Hardware-Software Co-Design
SystemC: Co-specification & Modeling

EE8205: Embedded Computer Systems
http://www.ee.ryerson.ca/~courses/ee8205/

Dr. Gul N. Khan
http://www.ee.ryerson.ca/~gnkhan
Electrical and Computer Engineering
Ryerson University

Overview
- Hardware-Software (HW/SW) Co-design
- SystemC and Co-specification
- Introduction to JPEG Coding and Decoding
- HW/SW Partitioning and JPEG Implementation

Introductory Articles on Hardware-Software Codesign, SystemC documents available at the course webpage, Part of Chapter 7 of the Text by Wolf
Introduction

Embedded computers are the processing devices.

- Home appliances and entertainment units
- Transportation including automobiles
- Medical instrumentation
- Wireless communication devices,
- Jet engines and other aerospace/space application
- Industrial control, nuclear systems and many more

By many estimates embedded computers make up 99% of worldwide computers

Embedded Computer Systems are the ideal candidate for hardware-software codesign.
Embedded System Design

- Separate HW and SW design has been explored and examined very thoroughly
- Joint design remains an area of rapidly growing study
- Old embedded devices always built from scratch
  – within reasonable amount of time
- Components - smaller and faster - IP cores
- Tools required for the product engineer.
Hardware-Software Codesign

- Functional exploration: Define a desired product's requirements and produce a specification of the system behavior.
- Map this specification onto various hardware and software architectures.
- Partition the functions between silicon and code; and map them directly to hardware or software components.
- Integrate system for prototype test.
HS-Codesign

• Co-Specification: Describe system functionality at the abstract level

• System description is converted into a task graph representation

• HW-SW Partitioning: Take the task graph and decide which components are implemented where/how?
  i.e. Dedicated hardware,
  Software -- one CPU or multiple CPUs
HS-Codesign

- HW-SW Co-Synthesis: Analyze the task graph and decide on the system architecture. (incorporates HW/SW partitioning as heart of co-synthesis process)

- HW-SW Co-Simulation: Simulate embedded device’s functionality before prototype construction.

- Co-Verification: Mathematical or simulation based verification that device meets requirements.
Traditional Embedded System Design

• HW/SW Partitioning performed at an early stage.
• Design mistakes have huge negative effect
• Inability to correct mistakes performed at the partitioning phase
Hardware-Software Codesign

Co-design of (embedded) computer systems encompassing the following parts:

- **Co-Specification**
  Developing system specification that describes hardware, software modules and relationship (interface) between the hardware and software

- **Co-Synthesis**
  Automatic and semi-automatic design of hardware and software modules to meet the specification

- **Co-Simulation and Co-verification**
  Simultaneous simulation of hardware and software
HW/SW Co-Specification

- Model the (embedded) system functionality from an abstract level.
- No concept of hardware or software yet.
- Common environment
  SystemC: based on C++.
- Specification is analyzed to generate a task graph representation of the system functionality.
Co-Specification

- A system design language is needed to describe the functionality of both software and hardware.
- The system is first defined without making any assumptions about the implementation.
- A number of ways to define new specification standards grouped in three categories:
  1) A new language
  2) Extend a hardware design language
  3) Make C/C++ language to work for hardware designers

- SystemC: An open source library in C++ that provides a modeling platform for systems with hardware and software components.
SystemC for Co-specification

Open SystemC Initiative (OSCI) 1999 by EDA venders including Synopsys, ARM, CoWare, Fujitsu, etc.

- A C++ based modeling environment containing a class library and a standard ANSI C++ compiler.
- SystemC provides a C++ based modeling platform for exchange and codesign of system-level intellectual property (IP) models.

- **SystemC is not an extension to C++**
  - SystemC 1.0 and 2.0 versions

It has a new C++ class library

User needs to learn the use of new classes to model hardware design
**SystemC Library Classes**

SystemC classes enable the user to
- Define modules and processes
- Add inter-process/module communication through ports and signals.

Modules/processes can handle a multitude of data types: Ranging from bits to bit-vectors, standard C++ types to user define types like structures

Modules and processes also introduce timing, concurrency and reactive behavior.
- Using SystemC requires knowledge of C and very little of C++ and the approach is similar to VHDL/Verilog.
SystemC 2.0 Language Architecture

- **Standard Channels for Various MOC's**
  - Kahn Process Networks
  - Static Dataflow, etc.

