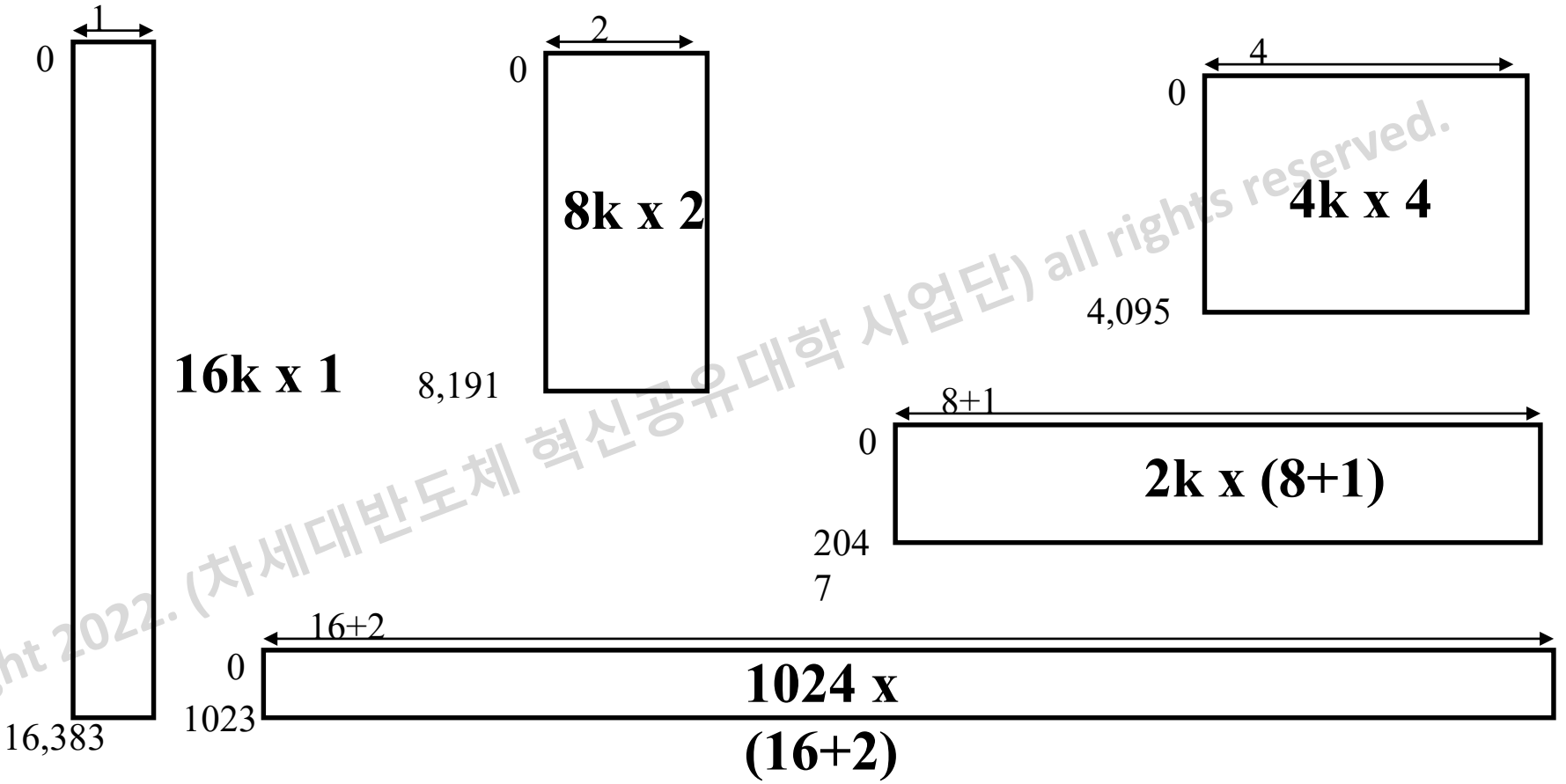
# Nexys A7 FPGA board

- Xilinx Artix-7 FPGA XC7A100T-1CSG324C
- 15,850 logic slices
  - Each with four 6-input LUTs and 8 FFs
- 4,860 Kbits of fast block RAM
  - 270 block RAMs
- 240 DSP slices (constrained to To, Ti)
  - Dedicated to multiplication and accumulation (MAC)
- Internal clock speeds exceeding 450 MHz
- 128 MB DDR2 Memory
- USB-JTAG port for FPGA programming and communication



# Block RAM Port Aspect Ratios

- Block RAM can have various configurations (port aspect ratios)



# Block RAM Port Aspect Ratios

- 18Kb BRAM can be configured
  - RAM_512×36 (512 words, each word has 36 bits).
  - RAM_1K×18 (1024 words, each word has 18 bits).
  - RAM_2K×9 (2048 words, each word has 9 bits).
- 16Kb BRAM can be configured
  - RAM_4K×4 (4096 words, each word has 4 bits).
  - RAM_8K×2 (8192 words, each word has 2 bits).
  - RAM_16K×1 (16384 words, each word has 1 bits).

| Organization | Memory Depth | Data Width | Parity Width | DI/DO | DIP/DOP | ADDR | Single-Port Primitive | Total RAM Kbits |
|---|---|---|---|---|---|---|---|---|
| 512x36 | 512 | 32 | 4 | (31:0) | (3:0) | (8:0) | RAMB16_S36 | 18K |
| 1Kx18 | 1024 | 16 | 2 | (15:0) | (1:0) | (9:0) | RAMB16_S18 | 18K |
| 2Kx9 | 2048 | 8 | 1 | (7:0) | (0:0) | (10:0) | RAMB16_S9 | 18K |
| 4Kx4 | 4096 | 4 | - | (3:0) | - | (11:0) | RAMB16_S4 | 16K |
| 8Kx2 | 8192 | 2 | - | (1:0) | - | (12:0) | RAMB16_S2 | 16K |
| 16Kx1 | 16384 | 1 | - | (0:0) | - | (13:0) | RAMB16_S1 | 16K |

# Lab 1: Single-port RAM

- Lab 1: Single-port RAM (spram): 16 bits per word, 6240 words
  - Use a single-port RAM
  - Generate a dedicated RAM using Vivado IP generator
  - Initialize a RAM using a coe file
  - Test bench

CONV20_W.coe

```
1  memory_initialization_radix=16;
2  memory_initialization_vector=
3  fe00fe01    ,
4  fd01ffff    ,
5  00000303    ,
6  00fdfe02    ,
7  0005ff00    ,
8  00fefd00    ,
9  0000fe00    ,
10 01000000    ,
11 fdff00fb    ,
12 01fefffb    ,
13 ffff0201    ,
14 00fa0001    ,
15 010400fd    ,
16 0500ff00    ,
```

