System Design of Digital Camera Using SpecC

Aseem Gupta
Rainer Dömer

Technical Report CECS-04-32
December 10, 2004

Center for Embedded Computer Systems
University of California, Irvine
Irvine, California 92697-3425, USA
Email: {aseemg,doemer}@uci.edu

System Design of Digital Camera Using SpecC

Aseem Gupta
Rainer Dömer

Technical Report CECS-04-32
December 10, 2004

Center for Embedded Computer Systems
University of California, Irvine
Irvine, California 92697-3425, USA
Email: {aseemg,doemer}@uci.edu

Abstract

The quality of a system design process is influenced by the semantics and syntax of the System Level Design Language (SLDL) adopted as the implementation vehicle. SpecC and SystemC are the two most popular SLDLs used today. In this work, we have demonstrated a system design flow using SpecC by use of a digital camera example. We also discuss how the choice of SystemC would have influenced the design process. A brief comparison between SpecC and SystemC is provided as a result. In order to emphasize the powerful design utility of SpecC and the System-on-Chip Environment (SCE), we have deliberately chosen to use various models of execution in the design model, including sequential, FSM, concurrent and pipelined execution, and communication using channels.

Contents

1. Introduction 1

2. Design Flow 1

3. Digital Camera as System under Design 3
3.1. System Description of Digital Camera 3

4. Specification Model of Digital Camera 4

5. Profiling 6

6. C/C++ Based Design Methodologies 6

7. Results and Conclusion 8

8. Appendix 9

List of Figures

1 Design Flow [5] 2
2 FSM for DigiCam 2
3 Charge Coupled Device 3
4 Zero Bias Adjustment 3
5 After DCT and Quantization 4
6 Huffman Encoding 4
7 Top Level System Chart 5
8 DigiCam behavior FSM 5
9 Procs behavior pipelined execution 5
10 *GetImg* behavior 6
11 *ZBA* behavior 6
12 *SAVE* behavior 6
13 Hierarchy of Behaviors in System 7
14 Digicam Profile 7
15 Digicam Code Profile 7
16 Digicam Computation Profile 8
17 Digicam Connections 8

System Design of Digital Camera Using SpecC

Aseem Gupta and Rainer Dömer
Center for Embedded Computer Systems
University of California, Irvine
Irvine, California 92697-3425, USA
Email: {aseemg,doemer}@uci.edu

Abstract

The quality of a system design process is influenced by the semantics and syntax of the System Level Design Language (SLDL) adopted as the implementation vehicle. SpecC and SystemC are the two most popular SLDLs used today. In this work, we have demonstrated a system design flow using SpecC by use of a digital camera example. We also discuss how the choice of SystemC would have influenced the design process. A brief comparison between SpecC and SystemC is provided as a result. In order to emphasize the powerful design utility of SpecC and the System-on-Chip Environment (SCE), we have deliberately chosen to use various models of execution in the design model, including sequential, FSM, concurrent and pipelined execution, and communication using channels.

1. Introduction

In recent years, growing system complexity and shrinking time-to-market requirements have resulted in a strong need for new design methods and tools. In order to keep pace with the increased system complexity, designers must work at a higher level of abstraction [1]. Depending on the abstraction level (namely, the number of details used to model the system) different concerns can be addressed and solved. At each step of the design process, the key to cope with complexity is to model the systems, only with the minimum number of details needed. Abstraction hides complexity and accelerates design process. Tools support is needed throughout all steps of the design flow, from the formal specification of the system to its physical implementation.

Traditionally, the design of embedded systems has been carried out by decomposing and allocating the system to hardware and software, then allowing separate hardware and software design teams to design their respective parts, and finally integrating hardware and software. This separation of design tasks leads to the potential for initial design mistakes to be carried until the integration phase, where they are much more difficult and costly to correct. This issue has been widely addressed by development of high level languages, that describe both hardware and software, thus keeping their design flow tightly coupled [2].

Given system functionality the goal is to find the best architecture and the best partitioning of functionality into the architectural components. Here, the term architecture is used to mean not only the set of hardware and software components forming the system but also their topology. Starting from the same specification, many different architectures and functionality-architecture mapping may be produced. The exploration of all theses alternatives requires the ability to rapidly estimate the performance resulting from a particular partitioning. In order to evaluate performance, we cannot afford to synthesize and simulate at the cycle level, every possible design alternative. The use of a C/C++ based methodology simplifies the system modeling task and maintains computation time within feasible ranges [3]. As a result: 1) design assessment can be done much earlier in the design cycle, and 2) execution time to explore different design tradeoffs is much shorter. In this project, we have used C based methodology to model our system.