- **Methodology-Specific Channels**
  - Master/Slave Library, etc.

- **Elementary Channels**
  - Signal, Timer, Mutex, Semaphore, Fifo, etc.

- **Core Language**
  - Modules
  - Ports
  - Processes
  - Interfaces
  - Channels
  - Events

- **Data Types**
  - Logic Type (01XZ)
  - Logic Vectors
  - Bits and Bit Vectors
  - Arbitrary Precision Integers
  - Fixed Point Integers

- **C++ Language Standard**
SystemC 2.0 Language Architecture

- All of SystemC builds on C++
- Upper layers are cleanly built on top of the lower layers
- The SystemC core language provides a minimal set of modeling constructs for structural description, concurrency, communication, and synchronization.
- Data types are separate from the core language and user-defined data types are fully supported.
- Commonly used communication mechanisms such as signals and FIFOS can be built on top of the core language. The MOCs can also be built on top of the core language.
- If desired, lower layers can be used without needing the upper layers.
SystemC Benefits

SystemC 2.0 allows the following tasks to be performed within a single language:

- Complex system specifications can be developed and simulated.
- System specifications can be refined to mixed software and hardware implementations.
- Hardware implementations can be accurately modeled at all the levels.
- Complex data types can be easily modeled, and a flexible fixed-point numeric type is supported.
- The extensive knowledge, infrastructure and code base built around C and C++ can be leveraged.
SystemC Counter Code

```c
struct counter : sc_module {
  // the counter module
  sc_inout<int> in;  // the input/output port of int type
  sc_in<bool> clk;  // Boolean input port for clock
  void counter_fn(); // counter module function
  
  SC_CTOR( counter ) {
    
    SC_METHOD( counter_fn ); // declare the counter_fn as a method
    dont_initialize(); // don't run it at first execution
    sensitive_pos << clk; // make it sensitive to +ve clock edge
  }
}

// software block that check/reset the counter value, part of sc_main
void check_for_10(int *counted) {
  if (*counted == 10) {
    printf("Max count (10) reached ... Reset count to Zero\n");
    *counted = 0;
  }
}
```
void check_for_10(int *counted);
int sc_main(int argc, char *argv[])
{
    sc_signal<int> counting; // the signal for the counting variable
    sc_clock clock("clock",20, 0.5); // clock period = 20 duty cycle = 50%
    int counted; // internal variable, to store the value in counting signal
    counting.write(0); // reset the counting signal to zero at start
    counter COUNT("counter"); // call counter module
    COUNT.in(counting); // map the ports by name
    COUNT.clk(clock); // map the ports by name
    for (unsigned char i = 0; i < 21; i++)
    {
        counted = counting.read(); // copy the signal onto the variable
        check_for_10(&counted); // call the software block & check for 10
        counting.write(counted); // copy the variable onto the signal
        sc_start(20); // run the clock for one period
    }
    return 0;
}
Counter Main Code with Tracing

```c
int sc_main(int argc, char *argv[]) {
    sc_signal<int> counting; // the signal for the counting variable
    sc_clock clock("clock", 20, 0.5); // clock; time period = 20 duty cycle = 50%
    int counted; // internal variable, to stores the value in counting signal

    // create the trace- file by the name of "counter_tracefile.vcd"
    sc_trace_file *tf = sc_create_vcd_trace_file("counter_tracefile");

    // trace the clock and the counting signals
    sc_trace(tf, clock.signal(), "clock");
    sc_trace(tf, counting, "counting");

    counting.write(0); // reset the counting signal to zero at start

    counter COUNT("counter"); // call counter module. COUNT is just a temp var
    COUNT.in(counting); // map the ports by name
    COUNT.clk(clock); // map the ports by name

    for (unsigned char i = 0; i < 21; i++) {
        
    }

    sc_close_vcd_trace_file(tf); // close the tracefile

    return 0;
}
```
#include "systemc.h"
#define COUNTER
struct counter : sc_module {   // the counter module
    sc_inout<int> in; // the input/output port of int type
    sc_in<bool> clk; // Boolean input port for clock
    void counter_fn(); // counter module function
    SC_CTOR( counter ) { // counter constructor
        SC_METHOD( counter_fn ); // declare the counter_fn as a method
        dont_initialize(); // don’t run it at first execution
        sensitive_pos << clk; // make it sensitive to +ve clock edge
    }
};
void counter :: counter_fn() {
    in.write(in.read() + 1); // add one to the port
    printf("in=%d\n", in.read()); // print it for viewing
}
Signals are like variables used for programming hardware modules very close to VHDL.

```
sc_signal<data_type> sig_name;
```

