Processes

Processes are functions identified to the SystemC kernel and called if a signal of the sensitivity list changes.

- Processes implement the funcionality of modules.
- Similar to C++ functions or methods

Three types of Processes: Methods, Threads and Cthreads

- Methods : When activated, executes and returns
  SC_METHOD(process_name)
- Threads: can be suspended and reactivated
  - wait( ) -> suspends
  - one sensitivity list event -> activates

  SC_THREAD(process_name)
- Cthreads: are activated by the clock pulse
  SC_CTHREAD(process_name, clock value);

Processes

Type	SC_METHOD	SC_THREAD	SC_CTHREAD
Activates Exec.	Event in sensit. list	Event in sensit. List	Clock pulse
Suspends Exec.	NO	YES	YES
Infinite Loop	NO	YES	YES
suspended/ reactivated by	N.D.	wait()	wait() wait_until()

Constructor & Sensibility definition	SC_METHOD(call_back);
	sensitive(signals);
	sensitive_pos(signals);
	sensitive_neg(signals);
		SC_THREAD(call_back);
		sensitive(signals);
		sensitive_pos(signals);
		sensitive_neg(signals);
			SC_CTHREAD(
			call_back,
			clock.pos());
			SC_CTHREAD(
			call_back,
			clock.neg());

Sensitivity List of a Process

- sensitive with the ( ) operator

  Takes a single port or signal as argument

  `sensitive(s1);sensitive(s2);sensitive(s3)`

- sensitive with the stream notation

  Takes an arbitrary number of arguments

  `sensitive << s1 << s2 << s3;`

- sensitive_pos with either ( ) or << operator

  Defines sensitivity to positive edge of Boolean signal or clock

  `sensitive_pos << clk;`

- sensitive_neg with either ( ) or << operator

  Defines sensitivity to negative edge of Boolean signal or clock

  `sensitive_neg << clk;`

Multiple Process Example

```
SC_MODULE(ram){
    sc_in<int> addr;
    sc_in<int> datain;
    sc_in<bool> rwb;
    sc_out<int> dout;
    int memdata[64];
            // local memory storage
    int i;
    void ramread(); // process-1
    void ramwrite();// process-2
    SC_CTOR(ram){
        SC_METHOD(ramread);
        sensitive << addr << rwb;
        SC_METHOD(ramwrite);
        sensitive << addr << datain << rwb;
        for (i=0; i++; i<64) {
            memdata[i] = 0;
    } }
};
```

Thread Process and wait() function

- wait( ) may be used in both SC_THREAD and SC_CTHREAD processes but not in SC_METHOD process block
- wait( ) suspends execution of the process until the process is invoked again
- wait(<pos_int>) may be used to wait for a certain number of cycles (SC_CTHREAD only)

In Synchronous process (SC_CTHREAD)

- Statements before the wait( ) are executed in one cycle
- Statements after the wait( ) executed in the next cycle

In Asynchronous process (SC_THREAD)

- Statements before the wait( ) are executed in the last event
- Statements after the wait( ) are executed in the next event

Thread Process and wait() function

```
void do_count() {
    while(1) {
        if(reset) {
            value = 0;
        }
        else if (count) {
            value++;
            q.write(value);
        }
        wait(); // wait till next event !
    }
}
```

Example Code

```
void wait_example:: my_thread_process(void)
{
        wait(10, SC_NS);
        cout << "Now at " << sc_time_stamp() << endl;
        sc_time t_DELAY(2, SC_MS);
        t_DELAY *= 2;
        cout << "Delaying " << t_DELAY<< endl;
        wait(t_DELAY);
        cout << "Now at " << sc_time_stamp()<< endl;
}
```

OUTPUT

Thread Example

```
SC_MODULE(my_module) {
    sc_in<bool> id;
    sc_in<bool> clock;
    sc_in<sc_uint<3> > in_a;
    sc_in<sc_uint<3> > in_b;
    sc_out<sc_uint<3> >
                                    out_c;
    void my_thread();

    SC_CTOR(my_module){
    SC_THREAD(my_thread);
    sensitive << clock.pos();
    }
};
```

Thread Implementation

```
//my_module.cpp
void my_module::
                        my_thread(){
while(true){
  if (id.read())
    out_c.write(in_a.read());
  else
    out_c.write(in_b.read());
  wait();
}
};
```

CThread

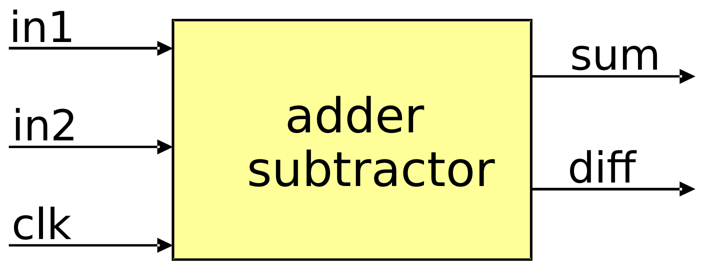
- Almost identical to SC_THREAD, but implements "clocked threads"
- Sensitive only to one edge of one and only one clock
- It is not triggered if inputs other than the clock change

- Models the behavior of unregistered inputs and registered outputs
- Useful for high level simulations, where the clock is used as the only synchronization device
- Adds wait_until( ) and watching( ) semantics for easy deployment.

