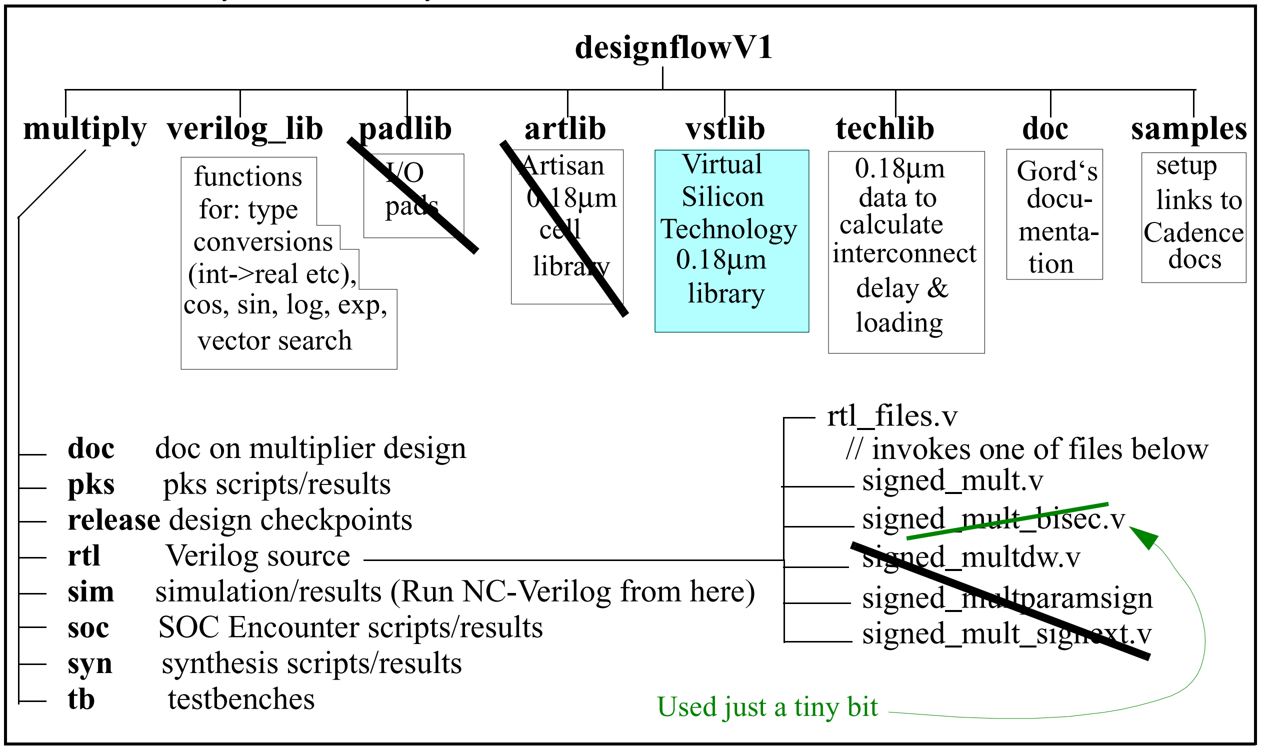
1.0 Design of an Integrated Multiplier

This is a sample design done by Gord Allan.
The design is a two's complement multiplier.
Gord's files allow a choice of two cell libraries and several types of multiply.
We will choose the VST cells and a "sign-extension" multiply for this lab
The design directories with sample files are built using a shell script (comand file, batch file) **instal**.

1.1 The Design Directories and Files

FIGURE 1 Directory structure built by **instal**. Crossed out files are not used in 4708.


designflowV1
multiply
verilog_lib
padlib
artlib
vstlib
techlib
doc
samples
functions for: type conversions (int->real etc), cos, sin, log, exp, vector search
~~I/O pads~~
~~Artisan 0.18µm cell library~~
Virtual Silicon Technology 0.18µm library
0.18µm data to calculate interconnect delay & loading
Gord's docu-menta-tion
setup links to Cadence docs
doc doc on multiplier design
pks pks scripts/results
release design checkpoints
rtl Verilog source
sim simulation/results (Run NC-Verilog from here)
soc SOC Encounter scripts/results
syn synthesis scripts/results
tb testbenches
rtl_files.v
// invokes one of files below
signed_mult.v
~~signed_mult_bisec.v~~
~~signed_multdw.v~~
~~signed_multparamsign~~
~~signed_mult_signext.v~~
Used just a tiny bit