# Spram and spram wrapper

- Ports
  - Clock (clka)
  - Read/Write enable (ena)
  - Write enable (we)
  - Address (addra)
  - Write data (dina)
  - Read data (douta)

- Two modes
  - Simulation
  - FPGA
    - Call a SRAM instance generated by IP generator (wrapper)
    - ***Optional: Initialize memory cell from a file***

# SPRAM: Simulation

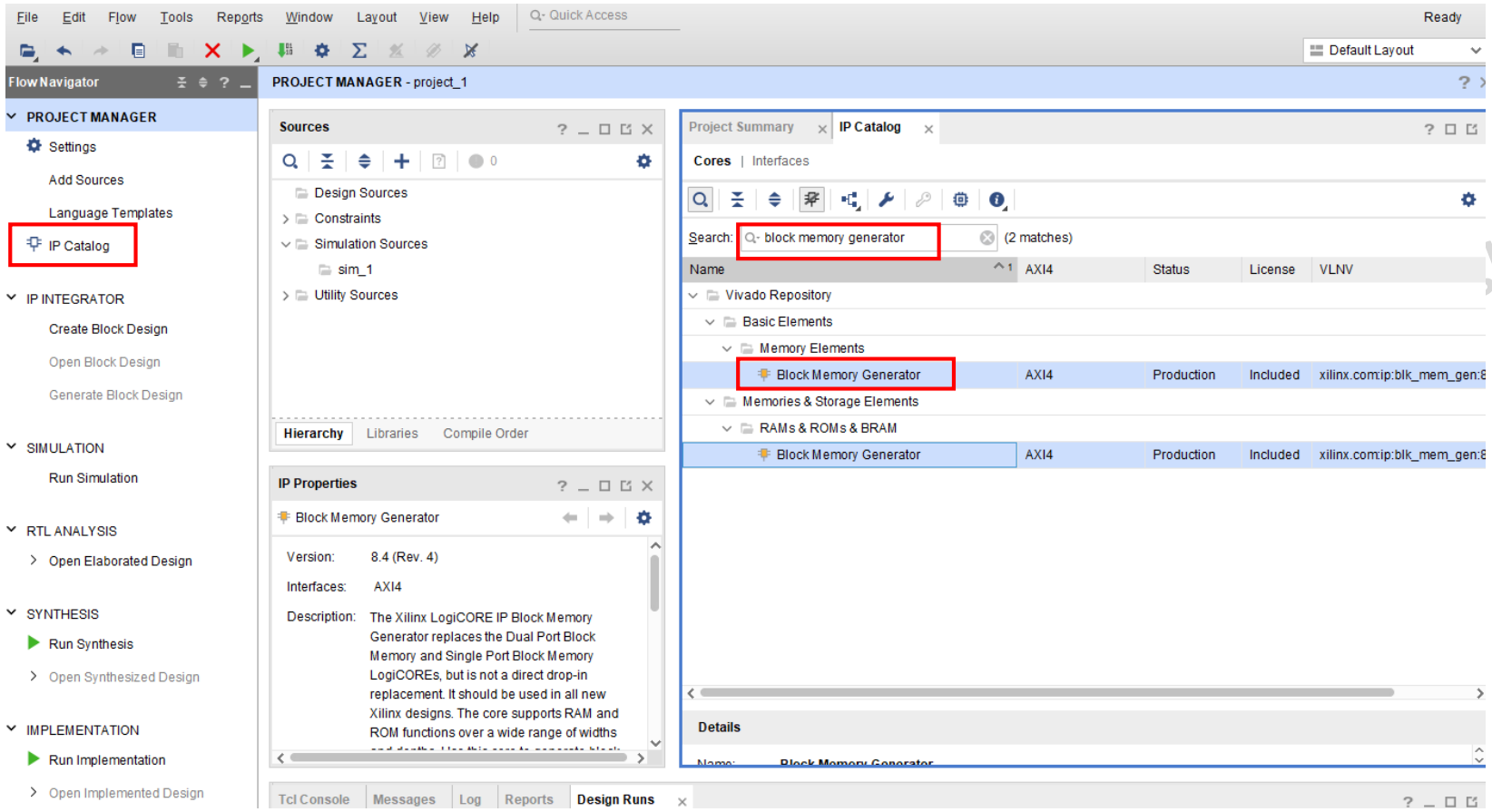
- Memory cell (mem)
  - Data width (DW)
    - Filter: 128, scale/bias: 16
  - Depth
    - Example: 16 lines
- Ports
  - Clock (clka)
  - Read/Write enable (ena)
  - Write enable (wea)
  - Address (addra)
  - Write data (dina)
  - Read data (douta)

- Read operation
  - If(ena)
    - douta <= mem[addra]

- Write operation
  - If(ena & wea)
    - mem[addra] <= dina

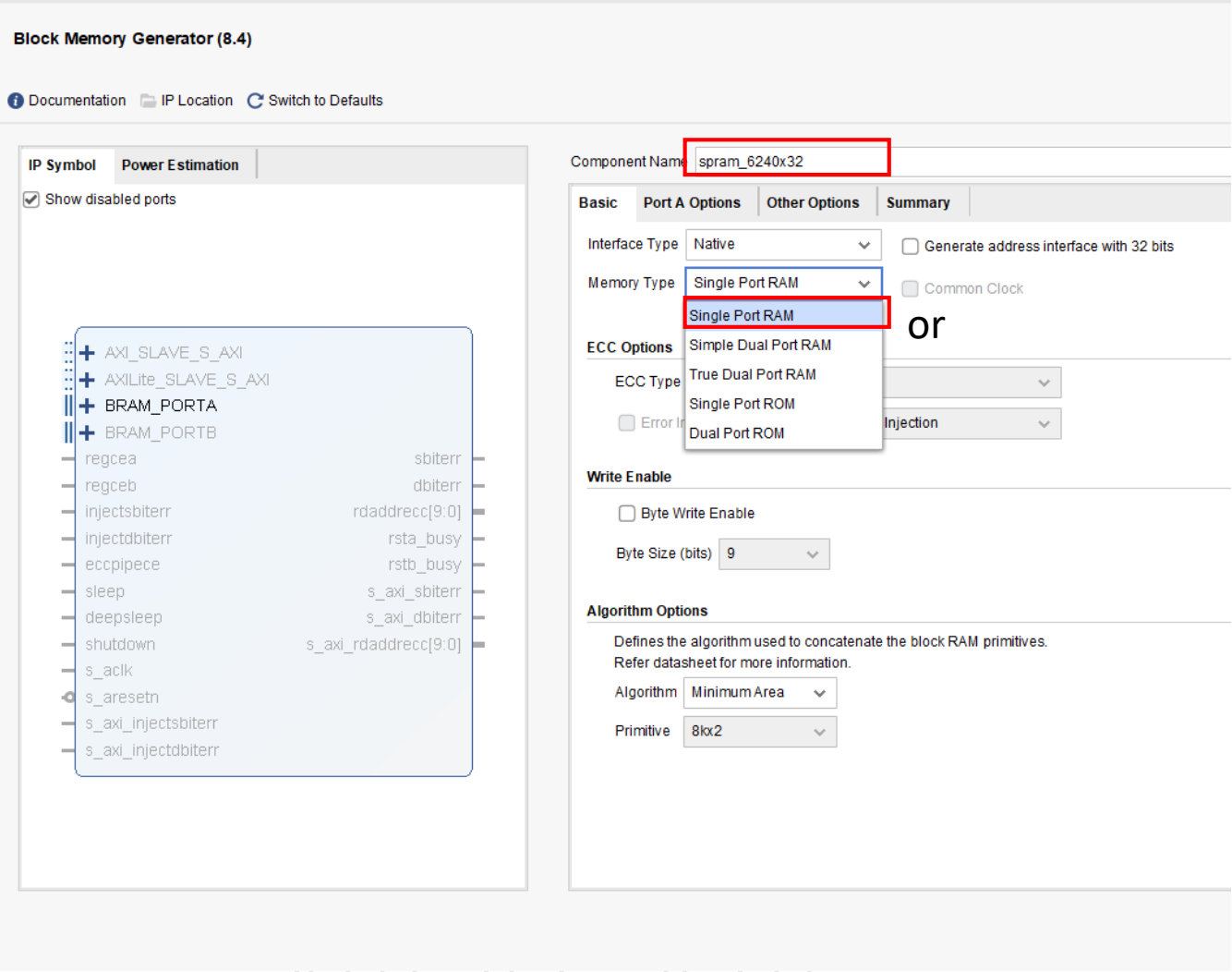
* Single-port
  - Read/wirte operations share "addra"
  - Only read or write occurs at a time

