

# EECS 222A: System-on-Chip Description and Modeling Lecture 4

Rainer Dömer

doemer@uci.edu

The Henry Samueli School of Engineering  
Electrical Engineering and Computer Science  
University of California, Irvine

## Lecture 4: Overview

- Language Semantics
- Execution and Simulation Semantics
  - Motivating Examples
- Simulation Semantics
  - Discrete Event Simulator
- Formal Execution Semantics
  - Time-Interval Formalism
  - Abstract State Machines
- Project Discussion
  - Digital Camera Example
  - JPEG Application
  - Assignment 2

## Language Semantics

- Concepts found in Embedded Systems
  - Behavioral and structural hierarchy
  - Concurrency
  - Synchronization and communication
  - Exception handling
  - Timing
  - State transitions
- SLDL must support these concepts
- Language semantics needed to define the *meaning*
  - Semantics of execution (modeling, simulation, synthesis)
  - Deterministic vs. non-deterministic behavior
  - Preemptive vs. non-preemptive concurrency
  - Atomic operations

EECS222A: SoC Description and Modeling, Lecture 4

(c) 2009 R. Doemer

3

## Language Semantics

- Language semantics are needed for
  - System designer (understanding)
  - Tools
    - Validation (compilation, simulation)
    - Formal verification (equivalence, property checking)
    - Synthesis
  - Documentation and standardization
- Objective:
  - Clearly define the execution semantics of the language
- Requirements and goals:
  - completeness
  - precision (no ambiguities)
  - abstraction (no implementation details)
  - formality (enable formal reasoning)
  - simplicity (easy understanding)

EECS222A: SoC Description and Modeling, Lecture 4

(c) 2009 R. Doemer

4

## Language Semantics

- Example: SpecC language
  - Documentation
    - Language Reference Manual (LRM)
      - ⇒ set of rules written in English (not formal)
    - Abstract simulation algorithm
      - ⇒ set of valid implementations (not general)
  - Reference implementation
    - SpecC Reference Compiler and Simulator
      - ⇒ one instance of a valid implementation (not general)
    - Compliance test bench
      - ⇒ set of specific test cases (incomplete)
  - Formal execution semantics
    - Time-interval formalism
      - ⇒ rule-based formalism (incomplete)
    - Abstract State Machines
      - ⇒ fully formal approach (not easy to understand)

EECS222A: SoC Description and Modeling, Lecture 4

(c) 2009 R. Doemer

5

## Execution and Simulation Semantics

- Motivating Example 1

- Given:

```
behavior B1(int x)
{
  void main(void)
  {
    x = 5;
  }
};
```

```
behavior B2(int x)
{
  void main(void)
  {
    x = 6;
  }
};
```

```
behavior B
{
  int x;
  B1 b1(x);
  B2 b2(x);

  void main(void)
  {
    b1; b2;
  }
};
```

- What is the value of x after the execution of B?

– Answer: x = 6

EECS222A: SoC Description and Modeling, Lecture 4

(c) 2009 R. Doemer

6

## Execution and Simulation Semantics

- Motivating Example 2

– Given:

```
behavior B1(int x)
{
  void main(void)
  {
    x = 5;
  }
};
```

```
behavior B2(int x)
{
  void main(void)
  {
    x = 6;
  }
};
```

```
behavior B
{
  int x;
  B1 b1(x);
  B2 b2(x);

  void main(void)
  {
    par{b1; b2;}
  }
};
```

– What is the value of x after the execution of B?

– Answer: The program is non-deterministic!  
(x may be 5, or 6, or any other value!)

## Execution and Simulation Semantics

- Motivating Example 3

– Given:

```
behavior B1(int x)
{
  void main(void)
  {
    waitfor 10;
    x = 5;
  }
};
```

```
behavior B2(int x)
{
  void main(void)
  {
    x = 6;
  }
};
```

```
behavior B
{
  int x;
  B1 b1(x);
  B2 b2(x);

  void main(void)
  {
    par{b1; b2;}
  }
};
```

– What is the value of x after the execution of B?