- create a signal of type ‘data_type’ and name it ‘sig_name’.
- hardware module has its own input and output ports to which these signals must be mapped or bound.

**Ports** declaration is similar to signals e.g.

```
sc_inout<data_type> prt_name;
```

- create an input-output port of ‘data_type’ with name ‘prt_name’.
- ‘sc_in and sc_out create input and output ports respectively.

There are two types of port binding: named and positional.

- **named binding** binds each port to a signal (one to one)
- **positional binding** binds the ports by their position that is quite unsafe.
SystemC Module

**Module:** is a basic object in SystemC that includes ports, constructors, data members, function members and may be internal memory storage and internal functions.

```
SC_MODULE( module ) ; or struct module: sc_module { … };
```

**Constructor:** creates the internal data structure of a module each module should include a constructor that identifies processes as methods using the SC_METHOD macro.

```
SC_METHOD ( funct ) ;
```

Identifies the function or process funct

A Method needs to be made sensitive to some internal or external signal.

```
e.g. sensitive_pos << clock or sensitive_neg << clock
```

Methods are called similar to normal C as:

```
function_type module_name::function_name(data_type var_name) { … }
```

**SC_METHOD** process is triggered by events and executes all the statements in it before returning control to the SystemC kernel.
SystemC Module

SC_MODULE(fifo) {
  sc_in<bool> load;
  sc_in<bool> read;
  sc_inout<int> data;
  sc_out<bool> full;
  sc_out<bool> empty;

  // rest of module not shown
}
Other Co-Specification Languages

UML

– Commonly used for modeling software systems.
– Supports many modeling specifications and profiles.
  e.g. UML Profile for real-time systems

ESTEREL

– To model reactive real-time systems where events trigger actions and vice versa.

SDL (systems, blocks, channels etc.)

– Formal Specification and Description Language.
– Often used to model real-time embedded systems.
JPEG-based Encoding

Four Stages of JPEG Compression
• Preprocessing and dividing an image into $8 \times 8$ blocks
  Level-shift, for 8-bit gray scale images, subtract 128 from each pixel i.e. $\text{pixel}[i] = \text{pixel}[i] - 128$;
• DCT, Discrete Cosine Transform of $8 \times 8$ image blocks.
• Quantization
• ZigZag
• Entropy Encoding either of:
  – Huffman coding
  – Variable Length Coding
JPEG Encoding and Decoding

Encoding

- Divide into 8x8 sub-images, level shift
- DCT
- Quantize
- ZigZag
- Huffman Coding

Decoding

- Compressed image
- Huffman Decoder
- Inverse ZigZag
- Inverse Quantize
- IDCT
- Assemble 8x8 sub-images, etc

Decoded image
DCT: Discrete Cosine Transform

Mathematical definitions of $8 \times 8$ DCT and $8 \times 8$ IDCT respectively.

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

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

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 $8 \times 8$ 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.

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

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

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

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 places 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

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

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

Zig-Zag

\[
\begin{pmatrix}
-26 & -3 & 1 & -3 & -2 & -6 & 2 & -4 & 1 & -4 & 1 & 1 & 5 & 0 & 2 \\
0 & 0 & -1 & 2 & 0 & 0 & 0 & 0 & -1 & -1 & EOB
\end{pmatrix}
\]
FDCT SC_Module

```c
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_pos << clk;
        SC_METHOD( calculate_dct );
dont_initialize();
sensitive_neg << clk;
    }
};
```
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));
}
#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},
 {49,64,78,87,103,121,120,77},
 {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 the data array and 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

// 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_initialize();       // 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_cycle(10 NS); // cycle the high for 10 nanoseconds
        clk1.write(0); // start the clock as low
        sc_cycle(10 NS); // cycle the 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
    }
}