# FPGA: How to make BRAM IP?



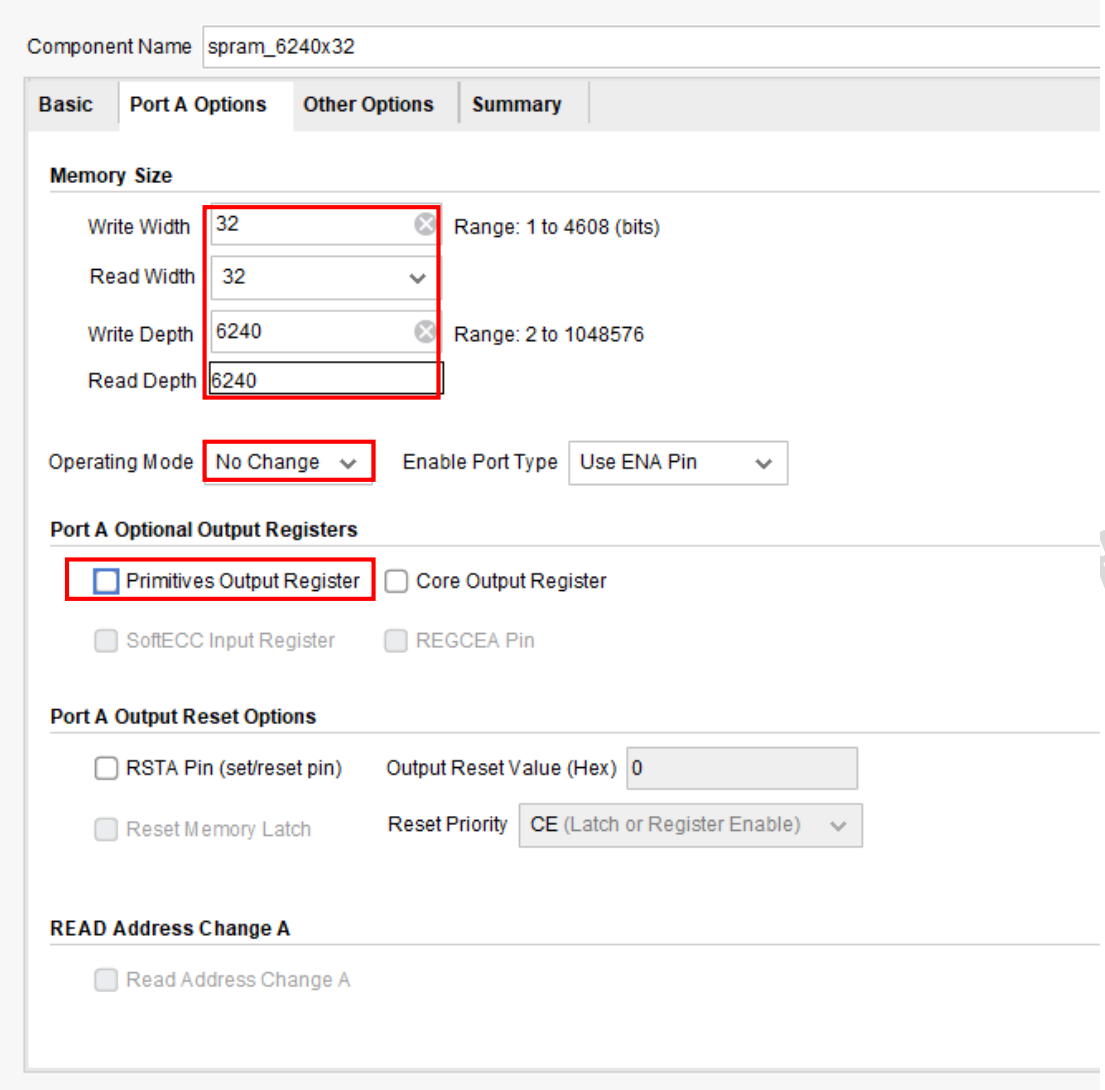

- 1. click IP catalog
- 2. search the "block memory generator"
- 3. double click the "block memory generator"