– Answer: x = 5

## Execution and Simulation Semantics

- Motivating Example 4

– Given:

```
behavior B1(int x)
{
  void main(void)
  {
    waitfor 10;
    x = 5;
  }
};
```

```
behavior B2(int x)
{
  void main(void)
  {
    waitfor 10;
    x = 6;
  }
};
```

```
behavior B
{
  int x;
  B1 b1(x);
  B2 b2(x);

  void main(void)
  {
    par{b1; b2;}
  }
};
```

– What is the value of x after the execution of B?

– Answer: The program is non-deterministic!  
(x may be 5, or 6, or any other value!)

## Execution and Simulation Semantics

- Motivating Example 5

– Given:

```
behavior B1(
  int x, event e)
{
  void main(void)
  {
    x = 5;
    notify e;
  }
};
```

```
behavior B2(
  int x, event e)
{
  void main(void)
  {
    wait e;
    x = 6;
  }
};
```

```
behavior B
{
  int x;
  event e;
  B1 b1(x,e);
  B2 b2(x,e);

  void main(void)
  {
    par{b1; b2;}
  }
};
```

– What is the value of x after the execution of B?

– Answer: x = 6

## Execution and Simulation Semantics

- Motivating Example 6

– Given:

```
behavior B1(
  int x, event e)
{
  void main(void)
  {
    notify e;
    x = 5;
  }
};
```

```
behavior B2(
  int x, event e)
{
  void main(void)
  {
    wait e;
    x = 6;
  }
};
```

```
behavior B
{
  int x;
  event e;
  B1 b1(x,e);
  B2 b2(x,e);

  void main(void)
  {
    par{b1; b2;}
  }
};
```

– What is the value of x after the execution of B?

– Answer: x = 6

## Execution and Simulation Semantics

- Motivating Example 7

– Given:

```
behavior B1(
  int x, event e)
{
  void main(void)
  {
    waitfor 10;
    x = 5;
    notify e;
  }
};
```

```
behavior B2(
  int x, event e)
{
  void main(void)
  {
    wait e;
    x = 6;
  }
};
```

```
behavior B
{
  int x;
  event e;
  B1 b1(x,e);
  B2 b2(x,e);

  void main(void)
  {
    par{b1; b2;}
  }
};
```

– What is the value of x after the execution of B?

– Answer: x = 6

## Execution and Simulation Semantics

- Motivating Example 8

- Given:

```
behavior B1(
  int x, event e)
{
  void main(void)
  {
    x = 5;
    notify e;
  }
};
```

```
behavior B2(
  int x, event e)
{
  void main(void)
  {
    waitfor 10;
    wait e;
    x = 6;
  }
};
```

```
behavior B
{
  int x;
  event e;
  B1 b1(x,e);
  B2 b2(x,e);

  void main(void)
  {
    par{b1; b2;}
  }
};
```

- What is the value of x after the execution of B?

- Answer: B never terminates!  
(the event is lost)