1.2 Verilog Source

The code is modular and hierarctical.
Sections of code which are likely to be independently reused, are put in seperate files.
These files are coupled together by **`include** comands
These copy the text from the other file into the file with the include statement.
Thus for the compilier, it looks like all the text is in one file.
For the design administrator, you, it looks like each concept has its own file.
Example:

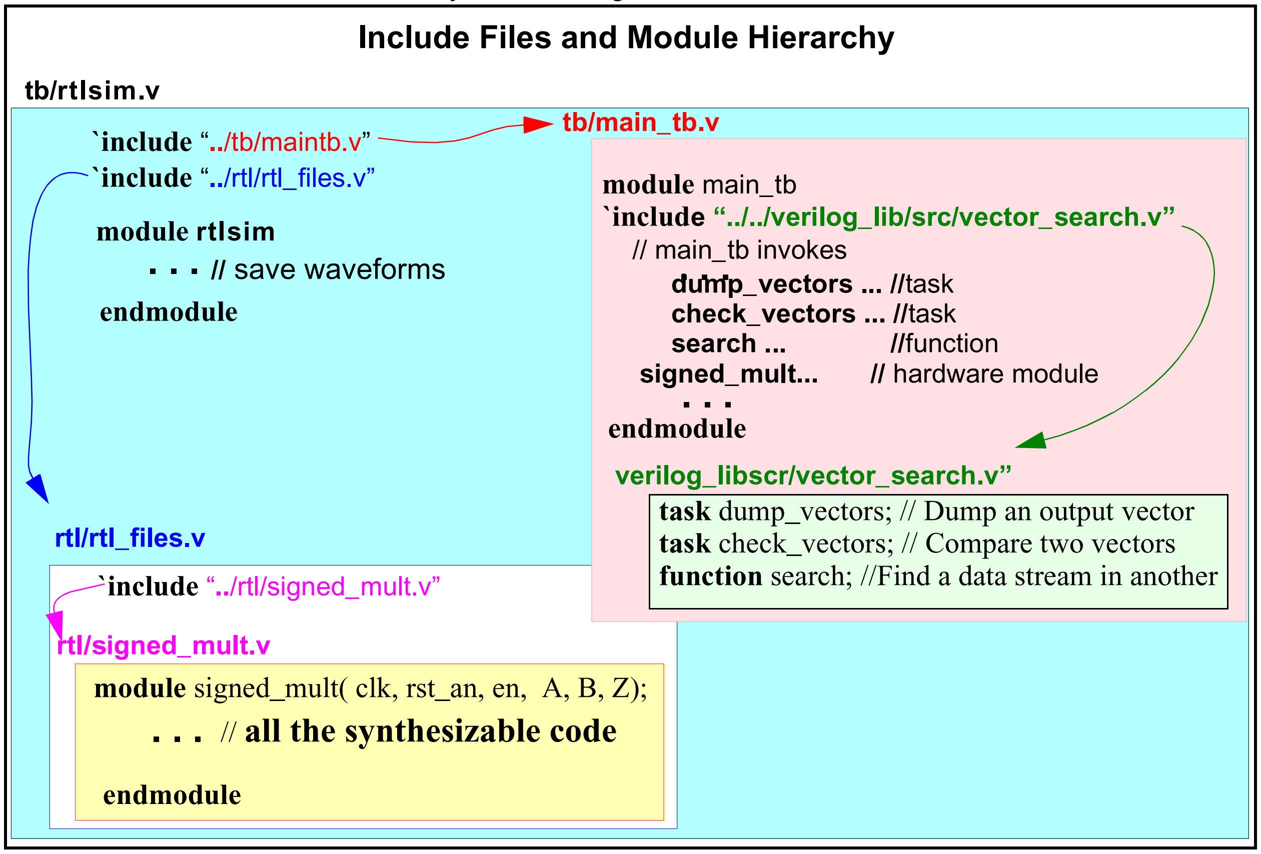
rtl_file.v

```
    `include "../rtl/signed_mult.v"
  // `include "..rtl/signed_bisect.v"
```

Only one active line

The user includes which multiplier implementation he/she wants to try.