# How to make BRAM IP?



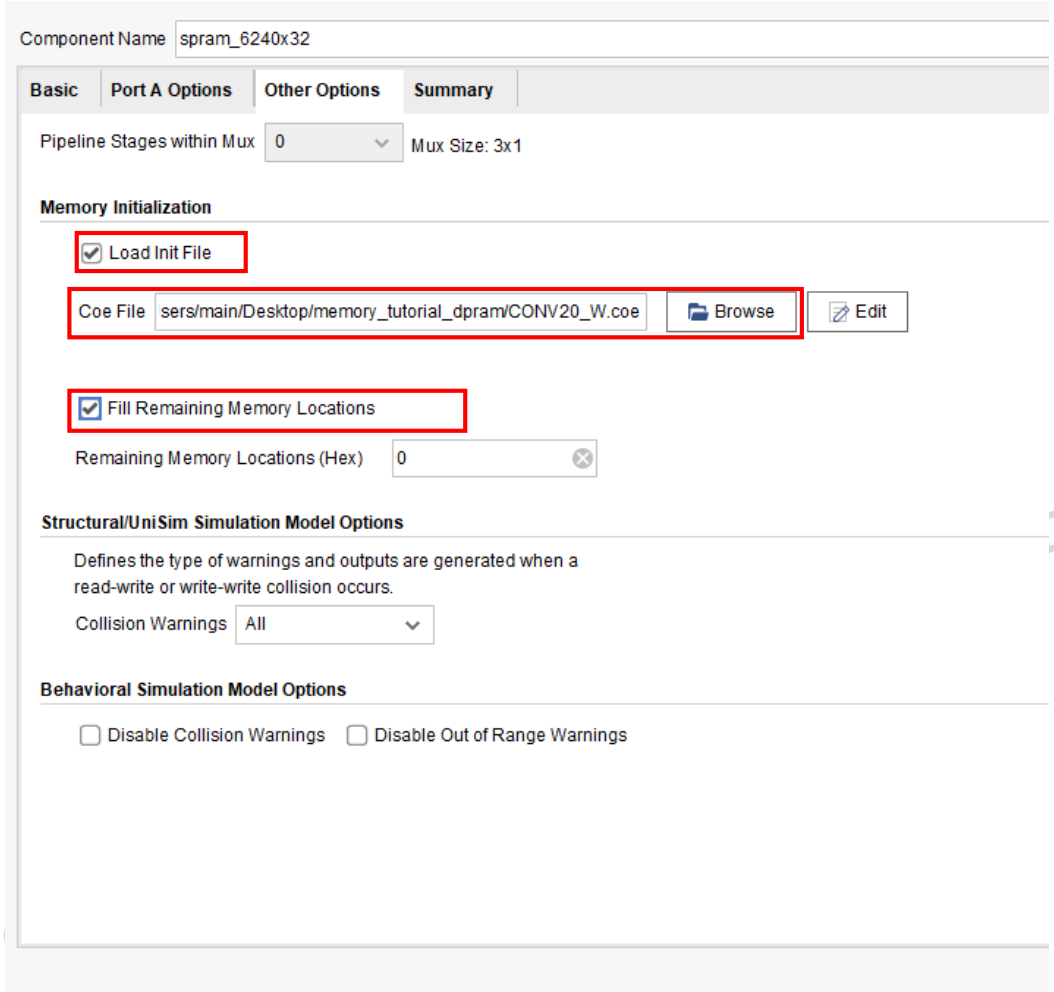

- 4. Enter the same name that you declare on your code
- 5. Select the memory type
  For reference, in given file, you will use Single Port Ram (spram) and Dual Port Ram(dpram)

# How to make BRAM IP?




- 6. Enter the proper width and depth of bram
- 7. Operating mode : No change
- 8. Primitives output register : deselect

# How to make BRAM IP?




- 9. click load init file
- 10. select the coe file
- 11. check the fill remaining memory locations
- 12. click ok button

* Memory initialization : initialize the bram data to selected "~.COE" file

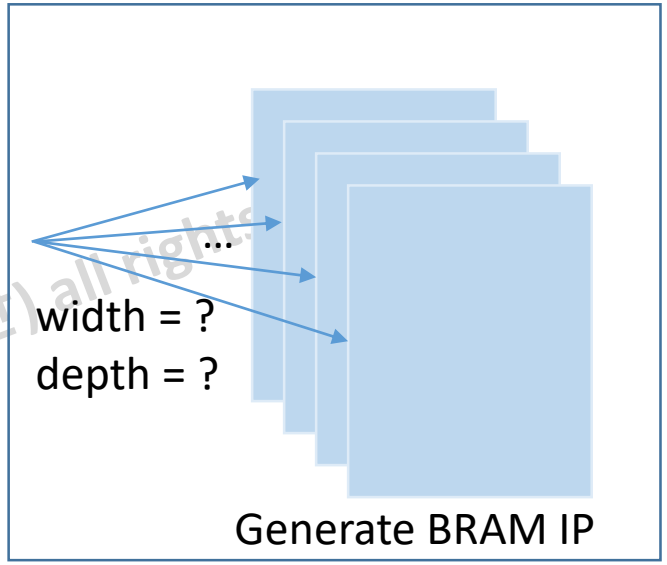
# Wrapper (spram_wrapper.v)

- Wrapper : wrapper chooses whether to make the BRAM IP or use the memory modeling (register)

```
12
13  `define FPGA 1
14  module spram_wrapper (
15      clk,                    // clock
16      addr,                   // input address
17      we,                     // input write enable
18      cs,                     // input chip-select
19      wdata,                  // input write data
20      rdata                   // output read-out data
21  );
```

wrapper

‘define FPGA = 1




// ‘define FPGA = 1

```
13  //`define FPGA 1
14  module spram_wrapper (
15      clk,                    // clock
16      addr,                   // input address
17      we,                     // input write enable
18      cs,                     // input chip-select
19      wdata,                  // input write data
20      rdata                   // output read-out data
21  );
```



Memory modeling

* memory modeling use “register”

# Wrapper (spram_wrapper.v) - parameter

- Parameter
  - DW : data bit width per word
  - AW : address bit width
  - DEPTH : word length

```
//------------------------------------------------------------------------+
// Declare parameters
//------------------------------------------------------------------------+
// output parameters
parameter DW = 64;          // data bit-width per word
parameter AW = 8;           // address bit-width
parameter DEPTH = 256;      // depth, word length
parameter N_DELAY = 1;
```

Ex)

DW = 4

DEPTH = 8

AW = $\log_2$DEPTH
= 3

| |
|---|
| 0 1 0 1 |
| 0 0 0 1 |
| 0 1 0 1 |
| 1 1 0 1 |
| 0 0 0 1 |
| 0 0 1 1 |
| 0 1 0 1 |
| 0 0 0 1 |

# Wrapper (spram_wrapper.v) - parameter

- Modify Parameter : you can customize the BRAM by modifying parameters

- 1. using BRAM IP (define FPGA = 1)
  - Set parameter
  - Add the proper "else if ~" code
  - Make BRAM IP

- 2. using memory modeling
  - Just set parameter

```
// output parameters
parameter DW = 32;
parameter AW = 13;
parameter DEPTH = 6240;
parameter N_DELAY = 1;
```

# Wrapper (spram_wrapper.v) - parameter

- Modify Parameter : you can customize the BRAM by modifying parameters

- 1. using BRAM IP (define FPGA = 1)
  - Set parameter
  - Add the proper "else if ~" code
  - Make BRAM IP