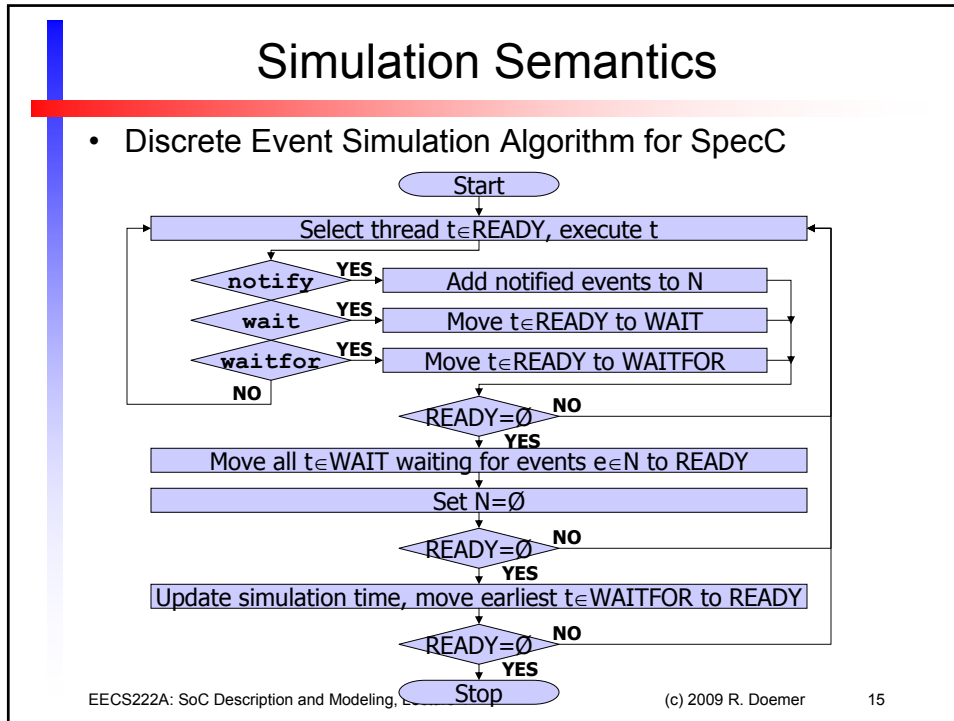
## Simulation Semantics

- Discrete Event Simulation Algorithm for SpecC

- available in LRM (appendix), good for understanding
- ⇒ set of valid implementations
- ⇒ not general (possibly incomplete)

- Definitions:

- At any time, each thread t is in one of the following sets:
  - **READY**: set of threads ready to execute (initially root thread)
  - **WAIT**: set of threads suspended by `wait` (initially  $\emptyset$ )
  - **WAITFOR**: set of threads suspended by `waitfor` (initially  $\emptyset$ )
- Notified events are stored in a set **N**
  - `notify e1` adds event `e1` to **N**
  - `wait e1` will wakeup when `e1` is in **N**
  - Consumption of event `e` means event `e` is taken out of **N**
  - Expiration of notified events means **N** is set to  $\emptyset$



- ### Simulation Semantics
- Discrete Event Simulation Algorithm for SpecC
    - Conforms to general Discrete Event (DE) Simulation
      - utilizes *delta-cycle* mechanism (i.e. inner event loop)
      - matches execution semantics of other languages
        - SystemC
        - VHDL
        - Verilog
    - Features
      - clearly specifies the simulation semantics
      - is easily understandable
      - can easily be implemented
    - Generality
      - is one valid implementation of the semantics
      - other valid implementations may exist as well
- EECS222A: SoC Description and Modeling, Lecture 4 (c) 2009 R. Doemer 16



## Formal Execution Semantics

- Two examples of semantics definition:
  - 1) Time-interval formalism
    - formal definition of timed execution semantics
    - sequentiality, concurrency, synchronization
    - allows reasoning over execution order, dependencies
  - 2) Abstract State Machines
    - complete execution semantics of SpecC V1.0
      - wait, notify, notifyone, par, pipe, traps, interrupts
      - operational semantics (no data types!)
    - influence on the definition of SpecC V2.0
    - straightforward extension for SpecC V2.0

EECS222A: SoC Description and Modeling, Lecture 4

(c) 2009 R. Doemer

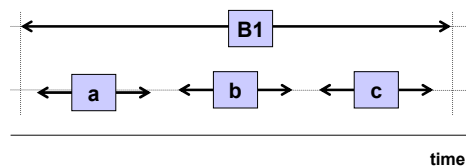
17

## Formal Execution Semantics

- Time-interval formalism
  - Definition of execution semantics of SpecC 2.0
    - sequential execution
    - concurrent execution (semantics of **par**)
    - synchronization (semantics of **notify**, **wait**)
  - Sequential execution

```
behavior B1
{ void main(void)
  { a;
    b;
    c;
  }
};
```

$$Tstart(B1) \leq Tstart(a) < Tend(a) \leq Tstart(b) < Tend(b) \leq Tstart(c) < Tend(c) \leq Tend(B1)$$