Counter Example

```
SC_MODULE(countsub)
{
    sc_in<double>  in1;
    sc_in<double>  in2;
    sc_out<double> sum;
    sc_out<double> diff;
    sc_in<bool>    clk;
    void addsub();

    // Constructor:
    SC_CTOR(countsub)
    {
// declare addsub as SC_METHOD
        SC_METHOD(addsub);
        // make it sensitive to
        // positive clock
        sensitive_pos << clk;
    }
};
```




```
// addsub method
void countsub::addsub()
{
    double a;
    double b;
    a = in1.read();
    b = in2.read();
    sum.write(a+b);
    diff.write(a-b);
};
```

sc_main()

The top level is a special function called sc_main.

- It is in a file named main.cpp or main.c
- sc_main() is called by SystemC and is the entry point for your code.
- The execution of sc_main() until the sc_start() function is called.

```
int sc_main (int argc, char *argv []) {
        // body of function
        sc_start(arg) ;
        return 0 ;
}
```

- sc_start(arg) has an optional argument:
  It specifies the number of time units to simulate.
  If it is a null argument the simulation will run forever.

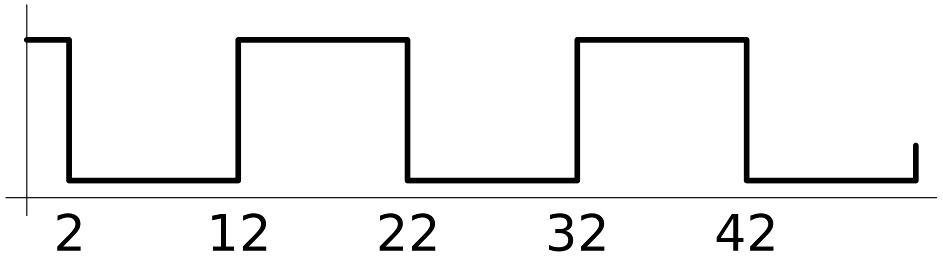
Clocks

- Special object
- How to create ?

  sc_clock clock_name (“clock_label”, period, duty_ratio, offset, initial_value);
- Clock connection

  f1.clk(clk_signal); //where f1 is a module
- Clock example:

sc_clock clock1 ("clock1", 20, 0.5, 2, true);



sc_time

sc_time data type to measure time. Time is expressed in two parts: a numeric magnitude and a time unit e.g. SC_MS, SC_NS, SC_PS, SC_SEC, etc.

sc_time t(20, SC_NS);

//var t of type sc_time with value of 20ns

More Examples:

```
sc_time   t_PERIOD(5, SC_NS) ;
sc_time   t_TIMEOUT (100, SC_MS) ;
sc_time   t_MEASURE, t_CURRENT, t_LAST_CLOCK;
t_MEASURE = (t_CURRENT-t_LAST_CLOCK) ;
if (t_MEASURE > t_HOLD) { error ("Setup violated") }
```

Time representation in SystemC

Set Time Resolution:

sc_set_time_resolution (10, SC_PS) ;

- Any time value smaller than this is rounded off
- default; 1 Peco-Second

sc_time t2(3.1416, SC_NS); // t2 gets 3140 PSEC

To Control Simulation:

sc_start( ) ;

sc_stop( ) ;

To Report Time Information:

sc_time_stamp( ) // returns the current simulation time

cout << sc_time_stamp( ) << endl ;

sc_simulation_time( )

Returns a value of type double with the current simulation time in the current default time unit

sc_event

Event

- Something that happens at a specific point in time.
- Has no value or duration

sc_event:

- A class to model an event
  - Can be triggered and caught.

Important
(the source of a few coding errors):

- Events have no duration → you must be watching to catch it
  - If an event occurs, and no processes are waiting to catch it, the event goes unnoticed.

sc_event

You can perform only two actions with an sc_event:

- wait for it
  - wait(ev1)
  - SC_THREAD(my_thread_proc);
  - sensitive << ev_1; // or
  - sensitive(ev_1)
- cause it to occur

  notify(ev1)

Common misunderstanding:

- if (event1) do_something
  - Events have no value
  - You can test a Boolean that is set by the process that caused an event;
  - However, it is problematic to clear it properly.

notify( )

To Trigger an Event:

event_name.notify(args);

event_name.notify_delayed(args);

notify(args, event_name);

Immediate Notification:

causes processes which are sensitive to the event to be made ready to run in the current evaluate phase of the current delta-cycle.

Delayed Notification:

causes processes which are sensitive to the event to be made ready to run in the evaluate phase of the next delta-cycle.