- 2. using memory modeling
  - Just set parameter

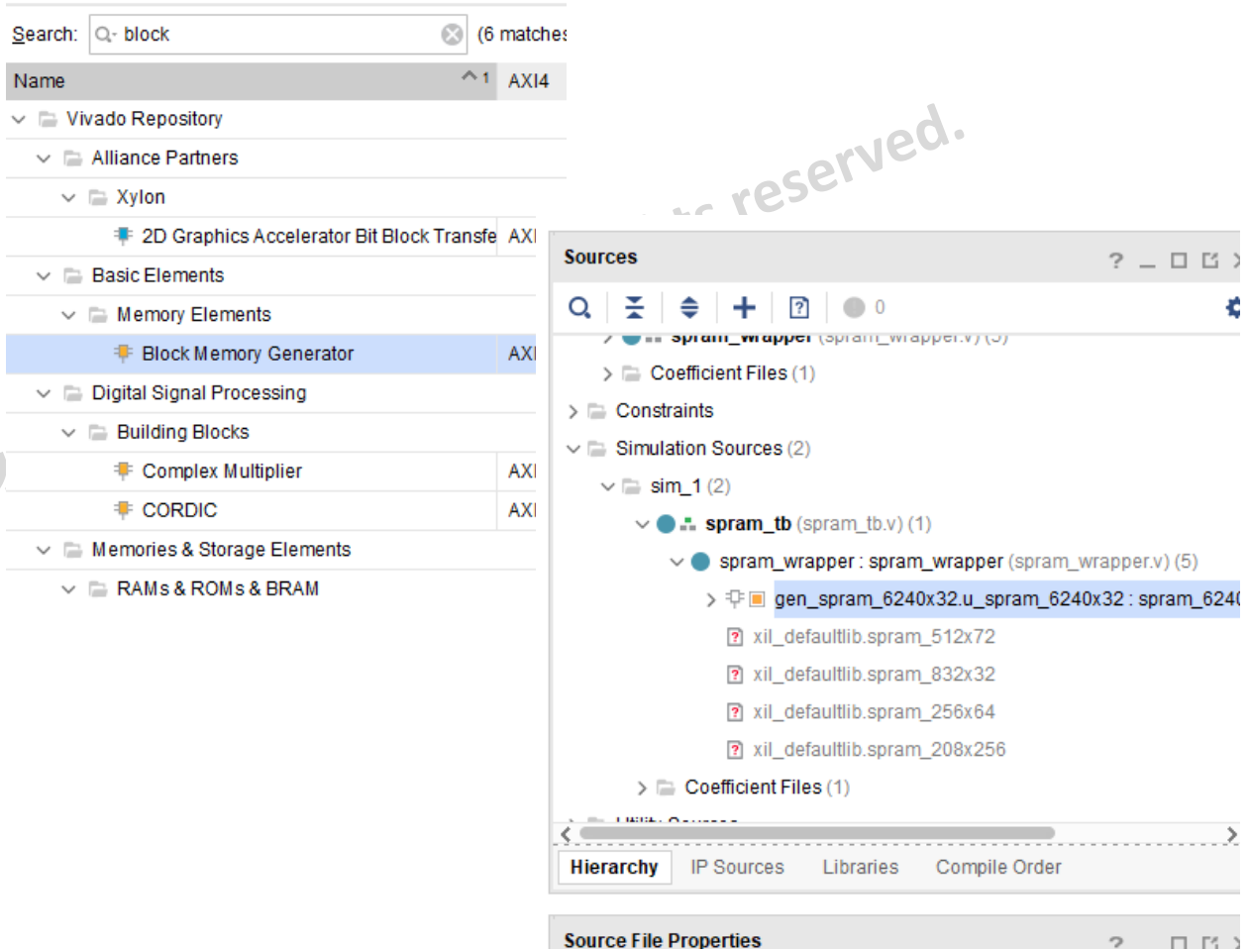
```
else if((DEPTH == 208) && (DW == 256)) begin: gen_spram_208x256
    spram_208x256 u_spram_208x256(
        // write
        .clka(clk),
        .ena(cs),
        .wea(we),
        .addra(addr),
        .dina(wdata),
        // read-out
        .douta(rdata)
    );
end
else if((DEPTH == 6240) && (DW == 32)) begin: gen_spram_6240x32
    spram_6240x32 u_spram_6240x32(
        // write
        .clka(clk),
        .ena(cs),
        .wea(we),
        .addra(addr),
        .dina(wdata),
        // read-out
        .douta(rdata)
    );
end
```

# Wrapper (spram_wrapper.v) - parameter

- Modify Parameter : you can customize the BRAM by modifying parameters

- 1. using BRAM IP (define FPGA = 1)
  - Set parameter
  - Add the proper "else if ~" code
  - Make BRAM IP

- 2. using memory modeling
  - Just set parameter



# Wrapper (spram_wrapper.v) - parameter

- Modify Parameter : you can customize the BRAM by modifying parameters

- 1. using BRAM IP (define FPGA = 1)
  - Set parameter
  - Add the proper "else if ~" code
  - Make BRAM IP

- 2. using memory modeling
  - Just set parameter

```
// output parameters
parameter DW = 32;
parameter AW = 13;
parameter DEPTH = 6240;
parameter N_DELAY = 1;
```

# Explanation - spram_wrapper.v

- Memory modeling : just operate like BRAM IP but it's actually register

```
//------------------------------------------------------------
// Memory modeling
//------------------------------------------------------------
reg [DW-1          :   0]      mem[0:DEPTH-1]; // Memory cell
```

Make "mem" register (data storage)

```
// Write
always @(posedge clk) begin
    if(cs && we)            mem[addr] <= wdata;
end
```

Data write to "mem"

```
// Read
generate
   if(N_DELAY == 1) begin: gen_delay_1
      always @(posedge clk)
         if (cs && !(|we)) rdata_o <= mem[addr];

      assign rdata = rdata_o;
   end
```

Data read from "mem"
: 1 cycle delay

```
   else begin: gen_delay_n
      reg [N_DELAY*DW-1:0] rdata_r;

      always @(posedge clk)
         if (cs && !(|we)) rdata_r[0*DW+:DW] <= mem[addr];
```

Data read from "mem"
: more than 1 cycle delay
(not used for this example! Just ignore)

```
      always @(posedge clk) begin: delay
         integer i;
         for(i = 0; i < N_DELAY-1; i = i+1)
            if(cs && !(|we))
               rdata_r[(i+1)*DW+:DW] <= rdata_r[i*DW+:DW];
      end
```

Support N cycle data delay
: 1 cycle delay for this example

```
      assign rdata = rdata_r[(N_DELAY-1)*DW+:DW];
   end
endgenerate
```