EECS222A: SoC Description and Modeling, Lecture 4

(c) 2009 R. Doemer

18

## Formal Execution Semantics

- Time-interval formalism
  - Sequential execution
    - waitfor rule:
      - only `waitfor` increases simulation time
      - other statements execute in zero simulation time

```
behavior B
{ void main(void)
  { a;
    waitfor 10;
    b;
  }
};
```

$$0 \leq Tstart(a) < Tend(a) < 1$$

$$0 \leq Tstart(w) < Tend(w) = 10$$

$$10 \leq Tstart(b) < Tend(b) < 11$$

time

EECS222A: SoC Description and Modeling, Lecture 4
(c) 2009 R. Doemer
19

## Formal Execution Semantics

- Time-interval formalism
  - Concurrent execution
    - Preemptive or non-preemptive scheduling:  
No atomicity guaranteed!

```
behavior B
{ void main(void)
  { par{ b1; b2; }
  }
};
```

```
behavior B1
{ void main(void)
  { a; b; c; }
};
```

```
behavior B2
{ void main(void)
  { d; e; f; }
};
```

$$Tstart(B) \leq Tstart(a) < Tend(a) \leq Tend(B)$$

$$Tstart(B) \leq Tstart(b) < Tend(b) \leq Tend(B)$$

$$Tstart(B) \leq Tstart(c) < Tend(c) \leq Tend(B)$$

$$Tstart(B) \leq Tstart(d) < Tend(d) \leq Tend(B)$$

$$Tstart(B) \leq Tstart(e) < Tend(e) \leq Tend(B)$$

$$Tstart(B) \leq Tstart(f) < Tend(f) \leq Tend(B)$$

Possible Schedule

time

EECS222A: SoC Description and Modeling, Lecture 4
(c) 2009 R. Doemer
20

## Formal Execution Semantics

- Time-interval formalism
  - Synchronization

```
behavior B
{ void main(void)
  { par{ b1; b2;}
  }
};

behavior B1
{ void main(void)
  { a; wait e; b; }
};

behavior B2
{ void main(void)
  { c; notify e; d; }
};
```

$$Tstart(B) \leq Tstart(a) < Tend(a) \leq Tstart(w) < Tend(w) \leq Tstart(b) < Tend(b) \leq Tend(B)$$

$$Tstart(B) \leq Tstart(c) < Tend(c) \leq Tstart(n) < Tend(n) \leq Tstart(d) < Tend(d) \leq Tend(B)$$

$Tend(w) \geq Tend(n)$

time

EECS222A: SoC Description and Modeling, Lecture 4
(c) 2009 R. Doemer
21

## Formal Execution Semantics

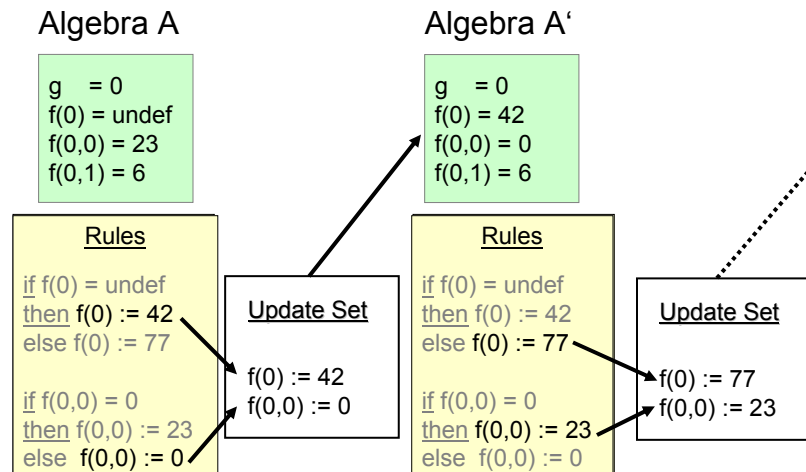
- Time-interval formalism
  - Atomicity
    - Since there is no atomicity guaranteed, a safe mechanism for mutual exclusion is necessary
    - SpecC 2.0: Channels behave as *Monitors!*
      - A *mutex* is implicitly contained in each channel instance
      - Each channel method implicitly
        - » *acquires* the mutex when it starts execution, and
        - » *releases* the mutex again when it finishes
      - `wait` and `waitfor` statements implicitly (and atomically!)
        - » *release* an acquired mutex in a channel, and
        - » *re-acquire* the mutex before execution resumes
      - This easily enables safe communication without heavy restrictions to the implementation!