Timed Notification:

causes processes which are sensitive to the event to be made ready to run at a specified time in the future.

notify( ) Examples

```
sc_event my_event ; // event
sc_time t_zero (0, SC_NS) ; // variable t_zero of type sc_time
sc_time t(10, SC_MS) ; // variable t of type sc_time
```

Immediate

```
my_event.notify();
notify(my_event); // current delta cycle
```

Delayed

```
my_event.notify_delayed();
my_event.notify(t_zero);
notify(t_zero, my_event); // next delta cycle
```

Timed

```
my_event.notify(t);
notify(t, my_event);
my_event.notify_delayed(t); // 10 ms delay
```

cancel ( )

Cancels pending notifications for an event.

- It is supported for delayed and timed notifications.
- not supported for immediate notifications.

Given:

```
sc_event a, b, c;   // events
sc_time t_zero (0,SC_NS);   // variable t_zero of type sc_time
sc_time t(10, SC_MS);   // variable t of type sc_time
...
a.notify();   // current delta cycle
notify(t_zero, b);   // next delta cycle
notify(t, c);   // 10 ms delay
```

Cancel of Event Notification:

```
a.cancel();   // Error! Can't cancel immediate notification
b.cancel();   // cancel notification on event b
c.cancel();   // cancel notification on event c
```

Problem with events

```
SC_MODULE(missing_event) {
    SC_CTOR(missing_event) {
        SC_THREAD(B_thread); // ordered
        SC_THREAD(A_thread); // to cause
        SC_THREAD(C_thread); // problems
    }
    void A_thread( ) {
        a_event.notify( ); // immediate!
        cout << "A sent a_event!" << endl;
    }
    void B_thread() {
        wait(a_event) ;
        cout << "B got a_event!" << endl;
    }
    void C_thread() {
        wait(a_event) ;
        cout << "C got a_event!" << e
    }
  sc_event a_event;
}
```

If wait(a_event) is issued after the immediate notification
a_event.notify( )
Then B_thread and C_thread can wait for ever.
Unless a_avent is issued again.

Properly Ordered Events

```
SC_MODULE(ordered_events){
  SC_CTOR(ordered_events) {
  SC_THREAD(B_thread);
  SC_THREAD(A_thread);
  SC_THREAD(C_thread);
   // ordered to cause problems
   }

void A_thread() {
while (true) {
a_event.notify(SC_ZERO_TIME);
cout << "A sent a_event!" << en
wait(c_event);
cout << "A got c_event!" << end
}  // endwhile
}
```

```
void B_thread() {
while (true) {
b_event.notify(SC_ZERO_TIME);
cout << "B sent b_event!" << endl;
wait(a_event);
cout << "B got a_event!" << endl;
}  // endwhile
}
void C_thread() {
while (true) {
c_event.notify(SC_ZERO_TIME);
cout << "C sent c_event!" << endl;
wait(b_event);
cout << "C got b_event!" << endl;
}  // endwhile
}
sc_event a_event, b_event, c_event;
} ;
```

Time & Execution Interaction

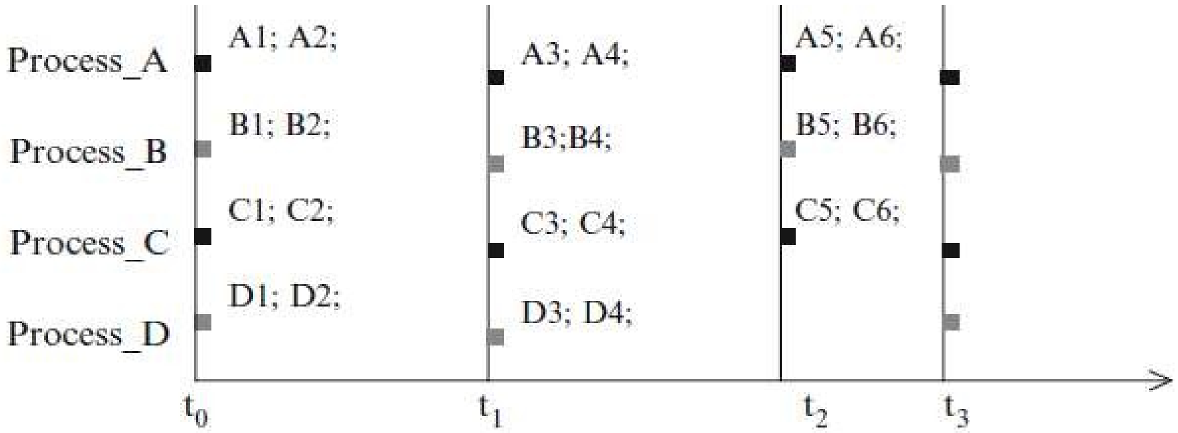
```
Process_A() {
  //@ t_0
  stmt_A1;
  stmt_A2;
  wait(t_1);
  stmt_A3;
  stmt_A4;
  wait(t_2);a
  stmt_A5;
  stmt_A6;
  wait(t_3);
}
```

```
Process_B() {
  //@ t_0
  stmt_B1;
  stmt_B2;
  wait(t_1);
  stmt_B3;
  stmt_B4;
  wait(t_2);
  stmt_B5;
  stmt_B6;
  wait(t_3);
}
```

```
Process_C() {
  //@ t_0
  stmt_C1;
  stmt_C2;
  wait(t_1);
  stmt_C3;
  stmt_C4;
  wait(t_2);
  stmt_C5;
  stmt_C6;
  wait(t_3);
}
```

```
Process_D() {
  //@ t_0
  stmt_D1;
  stmt_D2;
  wait(t_1);
  stmt_D3;
  wait(
   SC_ZERO_TIME);
  stmt_D4;
  wait(t_3);
}
```