Assign "mem" output port

# Test bench (spram_tb.v)

- Test bench for bram operation
  - Operation
    - 1. Read the initialized value (initialized by COE file)
    - 2. Write the test data set (width = 32, depth = 16)
    - 3. Read the changed value

```
// test data set: width =  DW , depth = 16
test_data[0] = 32'h00000000;
test_data[1] = 32'h11111111;
test_data[2] = 32'h22222222;
test_data[3] = 32'h33333333;
test_data[4] = 32'h44444444;
test_data[5] = 32'h55555555;
test_data[6] = 32'h66666666;
test_data[7] = 32'h77777777;
test_data[8] = 32'h88888888;
test_data[9] = 32'h99999999;
test_data[10] = 32'haaaaaaaa;
test_data[11] = 32'hbbbbbbbb;
test_data[12] = 32'hcccccccc;
test_data[13] = 32'hdddddddd;
test_data[14] = 32'heeeeeeee;
test_data[15] = 32'hffffffff;
```

write

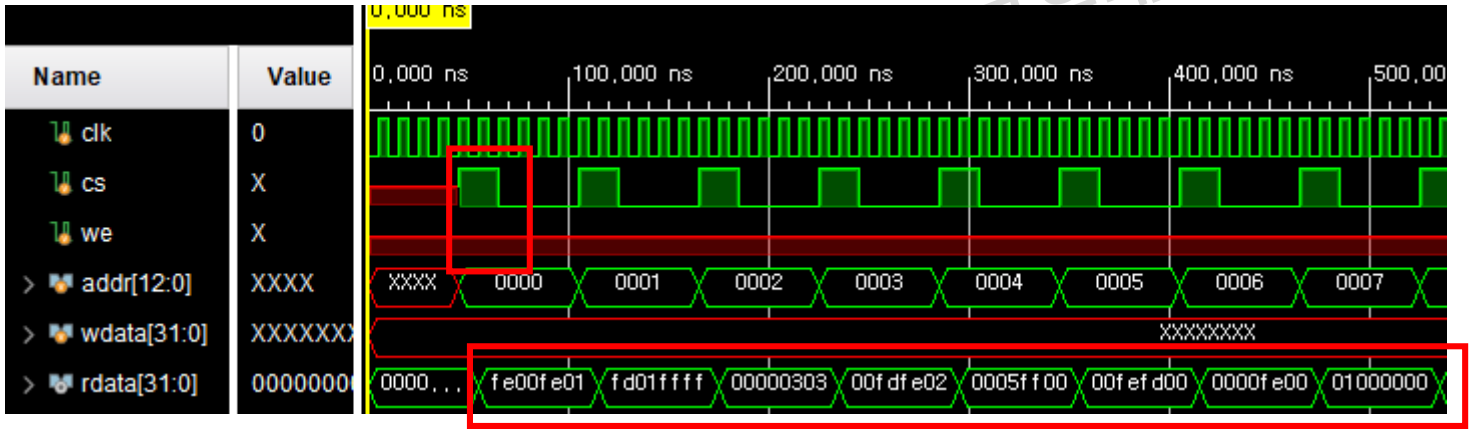
read

```
// ---------- read operation ---------- //
for(i=0;i<DEPTH;i=i+1) begin
    #(8*CLK_HALF_CYCLE)
    cs = 1'b1;
    addr = i;
    #(4*CLK_HALF_CYCLE)
    cs = 1'b0;
end

// ---------- write operation ---------- //
for(i=0;i<16;i=i+1) begin
    #(8*CLK_HALF_CYCLE)
    cs = 1'b1;
    we = 1'b1;
    addr = i;
    wdata = test_data[i];
    #(2*CLK_HALF_CYCLE)
    cs = 1'b0;
    we = 1'b0;
end

// ---------- read operation ---------- //
for(i=0;i<DEPTH;i=i+1) begin
    #(8*CLK_HALF_CYCLE)
    cs = 1'b1;
    addr = i;
    #(4*CLK_HALF_CYCLE)
    cs = 1'b0;
end
```

# Test bench (spram_tb.v) - waveform

- 1. Read the initialized value (initialized by COE file)

```
// ---------- read operation ---------- //
for(i=0;i<DEPTH;i=i+1) begin
    #(8*CLK_HALF_CYCLE)
    cs = 1'b1;
    addr = i;
    #(4*CLK_HALF_CYCLE)
    cs = 1'b0;
end
```




CONV20_W.coe

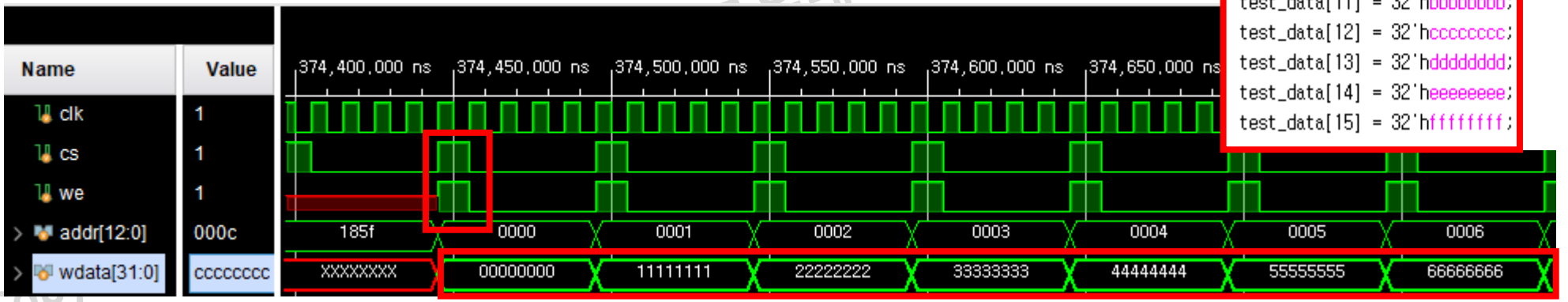
```
1  memory_initialization_radix=16;
2  memory_initialization_vector=
3  fe00fe01    ,
4  fd01ffff    ,
5  00000303    ,
6  00fdfe02    ,
7  0005ff00    ,
8  00fefd00    ,
9  0000fe00    ,
10 01000000    ,
11 fdff00fb    ,
12 01fefffb    ,
13 ffff0201    ,
14 00fa0001    ,
15 010400fd    ,
16 0500ff00    ,
```

# Test bench (spram_tb.v) - waveform

- 2. Write the test data set (width = 32, depth = 16) ;

```
// ---------- write operation ---------- //
for(i=0;i<16;i=i+1) begin
    #(8*CLK_HALF_CYCLE)
    cs = 1'b1;
    we = 1'b1;
    addr = i;
    wdata = test_data[i];
    #(2*CLK_HALF_CYCLE)
    cs = 1'b0;
    we = 1'b0;
end
```

```
// test data set: width = DW, depth = 16
test_data[0] = 32'h00000000;
test_data[1] = 32'h11111111;
test_data[2] = 32'h22222222;
test_data[3] = 32'h33333333;
test_data[4] = 32'h44444444;
test_data[5] = 32'h55555555;
test_data[6] = 32'h66666666;
test_data[7] = 32'h77777777;
test_data[8] = 32'h88888888;
test_data[9] = 32'h99999999;
test_data[10] = 32'haaaaaaaa;
test_data[11] = 32'hbbbbbbbb;
test_data[12] = 32'hcccccccc;
test_data[13] = 32'hdddddddd;
test_data[14] = 32'heeeeeeee;
test_data[15] = 32'hffffffff;
```