EECS222A: SoC Description and Modeling, Lecture 4
(c) 2009 R. Doemer
22

## Formal Execution Semantics

- Abstract State Machine (ASM)
  - aka. Evolving Algebras (Y. Gurevich, 1987)
  - ASM semantics already exist for
    - Prolog, Concurrent Prolog
    - C, C++, Java
    - VHDL, VHDL-AMS, SystemC
  - ASM semantics for SpecC published at ISSS'02
- ASM components
  - Sequence of algebras (functions over domains): *states*
  - Rules define updates of functions: *state transitions*

## ASM: Abstract State Machine



## ASM: SpecC Kernel Semantics

- Phase 1: **at least one BEHAVIOR is running**
- Phase 2: **no BEHAVIOR is running**

```

graph TD
    Start(( )) --> ExecuteBehaviors[ExecuteBehaviors]
    ExecuteBehaviors --> ProcessEvents[ProcessEvents]
    ProcessEvents --> CheckResetEvents[Check/ResetEvents]
    CheckResetEvents -- if events --> ExecuteBehaviors
    CheckResetEvents -- if no events --> AdvanceTime[AdvanceTime]
    AdvanceTime --> ProcessTimeouts[ProcessTimeouts]
    ProcessTimeouts --> ExecuteBehaviors
    AdvanceTime -- exit --> Exit(( ))
    
```

EECS222A: SoC Description and Modeling, Lecture 4
(c) 2009 R. Doemer
25

## ASM: SpecC Behavior Semantics

$p \in \text{BEHAVIOR}$ :  
 $\text{status}(p) \in \{\text{running}, \text{waiting}, \text{interrupted}, \text{completed}\}$

```

graph TD
    running([running]) -- last stmt --> completed([completed])
    running -- wait, waitfor, fork --> waiting([waiting])
    waiting -- event, timeout, join --> running
    waiting -- trap --> interrupted([interrupted])
    interrupted -- interrupt --> waiting
    interrupted -- last stmt --> completed
    
```

EECS222A: SoC Description and Modeling, Lecture 4
(c) 2009 R. Doemer
26

## ASM: SpecC Statement Semantics

- **modelling execution of statements of behavior "Self"**  
 Self executes  $\langle \text{statement} \rangle \equiv$   
 $\text{programCounter}(\text{Self}) = \langle \text{statement} \rangle \wedge \text{status}(\text{Self}) = \text{running}$
- **wait statement**  
 if Self executes  $\langle \text{wait}(\text{EventList}) \rangle$   
then  $\text{status}(\text{Self}) := \text{waiting}$ ,  
 $\text{sensitivity}(\text{Self}) := \text{EventList}$ ,  
 $\text{programCounter}(\text{Self}) := \text{nextStmt}(\text{Self})$   
endif;
- **notify statement**  
 if Self executes  $\langle \text{notify}(\text{EventList}) \rangle$   
then  $\forall e \in \text{EventList}: \text{notified}(e) := \text{true}$ ,  
 $\text{programCounter}(\text{Self}) := \text{nextStmt}(\text{Self})$   
endif;
- The simulation kernel sets each behavior to  
 $\text{status}(b) := \text{running}$  if  $\exists e: \text{notified}(e) = \text{true} \wedge e \in \text{sensitivity}(b)$