Simulated Execution Activity



wait( ) and watching( )

Legacy SystemC code for Clocked Thread

```
wait(N); // delay N clock edges
wait_until (delay_expr); // until expr true @ clock
```

Same as

```
For (i=0; i!=N; i++)
        wait( ) ;           //similar as wait(N)
do wait ( ) while (!expr) ;  // same as
                             // wait_until(delay_expr)
```

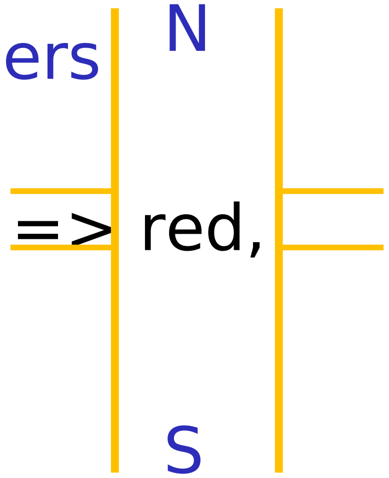
Previous versions of SystemC also included other constructs to watch signals such as watching(),

Traffic Light Controller

Highway

- Normally has a green light.

Sensor:



- A car on the East-West side road triggers the sensor
  - The highway light: green => yellow => red,
  - Side road light: red => green.

SystemC Model:

- Uses two different time delays:
  - green to yellow delay >= yellow to red delay (to represent the way that a real traffic light works).

Traffic Controller Example

```
// traff.h
#include "systemc.h“

SC_MODULE(traff) {

    // input ports
    sc_in<bool> roadsensor;
    sc_in<bool> clock;

    // output ports
    sc_out<bool> NSred;
    sc_out<bool> NSyellow;
    sc_out<bool> NSgreen;
    sc_out<bool> EWred;
    sc_out<bool> EWyellow;
    sc_out<bool> EWgreen;
    void control_lights();
    int i;
```

```
    // Constructor
    SC_CTOR(traff) {
        SC_THREAD(control_lights);
     // Thread
            sensitive << roadsensor;
            sensitive << clock.pos();
    }
};
```

Traffic Controller Example

```
// traff.cpp
#include "traff.h"
void traff::control_lights() {
    NSred = false;
    NSyellow = false;
    NSgreen = true;
    EWred = true;
    EWyellow = false;
    EWgreen = false;
    while (true) {
        while (roadsensor == false)
                wait();
    NSgreen = false;//      road sensor triggered
    NSyellow = true; // set NS to yellow
    NSred = false;
    for (i=0; i<5; i++)
        wait();
    NSgreen = false; //       yellow interval over
    NSyellow = false; // set NS to red
    NSred = true; // set EW to green
    EWgreen = true;
    EWyellow = false;
    EWred = false;
    for (i= 0; i<50; i++)
        wait();
```

```
        NSgreen = false; // times up for EW green
        NSyellow = false; // set EW to yellow
        NSred =  true ;
        EWgreen = false;
        EWyellow = true;
        EWred = false;
        for (i=0; i<5; i++)
                        // times up for EW yellow
            wait();
        NSgreen = true; // set EW to red
        NSyellow = false; // set NS to green
        NSred = false;
        EWgreen = false;
        EWyellow = false;
        EWred = true;
        for (i=0; i<50; i++) // wait one more long
            wait(); // interval before allowing
                          // a sensor again
        }
}
```

JPEG Compression/Decompression using SystemC

Overview

- Introduction to JPEG Coding and Decoding
- Hardware-Software Partitioning
- FDCT and IDCT HW module for 8 x 8 Block
- JPEG Implementation