FIGURE 2 File structure and hierarchy of the Verilog code.



```
Include Files and Module Hierarchy

tb/rtlsim.v
    `include "../tb/maintb.v"
    `include "../rtl/rtl_files.v"

    module rtlsim
        . . . // save waveforms
    endmodule

tb/main_tb.v
module main_tb
`include "../../verilog_lib/src/vector_search.v"
  // main_tb invokes
     dump_vectors ... //task
     check_vectors ... //task
     search ...        //function
   signed_mult...      // hardware module
       . . .
endmodule

verilog_libscr/vector_search.v"
  task dump_vectors; // Dump an output vector
  task check_vectors; // Compare two vectors
  function search; //Find a data stream in another

rtl/rtl_files.v
   `include "../rtl/signed_mult.v"

rtl/signed_mult.v
  module signed_mult( clk, rst_an, en,  A, B, Z);
      . . .  // all the synthesizable code

   endmodule
```

1.2.1 Verilog When Expanded

After the includes are expanded

There are two modules.

1) The synthesizable multiplier.

2) The test bench **main_tb.**
This contains the code for two tasks and a function.

Functions allow coding repetative combinational logic.

Tasks are like functions but may contain timing statements like **@, #, wait**.

Gord uses tasks to compare the simulated and theoretical outputs.

He could have made them all in one big module.

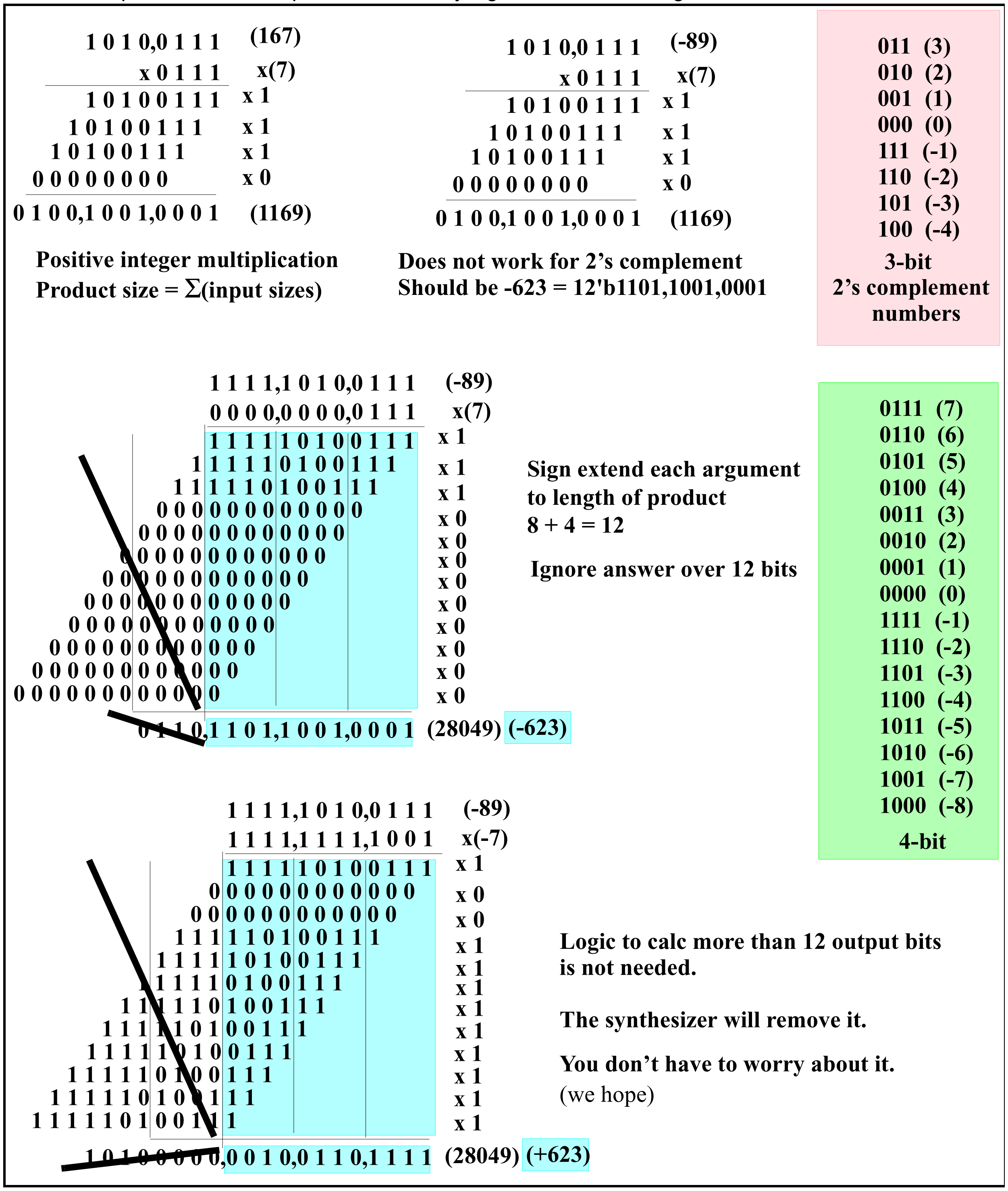
But the tasks are easier to remove and use on some other design (Viterbi?)