EECS222A: SoC Description and Modeling, Lecture 4

(c) 2009 R. Doemer

27

## ASM: SpecC Semantics Summary

- **Formal Semantics of SpecC Execution**
  - complete execution semantics of SpecC V1.0 by ASMs
    - wait, notify, notifyone, par, pipe, traps, interrupts
    - operational semantics (no data types!)
  - can be easily extended to V2.0
  - influenced the definition of SpecC V2.0
  - SpecC ASM specification is comparable to other ASM specifications
    - SystemC
    - VHDL 93

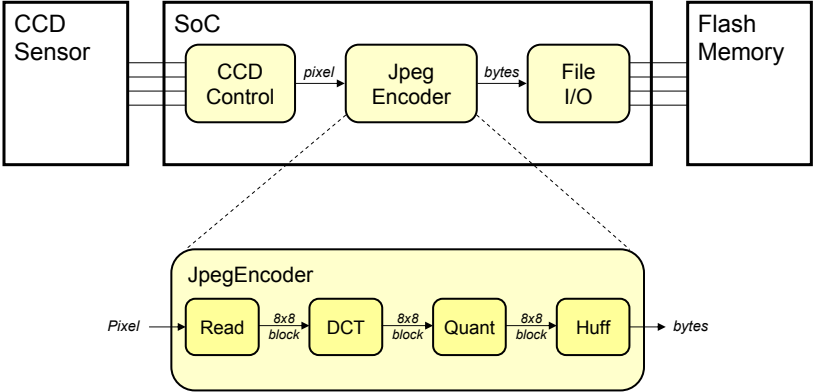
EECS222A: SoC Description and Modeling, Lecture 4

(c) 2009 R. Doemer

28

## Project Discussion

- Digital Camera Example
  - Component Model



The diagram illustrates the component model of a digital camera. It shows a CCD Sensor connected to a SoC (System on Chip). Inside the SoC, there is a CCD Control block that receives data from the sensor and sends 'pixel' data to a Jpeg Encoder. The Jpeg Encoder then sends 'bytes' to a File I/O block, which is connected to Flash Memory. A detailed view of the Jpeg Encoder shows a pipeline: Pixel input goes to a Read block, which outputs an 8x8 block to a DCT block. The DCT block outputs another 8x8 block to a Quant block, which outputs a third 8x8 block to a Huff block. The Huff block finally outputs bytes.

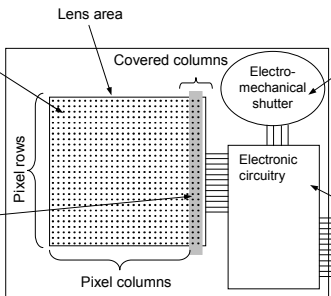
EECS222A: SoC Description and Modeling, Lecture 4 (c) 2009 R. Doemer 29

## Project Discussion

- Digital Camera Example
  - Charge-Coupled Device (CCD)
    - Special sensor that captures an image
    - Light-sensitive silicon solid-state device composed of many cells

When exposed to light, each cell becomes electrically charged. This charge can then be converted to a 8-bit value where 0 represents no exposure while 255 represents very intense exposure of that cell to light.

Some of the columns are covered with a black strip of paint. The light-intensity of these pixels is used for zero-bias adjustments of all the cells.



The diagram shows a grid of pixel rows and pixel columns. A lens area is positioned above the grid. Some columns are covered by a black strip. An electro-mechanical shutter is located to the right of the grid, and electronic circuitry is positioned below it.

The electro-mechanical shutter is activated to expose the cells to light for a brief moment.