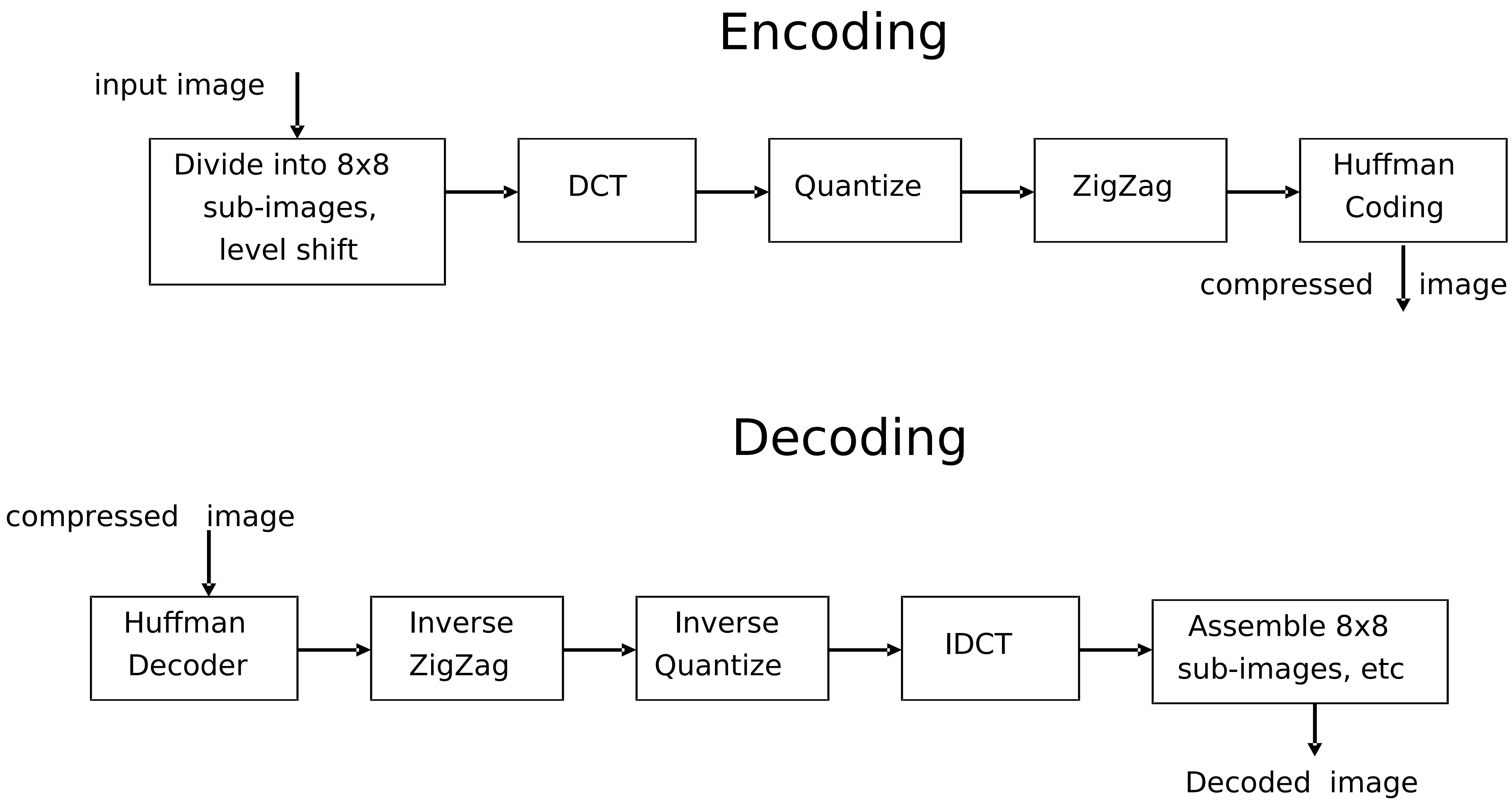
JPEG- -**based Encoding**

Four Stages of JPEG Compression

- Preprocessing and dividing an image into 8 x 8 blocks

  Level-shift, for 8-bit gray scale images, subtract 128 from each pixel i.e. pixel[i] = pixel[i] – 128 ;
- DCT, Discrete Cosine Transform of 8 x 8 image blocks.
- Quantization
- ZigZag
- Entropy Encoding either of:
  - Huffman coding
  - Variable Length Coding

JPEG Encoding and Decoding



DCT: Discrete Cosine Transform

Mathematical definitions of 8 x 8 DCT and 8 x 8 IDCT respectively.

$$F(u,v) = \frac{1}{4} C(u)\, C(v) \left[\sum_{x=0}^{7} \sum_{y=0}^{7} f(x,y) * \cos((2x+1)u)/16 * \cos((2y+1)v /16 \right]$$

$$f(x,y) = \frac{1}{4} \left[\sum_{u=0}^{7} \sum_{v=0}^{7} C(u)\, C(v) F(u,v) * \cos((2x+1)u)/16 * \cos((2y+1)v /16 \right]$$

where $C(u), C(v) = 1/\sqrt{2}$ for $u,v = 0$

$C(u), C(v) = 1$ otherwise

F(u,v) is the Discrete Cosine Transform of 8 x 8 block

f(x,y) is the Inverse Discrete Cosine Transform

Why DCT instead of DFT

DCT is similar to DFT with many advantages

- DCT coefficients are purely real
- Near-optimal for energy compaction
- DCT computation is efficient due to faster algorithms
- Hardware solutions available that do not need multipliers