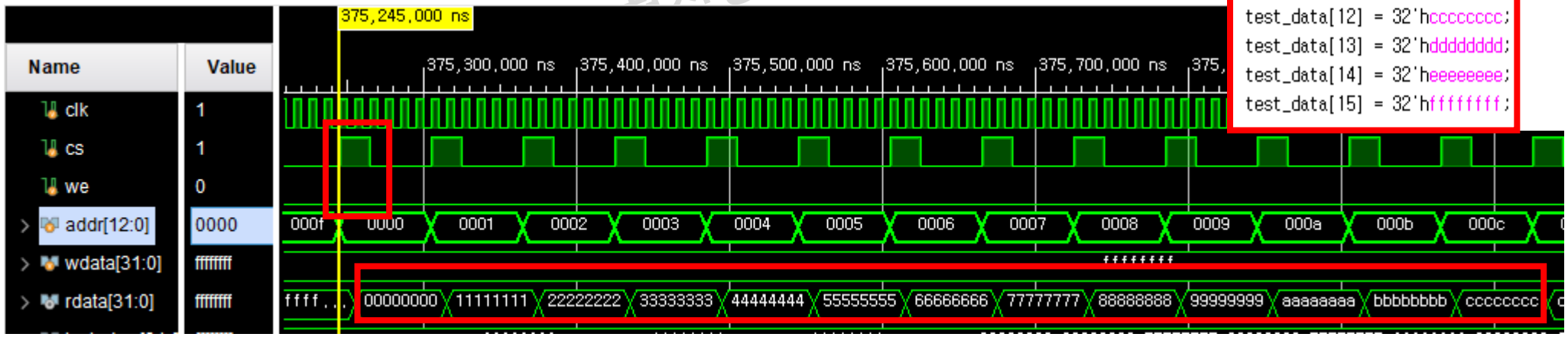


# Test bench (spram_tb.v) - waveform

- 3. Read the changed value

```
// ---------- read operation ---------- //
for(i=0;i<DEPTH;i=i+1) begin
    #(8*CLK_HALF_CYCLE)
    cs = 1'b1;
    addr = i;
    #(4*CLK_HALF_CYCLE)
    cs = 1'b0;
end
```

```
// test data set: width = DW, depth = 16
test_data[0] = 32'h00000000;
test_data[1] = 32'h11111111;
test_data[2] = 32'h22222222;
test_data[3] = 32'h33333333;
test_data[4] = 32'h44444444;
test_data[5] = 32'h55555555;
test_data[6] = 32'h66666666;
test_data[7] = 32'h77777777;
test_data[8] = 32'h88888888;
test_data[9] = 32'h99999999;
test_data[10] = 32'haaaaaaaa;
test_data[11] = 32'hbbbbbbbb;
test_data[12] = 32'hcccccccc;
test_data[13] = 32'hdddddddd;
test_data[14] = 32'heeeeeeee;
test_data[15] = 32'hffffffff;
```



# Lab 2: dual-port RAM

- Lab 2: Dual-port RAM (spram): 16 bits per word, 6240 words
  - Use a dual-port RAM
  - Generate a dedicated RAM using Vivado IP generator
  - Initialize a RAM using a coe file
  - Test bench

CONV20_W.coe

```
memory_initialization_radix=16;
memory_initialization_vector=
fe00fe01    ,
fd01ffff    ,
00000303    ,
00fdfe02    ,
0005ff00    ,
00fefd00    ,
0000fe00    ,
01000000    ,
fdff00fb    ,
01fefffb    ,
ffff0201    ,
00fa0001    ,
010400fd    ,
0500ff00    ,
```

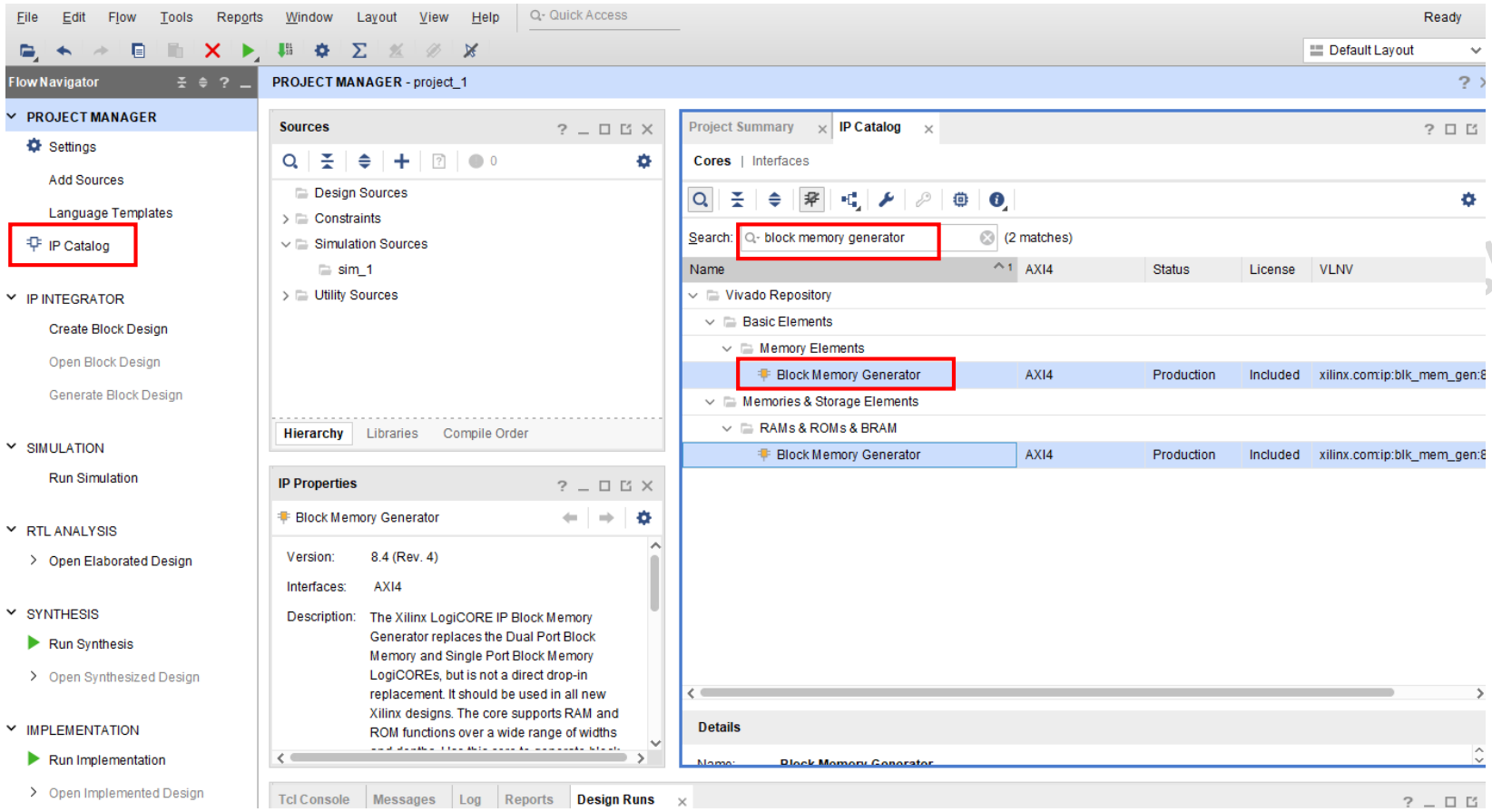
# Double-port block RAM (dpram)