2. Design Flow

Designing an embedded system requires many capabilities [4]: 1) describing the interaction between the system and the external environment, 2) describing the system architecture, 3) modeling the behavior of hardware and software components forming the system, 4) describing system constraints and requirements, 5) describing the test scenarios used to simulate the system, and 6) defining a set of gauges to measure various performance metrics during simulation execution. As a consequence, the complexity of the design process is determined by the semantic and syntax of the system level design language (SLDL) adopted as implementation vehicle. System level design approaches can be

broadly classified into three groups: system-level synthesis, platform-based design, and component-based design [5]. In system-level synthesis the design starts by describing system behavior. Then system architecture is generated by the behavior and finally a register transfer level (RTL) model or an instruction set simulation (ISS) model are generated depending whether the behavior is going to be mapped on hardware or software. In platform based design the system behavior is mapped to a predefined system architecture, instead of being generated from the behavior as in the system level synthesis approach. In component-based design the task of selecting components and combining them in a proper architecture is not defined a priori. Compared with platform based design this solution provides a higher flexibility, however it requires a well developed database of components (also known as intellectual properties or virtual components) before it can be effectively implemented.

In general a SLDL requires two essential attributes: 1) it should support modeling at all levels of abstraction, from purely behavioral un-timed models to cycle accurate RTL/ISS models, and 2) the models should be executable and simulatable, so that functionality and constraints can be validated. The two most commonly used SLDL in embedded system engineering are: SystemC [8] and SpecC [9]. In this work, we have demonstrated the design flow using SpecC and have provided a brief comparison between SpecC and SystemC.

The objective of system level design is to generate system implementation from behavior. To that end, we propose a design process based on the use of finite state machines as mathematical model of computation to describe behavior, and either SystemC or SpecC as implementation vehicle. In order to reduce the complexity of system design a number of intermediate models are built. Each intermediate model describes specific design tasks and objectives and can be independently executed and simulated.

The design process belongs to the system-level synthesis group and is illustrated in Figure 1. Here, we decompose the design process in four main steps: 1) specification modeling, 2) architecture modeling, 3) communication modeling and 4) implementation modeling. The specification model is a formal description of the system functionality, but does not carry any implementation details, and it is un-timed in terms of both computation and communication. After the specification model is analyzed and validated the system functionality is partitioned and the various partitions are mapped to different components. The architecture model defines the final set of components into which the functionality is mapped and its topology. The execution delays of the processes assigned to the components are modeled by means of unit delta delays, while communication among components is modeled via message passing. Hence

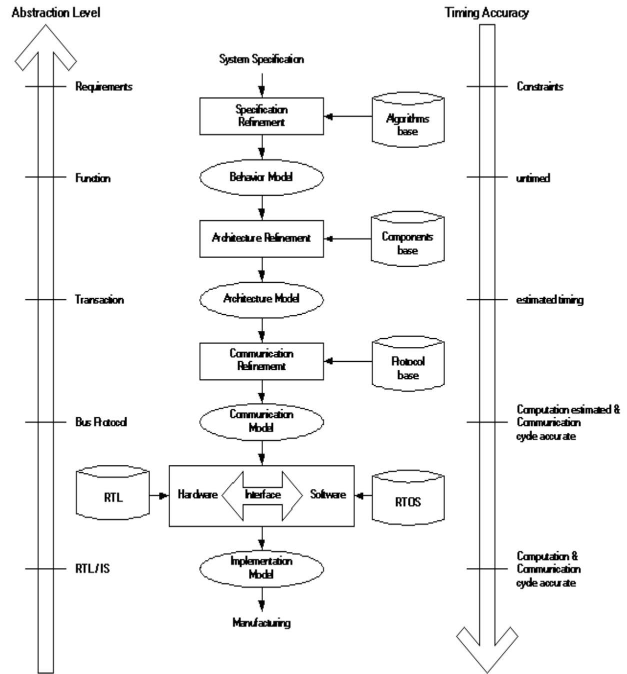


Figure 1. Design Flow [5]

at this level, computation is approximate-timed, while communication is un-timed. The communication model defines the protocol and the accurate timing followed by the various components to exchange information. The implementation model represents the hardware components in terms of resister transfers and the software components in terms of instruction set architecture. At this level, computation components as well as communication components are refined down to individual clock cycles.