DCT is extensively used in image compression standards including, JPEG, MPEG-1, MPEG-2, MPEG-4, etc.

$$\begin{pmatrix} 52 & 55 & 61 & 66 & 70 & 61 & 64 & 73 \\ 63 & 59 & 66 & 90 & 109 & 85 & 69 & 72 \\ 62 & 59 & 68 & 113 & 144 & 104 & 66 & 73 \\ 63 & 58 & 71 & 122 & 154 & 106 & 70 & 69 \\ 67 & 61 & 68 & 104 & 126 & 88 & 68 & 70 \\ 79 & 65 & 60 & 70 & 77 & 68 & 58 & 75 \\ 85 & 71 & 64 & 59 & 55 & 58 & 65 & 83 \\ 87 & 79 & 69 & 68 & 65 & 65 & 78 & 94 \end{pmatrix} \xrightarrow{\text{DCT}} \begin{pmatrix} -415 & -29 & -62 & 25 & 55 & -20 & -1 & 3 \\ 7 & -21 & -62 & 9 & 11 & -7 & -6 & 6 \\ -46 & 8 & 77 & -25 & -30 & 10 & 7 & -5 \\ -50 & 13 & 35 & -15 & -9 & 6 & 0 & 3 \\ 11 & -8 & -13 & -2 & -1 & 1 & -4 & 1 \\ -10 & 1 & 3 & -3 & -1 & 0 & 2 & -1 \\ -4 & -1 & 2 & -1 & 0 & -3 & 1 & -2 \\ -1 & -1 & -1 & -2 & -3 & -1 & 0 & -1 \end{pmatrix}$$

Quantization

The 8x8 block of DCT transformed values is divided by a quantization value for each block entry.

$$F_{quantized}(u,v) = F(u,v) / Quantization\_Table(x,y)$$

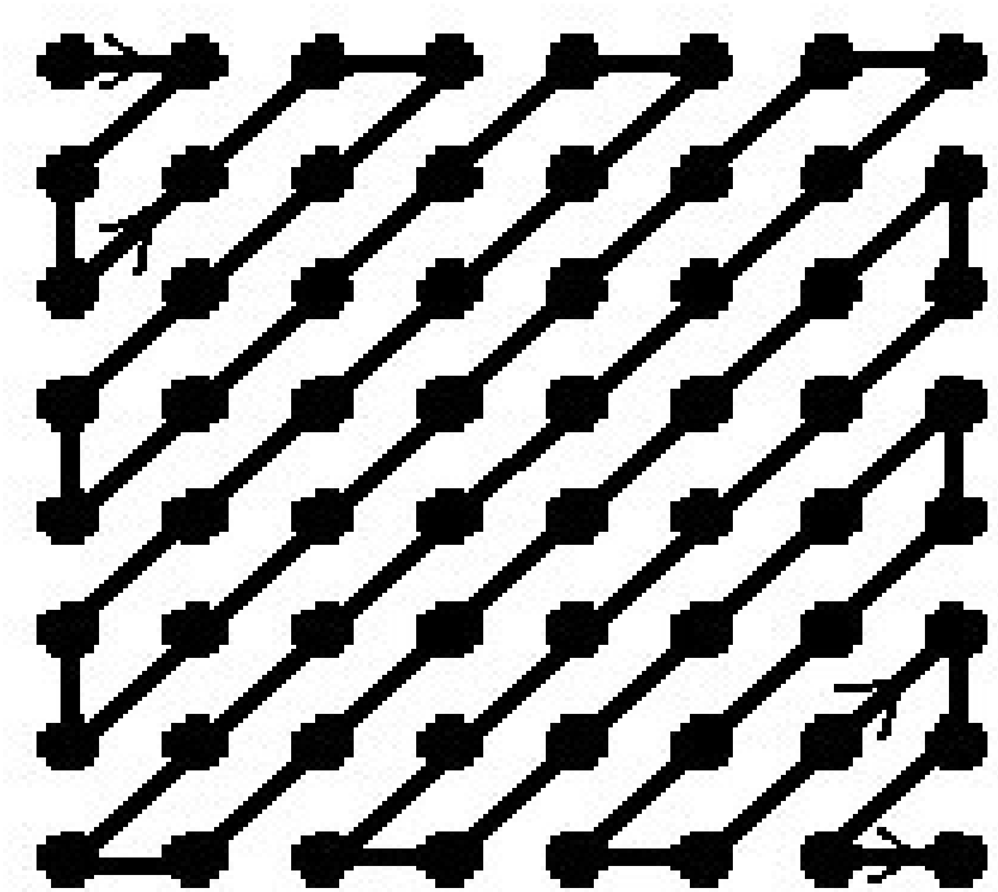
Quantization table :

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Zig-Zag

It takes the quantized 8x8 block and orders it in a 'Zig-Zag' sequence, resulting in a 1-D array of 64 entries,

- This process place low-frequency coefficients (larger values) before the high-frequency ones (nearly zero).
- One can ignore any continuous zeros at the end of block
- Insert a (EOB) at the end of each 8x8 block encoding .