- Ports
  - Port A : Write only
    - Clock (clka)
    - Read/Write enable (ena)
    - Write enable (wea)
    - Address (addra)
    - Write data (dina)
  - Port B : Read only
    - Clock (clkb)
    - Read/write enable (enb)
    - Address (addrb)
    - Read data (doutb)

- Read operation
  - If(enb)
    - doutb <= mem[addrb]
- Write operation
  - If(ena & wea)
    - mem[addra] <= dina

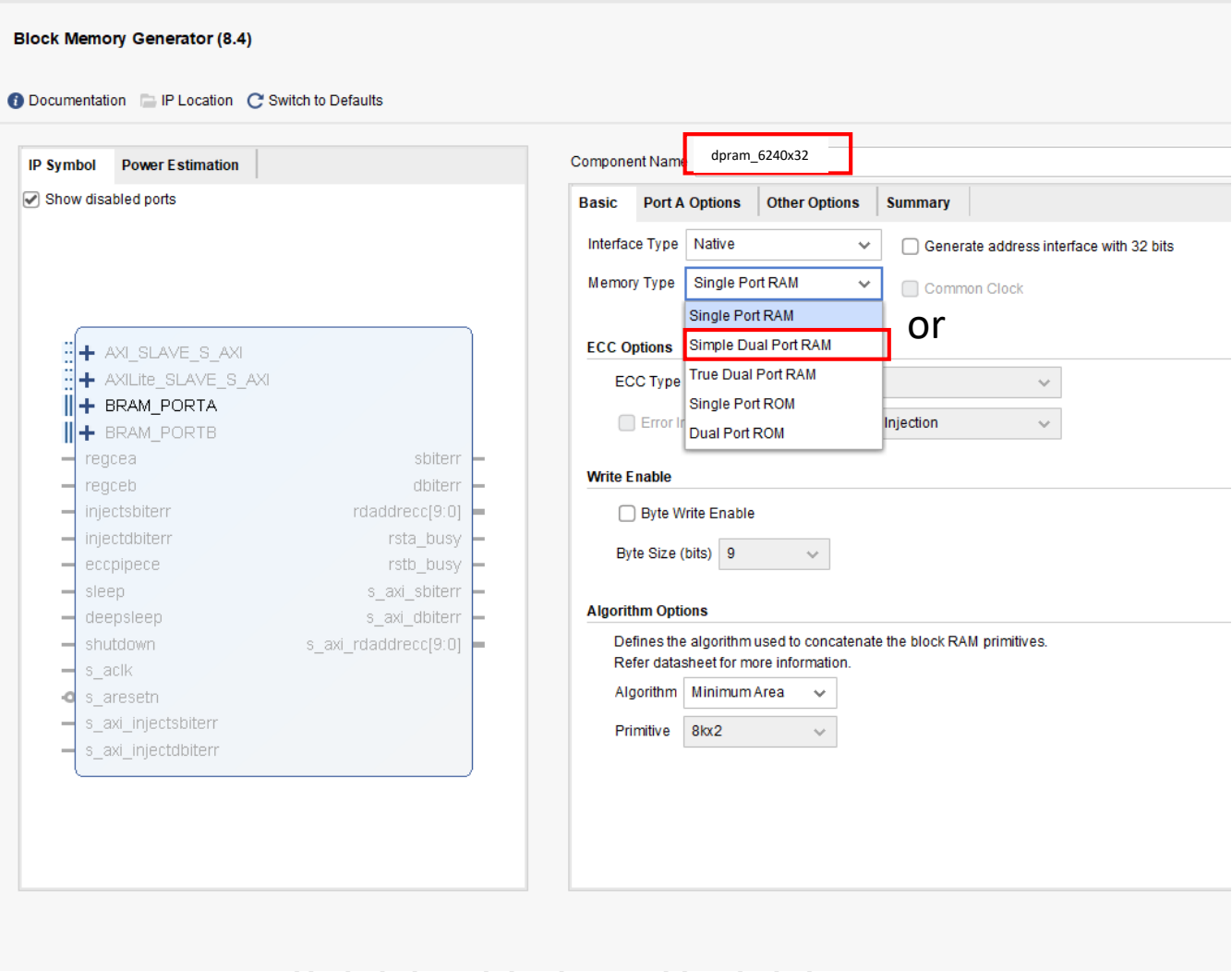
* Double-port
  - Allow read/write operations at the same time

# FPGA: How to make BRAM IP?




- 1. click IP catalog
- 2. search the "block memory generator"
- 3. double click the "block memory generator"

# How to make BRAM IP?




- 4. Enter the same name that you declare on your code

- 5. Select the memory type

  For reference, in given file, you will use Single Port Ram (spram) and Dual Port Ram(dpram)

# Double-port block RAM (dpram_tb.v)

- About wrapper, test bench… : It's almost same with spram's

- Point
  - dpram allow read/write operations at the same time!
  - dpram write the data through port A!
  - dpram read the data through port B!

These read & write can operate
at the same time, but what happen?

```
// ---------- read operation ---------- //
for(i=0;i<DEPTH;i=i+1) begin
    #(8*CLK_HALF_CYCLE)
    enb = 1'b1;
    addrb = i;
    #(4*CLK_HALF_CYCLE)
    enb = 1'b0;
end

// ---------- write operation ---------- //
for(i=0;i<16;i=i+1) begin
    #(8*CLK_HALF_CYCLE)
    ena = 1'b1;
    wea = 1'b1;
    addra = i;
    dia = test_data[i];
    #(2*CLK_HALF_CYCLE)
    ena =1'b0;
    wea = 1'b0;
end

// ---------- read operation ---------- //
for(i=0;i<DEPTH;i=i+1) begin
    #(8*CLK_HALF_CYCLE)
    enb = 1'b1;
    addrb = i;
    #(4*CLK_HALF_CYCLE)
    enb = 1'b0;
end
```

# Incoming lectures

- Sliding windows and controller
- DMA