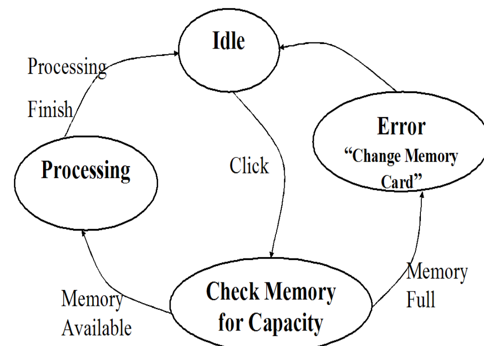


Figure 2. FSM for DigiCam

3. Digital Camera as System under Design

To achieve the goal of design using SpecC, we chose to design a digital camera [10] as an example. The motivation, behind choosing digital camera as an example, is its complexity as an embedded system and it being in vogue. This year, almost 50M units were sold worldwide, which is expected to grow to 80M units in the next 4 years. The first digital camera, offered nothing more to the user, than being able to take a few low quality images, and store them for future transfer to a PC. In last few years, digital cameras have reduced in size and grown in complexity. Image compression algorithms allow more pictures to be taken, advanced image enhancing features like red eye reduction, night mode, auto focus, digital as well as optical zoom have all made the digital camera overtake traditional film cameras in capability.

3.1. System Description of Digital Camera

This project aims to design a basic electronic device that can capture images and store them in digital format. From a user's point of view, a simple digital camera works as follows: The user turns on the camera, points it towards the object and clicks the shutter button. The user can take images as long as there is space available in the memory. [10] has a very simple description of digital camera's functionality. From a designer's point of view, the operation of digital camera can be described as a Finite State Machine, shown in Figure 2. The camera is turned ON and waits in *Idle* state, for the user to click the shutter button. Upon click, the camera first checks the memory for enough available free space to store an image, in *Check Memory for Capacity* state. If there is not enough memory, the camera prompts user to change the memory card of the camera, in *Error* state. If there is space available, the digital camera goes into a *Processing* state, where it processes the image and stores it in memory. After the processing is complete, digital camera goes into *Idle* state, waiting for more clicks from the user.

Let us see the *Processing* state from the perspective of a designer. At the time of click by user, provided memory check is successful, the camera is initiated to the task of taking and storing images. The life of image begins in the *Charge Coupled Device* (CCD). A CCD is a special light sensor with many light-sensitive silicon solid-state cells. The light falling on each cell is converted into a small amount of electric charged, which is measured by CCD electronics and stored as a number. The number usually ranges from 0, meaning no light, to 256 or 65,535, meaning very intense light per pixel. The internals of a CCD are shown in Figure 3. In order to take the image, the CCD must be initialized. This process is sometimes also called warming up. The electromechanical shutter opens briefly, and allows the light to fall on the CCD. The CCD pixels capture image as a measure of the light incident on it. The electronic circuitry dedicated to the operation of CCD, compute the digital equivalent of the charge produced by CCD cells and store the numbers in a buffer. The image, at this step, is just a 8x8 block of numbers. The image can be popped up from the buffer by other processing elements of the digital camera for error correction, compressing and saving in memory.

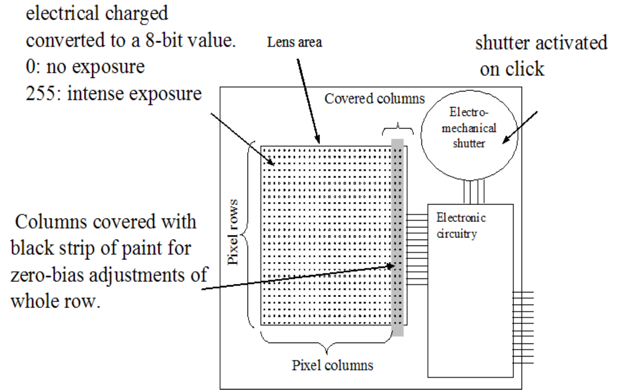


Figure 3. Charge Coupled Device

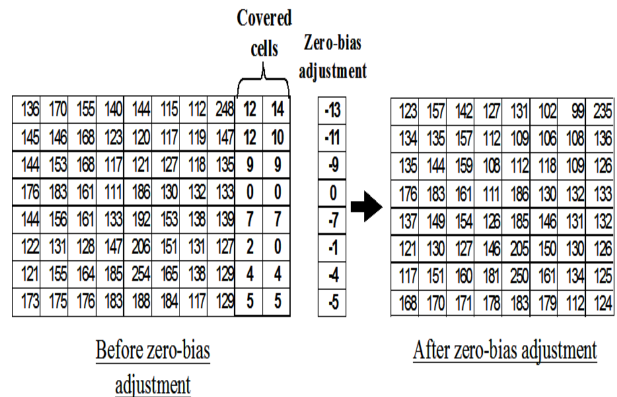


Figure 4. Zero Bias Adjustment

As shown in Figure 3, some columns of the CCD are covered with black paint. This prevents any light from falling on CCD. This is done for Zero Bias Adjustment of image. Due to inherent manufacturing defects, CCD cells never measure the incident light accurately. This error is called Zero Bias error. Usually, it is observed, that the error is same across all the cells that are arranged in the same row, but varies with cells in different rows. Since, the ideal reading for the obscured pixels should be zero, a value other

than that would indicate the magnitude of error. In our example, two columns on the right are used for providing us with the reference error value. We average the error over the two columns and subtract from all other values in the same column. The left array in Figure 4, shows the image captured by the CCD as a 8 X 8 block of pixels. The right array, shows the image after zero bias adjustment.