FIGURE 3 Code after expanding the `**includes**

```
module main_tb

   task dump_vectors; // Dump an output vector
       . . .code for task
   task check_vectors; // Compare two vectors
       . . .code for task
   function search; //Find a data stream in another
       . . .code for function

  // main_tb invokes
     dump_vectors ... //task
     check_vectors ... //task
     search ...        //function
   signed_mult...      // hardware module
       . . .
endmodule

  module signed_mult( clk, rst_an, en,  A, B, Z);
      . . .  // all the synthesizable code

   endmodule
```

1.3 The Signed Multiplier

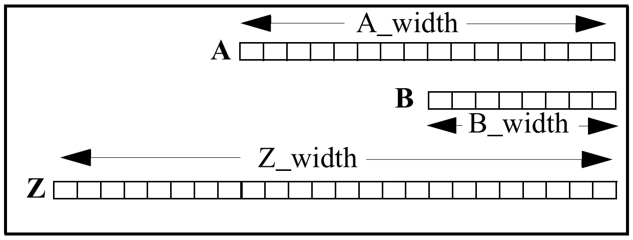
FIGURE 4 Operation of a multiplier that works by sign extension and logic minimization.



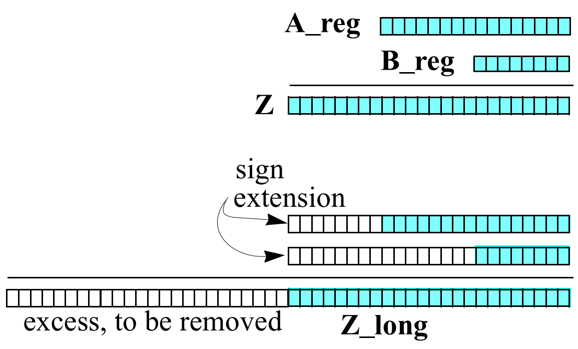
1 0 1 0,0 1 1 1 (167)
x 0 1 1 1 x(7)
1 0 1 0 0 1 1 1 x 1
1 0 1 0 0 1 1 1 x 1
1 0 1 0 0 1 1 1 x 1
0 0 0 0 0 0 0 0 x 0
0 1 0 0,1 0 0 1,0 0 0 1 (1169)
Positive integer multiplication
Product size = Σ(input sizes)
1 0 1 0,0 1 1 1 (-89)
x 0 1 1 1 x(7)
1 0 1 0 0 1 1 1 x 1
1 0 1 0 0 1 1 1 x 1
1 0 1 0 0 1 1 1 x 1
0 0 0 0 0 0 0 0 x 0
0 1 0 0,1 0 0 1,0 0 0 1 (1169)
Does not work for 2’s complement
Should be -623 = 12'b1101,1001,0001
011 (3)
010 (2)
001 (1)
000 (0)
111 (-1)
110 (-2)
101 (-3)
100 (-4)
3-bit 2’s complement numbers
1 1 1 1,1 0 1 0,0 1 1 1 (-89)
0 0 0 0,0 0 0 0,0 1 1 1 x(7)
1 1 1 1 1 0 1 0 0 1 1 1 x 1
1 1 1 1 1 0 1 0 0 1 1 1 x 1
1 1 1 1 1 0 1 0 0 1 1 1 x 1
0 0 0 0 0 0 0 0 0 0 0 0 x 0
0 0 0 0 0 0 0 0 0 0 0 0 x 0
0 0 0 0 0 0 0 0 0 0 0 0 x 0
0 0 0 0 0 0 0 0 0 0 0 0 x 0
0 0 0 0 0 0 0 0 0 0 0 0 x 0
0 0 0 0 0 0 0 0 0 0 0 0 x 0
0 0 0 0 0 0 0 0 0 0 0 0 x 0
0 0 0 0 0 0 0 0 0 0 0 0 x 0
0 0 0 0 0 0 0 0 0 0 0 0 x 0
0 1 1 0,1 1 0 1,1 0 0 1,0 0 0 1 (28049) (-623)
Sign extend each argument to length of product
8 + 4 = 12
Ignore answer over 12 bits
0111 (7)
0110 (6)
0101 (5)
0100 (4)
0011 (3)
0010 (2)
0001 (1)
0000 (0)
1111 (-1)
1110 (-2)
1101 (-3)
1100 (-4)
1011 (-5)
1010 (-6)
1001 (-7)
1000 (-8)
4-bit
1 1 1 1,1 0 1 0,0 1 1 1 (-89)
1 1 1 1,1 1 1 1,1 0 0 1 x(-7)
1 1 1 1 1 0 1 0 0 1 1 1 x 1
0 0 0 0 0 0 0 0 0 0 0 0 x 0
0 0 0 0 0 0 0 0 0 0 0 0 x 0
1 1 1 1 1 0 1 0 0 1 1 1 x 1
1 1 1 1 1 0 1 0 0 1 1 1 x 1
1 1 1 1 1 0 1 0 0 1 1 1 x 1
1 1 1 1 1 0 1 0 0 1 1 1 x 1
1 1 1 1 1 0 1 0 0 1 1 1 x 1
1 1 1 1 1 0 1 0 0 1 1 1 x 1
1 1 1 1 1 0 1 0 0 1 1 1 x 1
1 1 1 1 1 0 1 0 0 1 1 1 x 1
1 1 1 1 1 0 1 0 0 1 1 1 x 1
1 0 1 0 0 0 0 0,0 0 1 0,0 1 1 0,1 1 1 1 (28049) (+623)
Logic to calc more than 12 output bits is not needed.
The synthesizer will remove it.
You don’t have to worry about it.
(we hope)