Quantization and ZigZag

$$
\begin{pmatrix}
-415 & -29 & -62 & 25 & 55 & -20 & -1 & 3 \\
7 & -21 & -62 & 9 & 11 & -7 & -6 & 6 \\
-46 & 8 & 77 & -25 & -30 & 10 & 7 & -5 \\
-50 & 13 & 35 & -15 & -9 & 6 & 0 & 3 \\
11 & -8 & -13 & -2 & -1 & 1 & -4 & 1 \\
-10 & 1 & 3 & -3 & -1 & 0 & 2 & -1 \\
-4 & -1 & 2 & -1 & 0 & -3 & 1 & -2 \\
-1 & -1 & -1 & -2 & -3 & -1 & 0 & -1
\end{pmatrix}
\qquad
\begin{pmatrix}
16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\
12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\
14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\
14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\
18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\
24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\
49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\
72 & 92 & 95 & 98 & 112 & 100 & 103 & 99
\end{pmatrix}
$$

$$
\begin{pmatrix}
-26 & -3 & -6 & 2 & 2 & 0 & 0 & 0 \\
1 & -2 & -4 & 0 & 0 & 0 & 0 & 0 \\
-3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\
-4 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
$$

Zig-Zag

[-26 -3 1 -3 -2 -6 2 – 4 1 -4 1 1 5 0 2
0 0 -1 2 0 0 0 0 0 -1 -1 EOB]

FDCT SC_Module

```
struct fdct : sc_module {{
    sc_out<double> out64[8][8]; // the dc transformed 8x8 block
    sc_in<double> fcosine[8][8]; // cosine table input
    sc_in<FILE *> sc_input; // input file pointer port
    sc_in<bool> clk; // clock signal
    char input_data[8][8]; // the data read from the input file
    void read_data( void ); // read the 8x8 block
    void calculate_dct( void ); // perform dc transform
    // define fdct as a constructor
    SC_CTOR( fdct ) {
    // read_data method sensitive to +ve & calculate_dct sensitive to
    // -ve clock edge, entire read and dct will take one clock cycle
            SC_METHOD( read_data ); // define read_data as a method
            dont_initialize();
            sensitive << clk.pos;
            SC_METHOD( calculate_dct );
            dont_initialize();
            sensitive << clk.neg;
    }
}};
```

DCT Module

```
#include "fdct.h"
void fdct :: calculate_dct( void ) {
unsigned char  u, v, x, y;
double                          temp;
     for (u = 0; u < 8; u++) // do forward discrete cosine transform
         for (v = 0; v < 8; v++)                    {     temp = 0.0;
                for (x = 0; x < 8; x++)
                for (y = 0; y < 8; y++)
                    temp += input_data[x][y] * fcosine[x][u].read() *
                                fcosine[y][v].read();
                if ((u == 0) && (v == 0)) temp /= 8.0;
                else if (((u == 0) && (v != 0)) || ((u != 0) && (v == 0)))
                    temp /= (4.0*sqrt(2.0)); else temp /= 4.0;
                out64[u][v].write(temp);
}      }}
void fdct :: read_data( void )                              {{ // read the 8*8 block
     fread(input_data, 1, 64, sc_input.read());
     // shift from range [0, 2^8 - 1] to [2^(8-1), 2^(8-1) - 1]
     for (unsigned char uv = 0; uv < 64; uv++)
                input_data[uv/8][uv%8] -= (char) (pow(2,8-1));
}}
```

DCT Module Structures

```
#define PI 3.1415926535897932384626433832795 // the value of PI
unsigned char quant[8][8] =                                   // quantization table
            {{16,11,10,16,24,40,51,61},
             {12,12,14,19,26,58,60,55},
             {14,13,16,24,40,57,69,56},
             {14,17,22,29,51,87,80,62},
             {18,22,37,56,68,109,103,77},
             {24,35,55,64,81,104,113,92},
             {49,64,78,87,103,121,120,101},
             {72,92,95,98,112,100,103,99}};
unsigned char zigzag_tbl[64]={                                // zigzag table
            0,1,5,6,14,15,27,28,
            2,4,7,13,16,26,29,42,
            3,8,12,17,25,30,41,43,
            9,11,18,24,31,40,44,53,
            10,19,23,32,39,45,52,54,
            20,22,33,38,46,51,55,60,
            21,34,37,47,50,56,59,61,
            35,36,48,49,57,58,62,63};
signed char MARKER = 127; // end of block marker
```