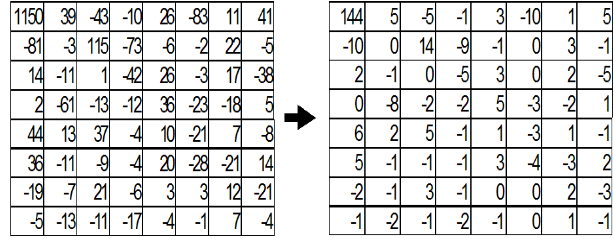
Before the image can be uploaded to a PC, it must be stored in internal memory of the digital camera. The internal memory or the on-board memory of digital camera is limited, hence compression of image is done before storing in the memory. The hardware needed for this extra step of compression is justified by the increase in the capacity of the digital camera to take more number of pictures. The most popular image compression technique is the JPEG standard for Joint Photographic Experts Group [11]. There are many different flavors of image compression, but we have selected compression using Discrete Cosine Transformation (DCT). There are three steps in compressing the image. The first step is to do Discrete Cosine Transformation (DCT). DCT operation is mathematically defined as:
$C(h) = 1/\sqrt{2}$ if $h = 0$, else 1.

$C(h)$ being the auxiliary function used in main function $F(u, v) =$

$1/4 \cdot C(u) \cdot C(v) \cdot \Sigma_{x=0..7}\Sigma_{y=0..7}(D_{xy} \cdot cos(\pi(2u + 1)u/16) \cdot cos(\pi(2y + 1)v/16))$

$F(u, v)$ gives the encoded pixel at row u and column v, from pixel D_{xy}. Inverse DCT can be performed by the PC at its end, using the converse of above formula. The array on left side of Figure 5 results after DCT on the image. The second step is to do Quantization. Though Quantization reduces the quality of the image, it helps in compressing the image as well. The pixels will occupy less space, if their representation is by smaller numbers. After DCT, a pixel may be represented by numbers in the range of few thousands. We divide the values by an exponent of 2. This is because, it allows to take advantage of the simplicity of division by 2 in hardware. A right shift of the bits, divides the number by 2. In our example, we chose to divide by 8 to perform quantization. The block after quantization by 8 is shown in the right side array of Figure 5. The third step is to perform Huffman encoding. Huffman encoding is a minimal variable-length encoding based on the frequency of each pixel. The details of constructing the Huffman tree can be found in relevant literature [12]. The encoded Huffman tree is shown in Figure 6. Once the Huffman encoded image is available, we can store the image in the internal memory of the digital camera.

Storing the image in memory involves synchronization

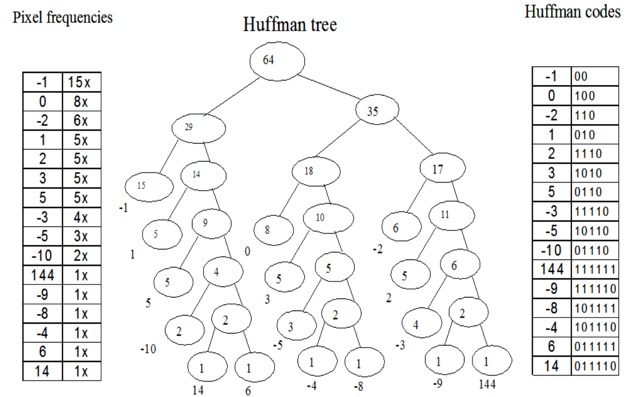


1150	39	-43	-10	26	-83	11	41
-81	-3	115	-73	-6	-2	22	-5
14	-11	1	-42	26	-3	17	-38
2	-61	-13	-12	36	-23	-18	5
44	13	37	-4	10	-21	7	-8
36	-11	-9	-4	20	-28	-21	14
-19	-7	21	-6	3	3	12	-21
-5	-13	-11	-17	-4	-1	7	-4

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

Figure 5. After DCT and Quantization

with the controller-circuit of the memory. The "write" time of the internal memory may not be equal to the time, taken by the camera circuit to send data. In other words, the circuit must send a byte of data to the memory for storing, only when the memory has finished writing the last byte of data. This synchronization guarantees prevention of loss of data. Another advantage is that, this makes the memory as a separate component. Another memory with a better technology, can easily replace existing memory technology, without requiring any change in other circuitry of the camera.