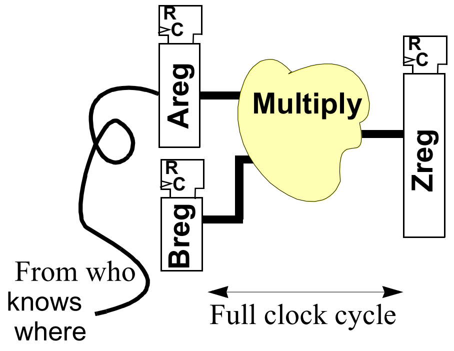
1.3.1 The Signed-Multiply Module

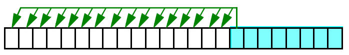
```
/* HEADER

 AUTHOR:           Gord Allan, Carleton University, Ottawa, Ont.

 MODULE NAME:       signed_mult
 SUBMODULES USED:    none

 DESCRIPTION:        A signed multiplier
                 - Paramaterized bit-widths
                 - Uses sign-extension approach and relies
                   on the synthesis tool to simplify.

 Parameters:        A_width
                   B_width

 Inputs:            A        16-bit signed input
                   B        8-bit signed input

                  rst_an  - asynchronous active low reset
                  en      - pauses the system
                  clk

 Outputs:           Z       24-bit signed multiply result

 Other Notes:      Inputs and outputs are directly registered, so bus delay is minimized.

 QUICK SYNTHESIS RESULTS (0.18u):    Gates:            Delay:

 */
 // The next line adds an RCS Header to each source file upon check-out (co)
 // $Log$
 // RCS is the Unix Revision Control System, which many designs use.


 module signed_mult( clk, rst_an, en,  A, B, Z);

   parameter A_width = 16;  // Width of one multiplier input
   parameter B_width = 8;   // Width of the multiplier output;
                            // Note it would be better if these were global with the test bench.

  // A multiplier output width is the sum of the input widths.
   parameter Z_width = A_width + B_width;

 input clk, rst_an, en;
 input [A_width-1:0] A;
 input [B_width-1:0] B;

 output [Z_width-1:0] Z;