The electronic circuitry, when commanded, discharges the cells, activates the electro-mechanical shutter, and then reads the 8-bit charge value of each cell. These values can be clocked out of the CCD by external logic through a standard parallel bus interface.

Source: T. Givargis, F. Vahid. "Embedded System Design", Wiley 2002.

EECS222A: SoC Description and Modeling, Lecture 4 (c) 2009 R. Doemer 30

## Project Discussion

- Digital Camera Example
  - Image Compression
  - JPEG (Joint Photographic Experts Group)
    - Popular standard format for representing digital images in a compressed form
    - Provides for a number of different modes of operation
    - Mode used in this chapter provides high compression ratios using DCT (discrete cosine transform)
    - Image data divided into blocks of 8 x 8 pixels
    - 3 steps performed on each block
      - DCT
      - Quantization
      - Huffman encoding

*Source: T. Givargis, F. Vahid. "Embedded System Design", Wiley 2002.*

## Project Discussion

- Digital Camera Example
  - Discrete Cosine Transform (DCT)
  - Transforms original 8 x 8 block into a cosine-frequency domain
    - Upper-left corner values represent low frequency components
      - Essence of image
    - Lower-right corner values represent finer details
      - Can reduce precision of these values and retain reasonable image quality
  - FDCT (Forward DCT) formula
    - $C(h) = \text{if } (h == 0) \text{ then } 1/\sqrt{2} \text{ else } 1.0$ 
      - Auxiliary function used in main function  $F(u,v)$
    - $F(u,v) = \frac{1}{4} \times C(u) \times C(v) \sum_{x=0..7} \sum_{y=0..7} D_{xy} \times \cos(\pi(2u + 1)u/16) \times \cos(\pi(2y + 1)v/16)$ 
      - Gives encoded pixel at row  $u$ , column  $v$
      - $D_{xy}$  is original pixel value at row  $x$ , column  $y$
  - IDCT (Inverse DCT)
    - Reverses process to obtain original block (not needed for this design)

*Source: T. Givargis, F. Vahid. "Embedded System Design", Wiley 2002.*



## Project Discussion

- Digital Camera Example
  - Quantization
  - Achieve high compression ratio by reducing image quality
    - Reduce bit precision of encoded data
      - Fewer bits needed for encoding
      - One way is to divide all values by a factor of 2
        - » Simple right shifts can do this
    - Dequantization would reverse process for decompression

1150	39	-43	-10	26	-83	11	41
-81	-3	115	-73	-6	-2	22	-5
14	-11	1	-42	26	-3	17	-38
2	-61	-13	-12	36	-23	-18	5
44	13	37	-4	10	-21	7	-8
36	-11	-9	-4	20	-28	-21	14
-19	-7	21	-8	3	3	12	-21
-5	-13	-11	-17	-4	-1	7	-4

After DCT

Divide each cell's  
value by 8

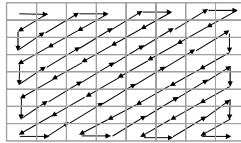
144	5	-5	-1	3	-10	1	5
-10	0	14	-9	-1	0	3	-1
2	-1	0	-5	3	0	2	-5
0	-8	-2	-2	5	-3	-2	1
6	2	5	-1	1	-3	1	-1
5	-1	-1	-1	3	-4	-3	2
-2	-1	3	-1	0	0	2	-3
-1	-2	-1	-2	-1	0	1	-1

After quantization

Source: T. Givargis, F. Vahid. "Embedded System Design", Wiley 2002.

## Project Discussion

- Digital Camera Example
  - Huffman Encoding
  - Serialize 8 x 8 block of pixels
    - Values are converted into single list using zigzag pattern



- Perform Huffman encoding
  - More frequently occurring pixels assigned short binary code
  - Longer binary codes left for less frequently occurring pixels
- Each pixel in serial list converted to Huffman encoded values
  - Much shorter list, thus compression

Source: T. Givargis, F. Vahid. "Embedded System Design", Wiley 2002.