-1	15x
0	8x
-2	6x
1	5x
2	5x
3	5x
5	5x
-3	4x
-5	3x
-10	2x
144	1x
-9	1x
-8	1x
-4	1x
6	1x
14	1x

-1	00
0	100
-2	110
1	010
2	1110
3	1010
5	0110
-3	11110
-5	10110
-10	01110
144	111111
-9	111110
-8	101111
-4	101110
6	011111
14	011110

Figure 6. Huffman Encoding

4. Specification Model of Digital Camera

In this section, we include the charts generated by the System-on-chip (SCE) environment alongside. Figure 7 shows the complete system, which includes test bench and stimulus generator. The block, "Digicam", represents our system under design. Figure 8 shows the FSM inside the behavior DigiCam. It has four behaviors, each corresponding to a state in the FSM shown in Figure 2.

The processing of the image is a complex sequence of functions, that start from initializing the charge coupled device (CCD) to storing the processed image in internal memory. We define the execution of these functions in pipelined order. SpecC language provides explicit support for the

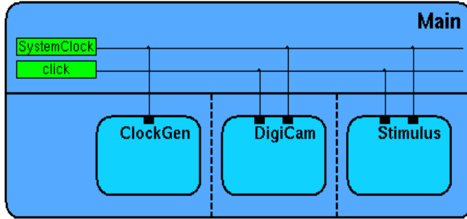


Figure 7. Top Level System Chart

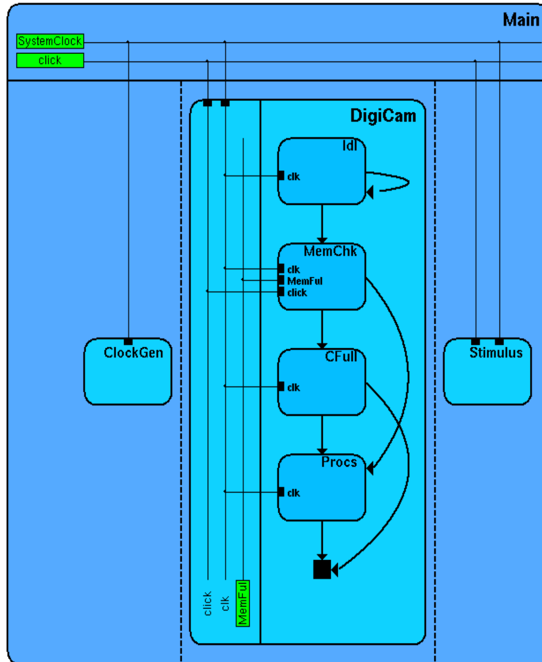


Figure 8. DigiCam behavior FSM

specification of pipelines. Figure 9 shows the pipeline execution of behaviors inside the behavior "Procs". In "Procs", there are six blocks of behaviors, which execute in pipelined order. Note the arrows joining these blocks are intersected by horizontal dashed lines. This is the standard notation for pipeline execution in SpecC. The blocks correspond to the six steps of processing:

1. *GetImg*: Get Image from the buffer, where image is stored by CCD.
2. *ZBA*: The block that does Zero Bias Adjustment of the image.
3. *DCT*: The block that does Discrete Cosine Transformation of image.
4. *QTZ*: Block that does Quantization of image.
5. *HFM*: Huffman Encoder block.
6. *SAVE*: Block that saves the compressed image in memory. Also can be seen, are five piped variables *Add1, Add2..Add5*. These are the variables, that hold addresses for the image at different stage of the pipeline.

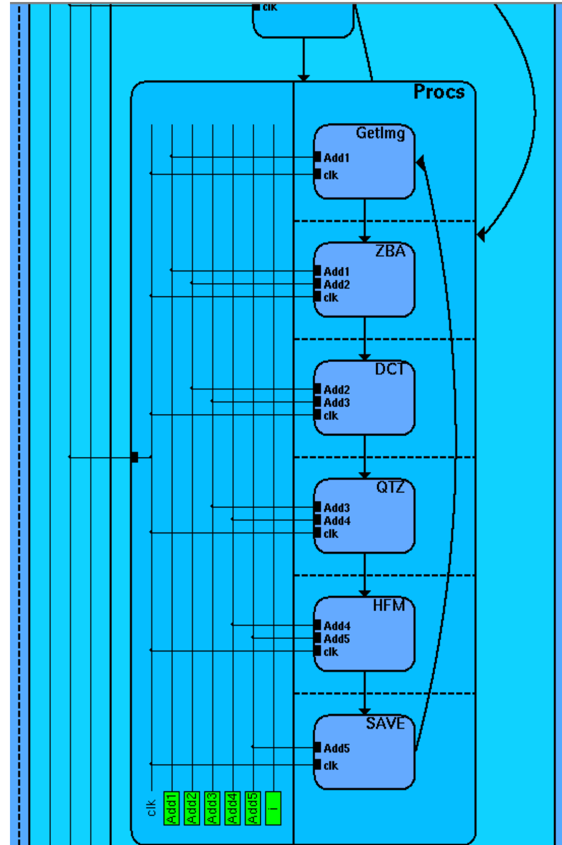


Figure 9. Procs behavior pipelined execution

Figure 10 shows the details of behavior *GetImg*. *GetImg* is responsible for initializing the CCD, capturing image with CCD and storing the image in buffer. It has three functions *CCDInit, CCDCap, CCDPop*, that correspond to the above three tasks, and are executed sequentially. *GetImg* pops up the image into memory and stores its location address in variable *Add1*.