```



```
// Register Declarations
// we should register our IO to simplfy timing constraints and analysis
reg [A_width-1:0] Areg;
reg [B_width-1:0] Breg;

 reg [Z_width-1:0] Z;

 wire [Z_width-1:0]      Aext, Bext;
 wire [(2*Z_width)-1:0]  Z_long;
   // Z_width bits * Z_width bits is right-
  //  fully a 2*Z_width bit result.
   //  but we throw out the top Z_width bits
  //  and the tool throws out its logic.

 // register the inputs to ease timing
 always @(posedge clk or negedge rst_an)
     if(~rst_an) begin   // asynchronous, active-low reset
         Areg <=0;
         Breg <=0;
     end
     else if(en) begin
         Areg <=A;
         Breg <=B;
     end

 // ==================================================================
 // Perform the input sign extension (to 24 bits if inputs are 16 and 8 bits)
 assign Aext[Z_width-1:0] = { {(Z_width - A_width){Areg[A_width-1]}}, Areg };
 assign Bext[Z_width-1:0] = { {(Z_width - B_width){Breg[B_width-1]}}, Breg };
                // assign Bext[23:0] = { {(16){Breg[7]}}, Breg };

 // This is where the synthesis tool will do all its work
 // It starts by making  a   Z_width*Z_width     bit multiplier (48 bit output 4 16x8 bit inputs)

 assign  Z_long = Aext * Bext;   // Synthesizer will generate multiply logic

 // Only keep the lower Z_width bits - the synthesis tool will remove all of the unused logic
 always @(posedge clk or negedge rst_an)
     if(~rst_an)
         Z <= 0;
     else if(en)
         Z <= Z_long[Z_width-1:0];   // Truncation done here.(48 bits to 24 bits for 16 x 8 bit inputs)

endmodule
```

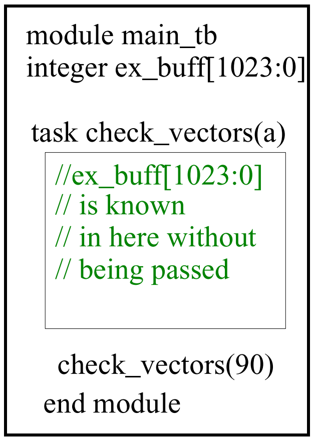






1.4 Main Test Bench

```
`timescale 1ns/10ps
            //The `timescale directive should only be included once at the beginning of a simulation.

// The Main line driver testbench
module main_tb;

   // used in if(`VERBOSE) $display(...) statements to control how much status is displayed
   `define VERBOSE 1
       /* Determine the extent of debugging information displayed.
          0 for none; higher values to dump more information. */

   // BUFFER_SIZE defines the maximum size of the expected and output buffers
   // ERRORFATAL is used by check_vectors to end the simulation if a match is not found
   `define BUFFER_SIZE 1023
   `define ERRORFATAL 0

   // Declarations ================================================
   reg clk, rst_an;

   wire [15:0] A;          // the least-sig bits of A_int and B_int
   wire [7:0]  B;
   wire [23:0] result;

   integer A_int, B_int;   // the inputs are declared as integers
   integer result_int;     // the converted version of the result
   integer i;              // temporary loop variable

   integer clk_counter;    // records the number of clock cycles since reset
   /* Verilog, as yet,  does not have global variables. One does the test bench
comparisons in integers because it is much easier than with reg or wir.
One cannot create a vector of integers  ie. integer  [N:0]  xyz;
One must create an array ie.                integer  xyz  [N:0];
Unfortunately arrays cannot be passed in Verilog.
Also there are no global variables between modules.
The work around is to do the comparison in a task. Variables known in the mod-
ule containing the task are known in the task (sort of global to the task). */
```



```
module main_tb
integer ex_buff[1023:0]

task check_vectors(a)
  //ex_buff[1023:0]
  // is known
  // in here without
  // being passed

  check_vectors(90)
end module
```

```
   integer expected_buffer [`BUFFER_SIZE:0];     // holds the expected output vectors
   integer output_buffer   [`BUFFER_SIZE:0];     // holds the recorded output vectors

        // A  simple clock
   initial clk = 0;
   initial forever #100 clk <= ~clk;

   // Instantiate the hardware (DUT stands for Device Under Test) =====================
   signed_mult dut ( .clk(clk), .rst_an(rst_an), .en(1'b1),  .A(A), .B(B), .Z(result));