Functions: Read File Header

```
#define rnd(x) (((x)>=0)?((signed char)((signed char)((x)+1.5)-
    1)):((signed char)((signed char)((x)-1.5)+1)))
#define rnd2(x) (((x)>=0)?((short int)((short int)((x)+1.5)-
    1)):((short int)((short int)((x)-1.5)+1)))

// read the header of the bitmap and write it to the output file

void write_read_header(FILE *in, FILE *out)                {{
    unsigned char temp[60]; // temporary array of 60 characters,
                // which is enough for the bitmap header: 54 bytes
    printf("\nInput Header read and written to the output file");
    fread(temp, 1, 54, in); // read 54 bytes and store them in temp
    fwrite(temp, 1, 54, out); // write 54 bytes to the output file
    printf("......Done\n");
    printf("Image is a %d bit Image. Press Enter to Continue\n>",
    temp[28]);
    getchar();
}
```

Functions: Cosine-table

```
// make the cosine table
void make_cosine_tbl(double cosine[8][8]);
void make_cosine_tbl(double cosine[8][8])                    {{
    printf("Creating the cosine table to be used in FDCT and
    IDCT");
    // calculate the cosine table as defined in the formula
    for (unsigned char i = 0; i < 8; i++)
            for (unsigned char j = 0; j < 8; j++)
                        cosine[i][j] = cos((((2*i)+1)*j*PI)/16);
    printf("......Done\n");
}
```

Functions: ZigZag

```
// zigzag the quantized input data

void zigzag_quant(double data[8][8], FILE *output)                                    {
      signed char to_write[8][8];
           // this is the rounded values, to be written to the file
      char last_non_zero_value = 0; // index to last non-zero in a block
        // zigzag data array & copy it to to_write, round the values
       // and find out the index to the last non-zero value in a block
      for (unsigned char i = 0; i < 64; i++)                                {{
        to_write[zigzag_tbl[i]/8][zigzag_tbl[i]%8] =
                   rnd(data[i/8][i%8] /   quant[i/8][i%8]);
                   if (to_write[i/8][i%8] != 0) last_non_zero_value = i;
     }}
      // write all values in the block including the last non-zero value
      for (unsigned char i = 0; i <= last_non_zero_value; i++)
                fwrite(&to_write[i/8][i%8], sizeof(signed char), 1, output);
                     // write the end of block marker
      fwrite(&MARKER, sizeof(signed char), 1, output);
}}
```

Functions: Main

```
#include "systemc.h"
#include "functions.h"
#include "fdct.h"
#include "idct.h"
#define NS *1e-9 // constant for clock signal is in nanoseconds
int sc_main(int argc, char *argv[]) {
    char choice;
    sc_signal<FILE *> sc_input; // input file pointer signal
    sc_signal<FILE *> sc_output; // output file pointer signal
    sc_signal<double> dct_data[8][8]; // signal to the dc transformed
    sc_signal<double> cosine_tbl[8][8]; // signal for cos-table values
    sc_signal<bool> clk1, clk2; // clock signal for FDCT and IDCT
    FILE *input, *output; // input and output file pointers
    double cosine[8][8]; // cosine table
    double data[8][8]; // data read from signals to be zigzagged
    if (argc == 4) {
        if (!(input = fopen(argv[1], "rb")))
                // some error occurred while trying to open the input file
        printf("\nSystemC JPEG-LAB:\nCannot Open File '%s'\n",argv[1]),
                                                                    exit(1);
```

Functions - - Main

```
write_read_header(input, output);
                    // write the header read from the input file
make_cosine_tbl(cosine); // make the cosine table

// copy cosine and quantization tables onto corresponding signals
for (unsigned char i = 0; i < 8; i++)
        for (unsigned char j = 0; j < 8; j++)
                    cosine_tbl[i][j].write(cosine[i][j]);

fdct FDCT("fdct"); // call the forward discrete transform module
// bind the ports
for (unsigned char i = 0; i < 8; i++)
        for (unsigned char j = 0; j < 8; j++)                       {
             FDCT.out64[i][j](dct_data[i][j]);
             FDCT.fcosine[i][j](cosine_tbl[i][j]);
        }
FDCT.clk(clk1);
FDCT.sc_input(sc_input);
```

Functions: Main

cont.

```
// we must use two different clocks. That will make sure that when
// we want to compress, we only compress and don't decompress it
    sc_start(SC_ZERO_TIME);                    // initialize the clock
    if ((choice == 'c') || (choice == 'C'))                  {{ // for compression
            while (!(feof(input)))                {{ // create the FDCT clock signal
                clk1.write(1);                      // convert the clock to high
                sc_start(10, SC_NS);  // cycle high for 10 nanoseconds
                clk1.write(0);                     // start the clock as low
                sc_start(10, SC_NS); //cycle low for 10 nanoseconds
                        // read all the signals into the data variable
                        // to use these values in a software block
                    for (unsigned char i = 0; i < 8; i++)
                        for (unsigned char j = 0; j < 8; j++)
                                data[i][j] = dct_data[i][j].read();
                zigzag_quant(data, output);
                        // zigzag and quantize the read data
            }
     }
}
```