Figure 11 shows the details of behavior *ZBA*. It has four instances *ZBR0_an_1, ZBR2_an_3, ZBR4_an_5, ZBR6_an_7*, which are executed in parallel. Each instance does Zero Bias Adjustment for two rows. These processes are independent of each other and there is no need to synchronize their parallel execution. Note that, there are no arrows between these blocks and only dashed lines separate them. This is the standard SOCE notation for parallelism. The block *ZBA* takes the image stored at address location *Add1*, does zero bias adjustment in parallel, and stores resultant image array at location *Add2*.

The block *DCT* takes the image stored at address location *Add2*, does Discrete Cosine Transformation, and stores resultant image array at location *Add3*. The block *QTZ* takes the image stored at address location *Add3*, put there by

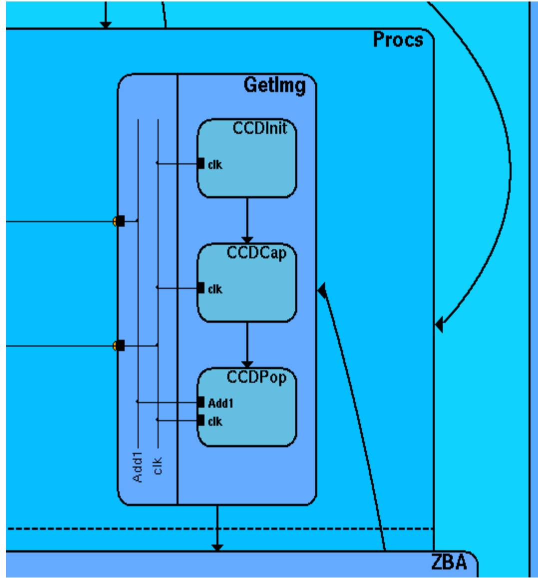


Figure 10. *GetImg* **behavior**

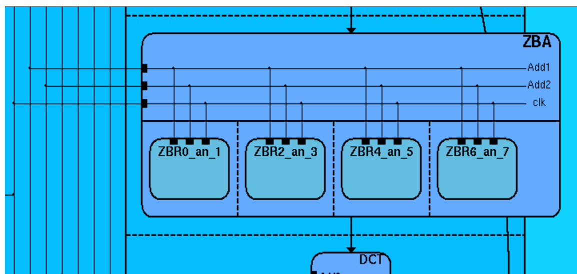


Figure 11. *ZBA* **behavior**

DCT, does Quantization, and stores the image array at location *Add4*. Similarly, the Huffman Encoding is performed in *HFM* block and stored at location *Add5*.

Figure 12 shows the details of behavior *SAVE*. It has two blocks, *Send2Mem*, which send the encoded image from address location *Add5* to the memory for storage, and block *MemGet*, which receives this data and performs the actual write operation on the memory. This block hides the complexity of circuits for writing data into memory. However, memory writing might be slower than the rate at which *Send2Mem* can send data. This is overcome by using a channel, *Chan1*, which encapsulates a handshake protocol. This synchronizes flow of data between the two blocks.

The complete hierarchy of the design, is shown in Figure 13. Let us observe the symbolic notations in the figure, to be able to appreciate the semantic depth of the environment. The "green leaves" indicate, a clean behavior with simple execution statements. Behaviors are defined hierarchically, each behavior can also contain a number of behavior instantiations of other behaviors. In clean leaf behavior,

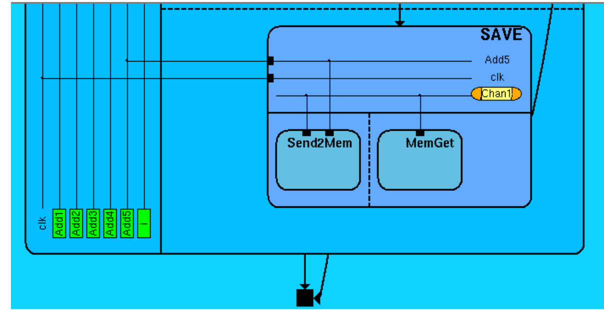


Figure 12. *SAVE* **behavior**

there are only sequence of statements, without any behavior instances [13]. The round symbol, next to DigiCam, represents a Finite State Machine. The two small dots like symbol, next to Procs, represents a behavior with pipeline execution. The two vertical squares, next to GetImg, represent sequentially executed behavior. The two vertical bars, next to ZBA, represent a behavior with parallel execution. The capsule shaped symbol, next to Chan1, indicates a channel for communication with a pre-defined protocol.