```

```
//==============================================================
//     Include functions for vector searching and data-type conversions

//--------------------------------------------------------------------------------------------
// vector_search.v  - Includes routines to search for a partial vector within another. Thus they
//  can find the correct output without us bothering to calculate the exact time it appears.
//          - very useful for checking expected results that may experience unknown latency
//           - ASSUMES THE FOLLOWING ARE DECLARED AND USED:
//            integer expected_buffer[], output_buffer[], clk_counter;
//
//   check_vectors(number_of_words);   --> searches for
//                 expected_buffer[1:number_of_words] in output_buffer[0:`BUFFER_SIZE]
//     dump_vectors(max);                  --> Displays the recorded vectors from 0 to max
/* Include will insert the search function code here during compile. */
`include "../../verilog_lib/src/vector_search.v"

// ==================================================
//  Vector IO

// Convert the DUT's 24-bit result to a 32-bit integer for logging comparisons
always @(result) result_int <= {{8{result[23]}},result[23:0]}; // Sign extend result to 32 bits
// recall:    {8{result[23]}}, ...  concatenates 8 copies of the leading bit.

/* Sample and Record the output vectors
The interface to the DUT should behave like hardware, capturing the result on the positive edge of the clock
like a register. The integer results are stored sequentially in output buffer for later comparison. */
always @(posedge clk or negedge rst_an)
   if(~rst_an) begin
      clk_counter <= 0;
      for (i=0;i<BUFFER_SIZE;i=i+1) output_buffer[i]<=0;
      end
   else    begin
      output_buffer[clk_counter] <= result_int;
      clk_counter <= clk_counter + 1;
   end

// convert the stimulus from integers to 16/8-bit signed vectors for the DUT.
assign A = A_int;    // automatically truncate integers.
assign B = B_int;


// Status monitoring ================================================
always @(posedge clk)
   if(rst_an & `VERBOSE)   $display("%t - A_int = %d,  B_int = %d,    Current Result = %d",
                                               $time, A_int, B_int, result_int);
```

```

//
//
//
//==========================================================================
//    Test Sequencing
//
//     if there are multiple tests, they should be broken apart into seperate files
//      eg.  `include "../tb/test1_random.v"
//           `include "../tb/test2_overflow.v"
//       ...

  initial
    begin:  Test_sequence

  // Each test should:
  //    reset the system
  //    set A_int and B_int on each clock cycle (eg.     always @(posedge clk) A_int <= $random % 128;  )
  //    record a set of expected vectors in    expected_buffer[0...BUFFER_SIZE]
  //    call the check_vectors function to verify results


        $display("%t ==========================================", $time);
        $display("%t - Starting New Test -", $time);
        $display("%t -     Testname:    test1_random", $time);
        $display("%t -     Description: Sets random values for A_int and B_int. ", $time);
        $display("%t -                  Records the expected results and compares.", $time);
        $display("%t - ------------------------------------------------------------", $time);

        rst_an <= 1;     #1;                          // raise reset so it can fall and give a negedge.
        @(posedge clk)   #1  rst_an <= 0;       // assert reset after next clock edge.
        $display("%t - Resetting System", $time);
        A_int = 0;       B_int = 0;

        @(posedge clk)    rst_an <= 1;          // Remove reset synchronously.
        $display("%t - Removing Reset", $time);

        $display("%t - Applying 100 random inputs for A and B and recording vectors.", $time);
        repeat(100) begin
            @(posedge clk);                             // change all input just after clock edge
          // The circuit uses posedge clock, but delay signal changes by a minimum amount.
            #1
            A_int =  ($random) %65536;                 // get a value from 0 to 65535
         //  The  mod % operator reduces random value the range 0 to 65535.
            A_int = (Aint <0) ? -A_int: A_int;
            /* This is probably redundant, but some Verilogs have given neg numbers  from  %.
               Numbers over 32767 are now converted to negative integers. */
             A_int = (A_int >32767) ? (65536 - A_int)*( -1) : A_int;
          //  Give range -32768 to 32767, irrespective of how many bits are in a machine integer.

        B_int = ($random) %255;                    // get a value from 0 to 255
        B_int = (Bint <0) ? -B_int: Bint;
        B_int =  (B_int >127) ? (256 - B_int)*( -1) : B_int;    //ensure -128 =< B_int =<+127


      expected_buffer[clk_counter] = A_int * B_int;   // Calc the ideal result from integers.
      // It was very important here that  A_int  and  B_int  were sign extended when negative.
    end  // repeat(100)

      // verify that the expected vectors were found somewhere in the output stream
      check_vectors(90);   // Task that compares  expected_buffer with output_buffer.
                           // for 90 sets of test vectors.   Why not 100?
      $display("%t - -------------Random Inputs Test Complete -----------------", $time);

    //=================================================================

      $display("%t -===================================================", $time);
      $display("%t - Starting New Test -", $time);
      $display("%t -      Testname:    test2_overflow", $time);
      $display("%t -      Description: Sets extreme values for operands", $time);
      $display("%t -                   Records the expected results and compares.", $time);
      $display("%t - -------------------------------------------------------", $time);

      $display("%t - Resetting System", $time);
      rst_an  = 1;          #1;                        // to ensure an edge; don't start at 0.
      @(negedge clk)    rst_an <= 0;                   // assert reset on the next negative clock edge
      A_int = 0; B_int = 0;

      $display("%t - Removing Reset", $time);
      @posedge clk)   rst_an <= 1;                     // Remove rst synchronously

      $display("%t - Sets Extreme values for operands and verifies results.", $time);

      @(posedge clk);  #1   A_int = 32767; B_int=127;     // block the next statement
      expected_buffer[clk_counter] = A_int*B_int;   // Don't let this happen before prev line.

      @(posedge clk); #1   A_int = -32768; B_int=127;
      expected_buffer[clk_counter] = A_int*B_int;

      @(posedge clk); #1 A_int = -32768; B_int=-128;
      expected_buffer[clk_counter] = A_int*B_int;

      @(posedge clk); #1 A_int = 32767; B_int=-128;
      expected_buffer[clk_counter] = A_int*B_int;

      repeat(4) @(posedge clk);   // flush out any remaining data from the hardware

      // verify that the expected vectors were found somewhere in the output stream
      check_vectors(4);