5. Profiling

The System on Chip Environment (SCE) can profile the system for us [14]. In this section, we present the profiling charts generated by SCE. Figure 14, shoes the profile for DigiCam. The profile shows code, computation, connectors and member variables. Observe that there are 611 code expressions, 89K computation operations, and 29 connections in DigiCam. Detailed profile of code of Digicam is shown in Figure 15. Detailed profile of computation of Digicam is shown in Figure 16. The charts are self-explanatory. Connections in Digicam are shown in Figure 17. Profiles for "Procs" and "Save" are shown in Figures 18 and 19.

6. C/C++ Based Design Methodologies

SystemC is a modeling platform consisting of C++ class libraries and a simulation kernel for designing at the system and register transfer level. Besides, providing a common high-level language, for modeling, analyzing and simulating an embedded system, it can be also linked to commercial tools such as Synopsys design compiler [15], for hardware synthesis. SpecC is a super-set of the C language. It is a complete language, not just a library, and it was specifically conceived for the specification and design of digital embedded systems. Detailed information on the syntax and semantics of SpecC and SystemC is available in ref. [9] and [8] respectively.

In this section, based on our experience with SystemC

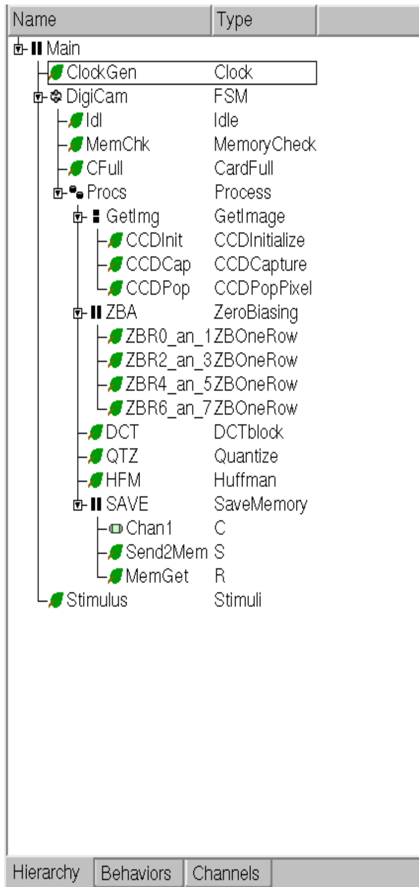


Figure 13. Hierarchy of Behaviors in System

Name	Type
Main	
ClockGen	Clock
DigiCam	FSM
Idl	Idle
MemChk	MemoryChe
CFull	CardFull
Procs	Process
Stimulus	Stimuli

Name	Type	N	Code [expressions]	Computation [operations]	Data [variables]	Heap [blocks]	Connections [acessors]	Traffic [transfers]
FSM		3	611	89212	66	0	29	2
click	inout bit[0:0]						1	2
clk	inout event						28	0
MemFul	bit[0:0]				1		1	1
CFull	CardFull	0	10	0	0	0	2	0
Idl	Idle	3	10	9	0	0	2	0
MemChk	MemoryCheck	3	18	15	0	0	4	2
Procs	Process	3	565	89182	65	0	22	0

Figure 14. Digicam Profile

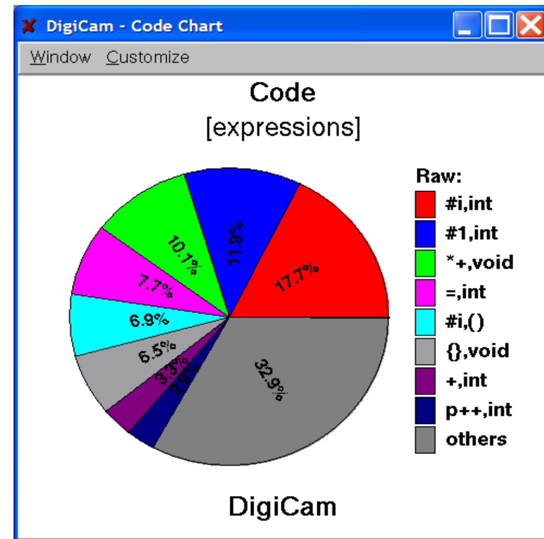


Figure 15. Digicam Code Profile

[4], we briefly compare C++ based methodology (SystemC) and C based methodology (SpecC) with respect to three aspects: 1) capability of modeling functionality, 2) capability of modeling the transfer of information between functional blocks, and 3) capability of modeling the execution sequence among functional blocks [13].