```

```
        $display("%t - ---------Extream Input Test Complete --------------------", $time);

    //==================================================================


    @(posedge clk) $finish;

  end  //Test_sequence


endmodule

```

1.5 Tasks and Functions to Search Output Date For Expected Output

```
`ifdef BUFFER_SIZE   // If BUFFER_SIZE is defined, go on, else define it.
`else
  `define BUFFER_SIZE 1023
`endif

`ifdef VERBOSE
`else
  `define VERBOSE 0
`endif

`ifdef ERRORFATAL
`else
  `define ERRORFATAL 1
`endif

integer searchstring_buffer [`BUFFER_SIZE:0];   // temporary buffer for search routine

//  ========= Functions and Tasks for vector searching
// ==================TASK DUMP_VECTORS ====================
task dump_vectors;
  input[31:0] max;
  begin
      $display("========= Dumping Output/Expected Vectors ========= ");
      for (i=0;i<max;i=i+1)
        $display("output_buffer[%d]=%d expected_buffer[%d]=%d",
                 i, output_buffer[i], i, expected_buffer[i] );
  end
endtask

```

```
//=================================================================
// call this from the individual tests to automatically compare the expected_buffer with
// output_buffer
// ================== TASK   CHECK-VECTORS ======================
task check_vectors;
   input [31:0] number_of_words;
   integer i,j;
   begin
      $display(“%t - %m INFO: --- Searching for Expected vectors in the output stream”, $time);
      if(`VERBOSE>1) dump_vectors(number_of_words);
      // skip the first couple expected values since they are often erronious
      for (i=1;i<number_of_words+3;i=i+1)
         searchstring_buffer[i-1] = expected_buffer[i];
      j = search(number_of_words);
   end
endtask


//=================================================================
// to use, first set searchstring_buffer[x:0] = search_value;
//                 output_buffer[y:0] = the string being searched
//  x and y < 1024
//
//    then call      search(number of words in search_value to match);
//
// This function searches for the occurance of one data stream in another
//  - it returns the position of the first string in the second
// =================== FUNCTION   SEARCH ====================
function [31:0] search;
   input [31:0] number_of_words;
   integer i,j, result;
   reg searching;
   begin
      i=0;  j=0; searching = 1;
      result = -1;
      while(searching) begin
         if(i+j >1023) searching = 0; result = -1; // end the search - we didn’t find it
         while(searching & (output_buffer[i+j]==searchstring_buffer[j])) begin
      // we have a temporary match
            if(j==number_of_words-1) begin searching = 0; result = i; end     // MATCHED
            if(i+j>1022) searching = 0;                                // got too far
            j=j+1;
```

```
                end
                i=i+1;
            end
            if(result < 0) begin
                $display(
“%t *-* SEARCH FUNCTION: ERROR - Searchstring (of %0d words) was NOT found in the  comparison buffer.”
 , $time,number_of_words);
                $display(“========= Dumping Output/Expected Vectors ========= “);
                for (i=0;i<clk_counter;i=i+1)
                    $display(“output_buffer[%d]=%d
searchstring_buffer[%d]=%d”,i,output_buffer[i],i,searchstring_buffer[i]);
                if(`ERRORFATAL) begin
                 $display(“ERRORFATAL Set - EXITING with ERROR Status”); $finish; end
            end
            else  $display(
                        “%t *-* SEARCH FUNCTION: SUCCESS - Found searchstring (of %0d words) at
                        position %0d in the comparison buffer.”
                      , $time, number_of_words, result);
            search = result;
        end
    endfunction
```