Both SpecC and SystemC support hierarchical modeling of system behavior. In SpecC the term behavior indicates a consolidate representation of both functionality and structure. Behavior and structure of the system are represented by a hierarchy of behaviors. A leaf behavior may contain hierarchical calls to functions but it does not contain any further sub instance. SystemC isolates functionality and structure into processes and modules respectively.

SpecC [6],[7] is more versatile and suited for system level deign. This is evident by the fact that, SpecC supports interrupts like *try* and *trap*. SystemC, it may be argued that, also has a feature of defining the sensitivity list for a function, which can simulate similar to interrupts. However, at system level design, designers use interrupts and the sensitivity list of SystemC is specified at the transaction level.

SpecC models data transfer among behaviors through the use of variables or channels. SystemC supports data transfer by connecting module ports through either signals or channels. A channel is a class that encapsulates communication. In SpecC a channel consists of a set of variables and functions (also called methods) which operate on the variables and define the communication protocol. Similarly, in SystemC, a channel consists of a set of signals and methods that operate on them. The difference in using variables and signals is that changes on variables are scheduled immediately, while changes on signals are queued and scheduled at the occurrence of the next event (i.e., the value of the signal is updated only after a delta delay). SystemC does not allow binding of variables to ports of modules, thus the use of variables for data transfer between processes in different modules is not permitted.

In SpecC the order of execution is by default sequential, however two mechanisms are provided to alter the execution sequence: 1) static scheduling and 2) dynamic scheduling. In static scheduling the sequence of execution is explicitly specified using dedicated constructs par (for parallel execution), pipe (for pipelined execution), fsm (for Finite State Machine execution). For dynamic scheduling, SpecC rely on the data type event and the wait and notify

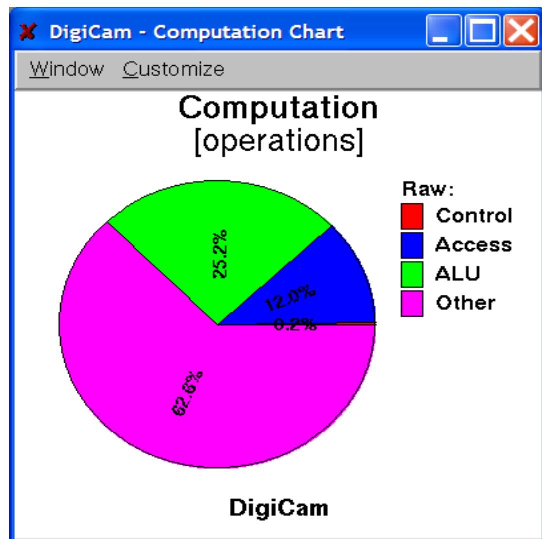


Figure 16. Digicam Computation Profile

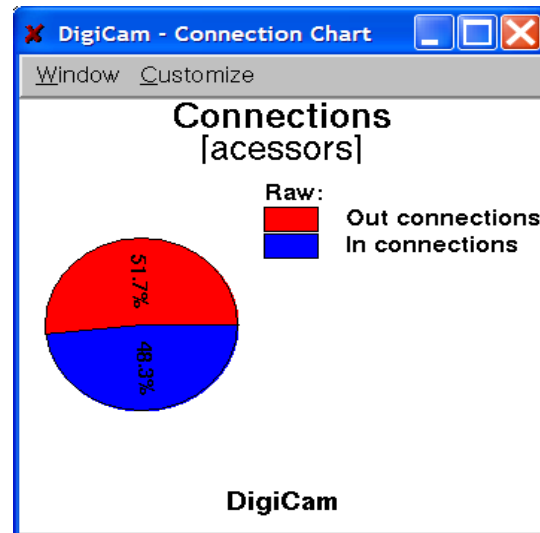


Figure 17. Digicam Connections

statements for synchronization between behaviors. SystemC supports only dynamic scheduling. In SpecC, static scheduling permits to precisely determine the execution sequence and as a consequence make architecture exploration much easier than with SystemC. In addition, since SystemC is a C++ library extension, the computation needs of the system under design are tightly coupled with the computational needs of the SystemC kernel, so the profiling of the model becomes prohibitive.

At the hardware level, in order to specify cycle-accurate finite state machines SpecC provides the construct fsm, while SystemC provides two mechanisms: 1) implicit modeling using the class SC_THREAD and wait statement, and 2) explicit modeling using the class SC_METHOD and switch statement.

At the software level, in order to obtain C code compilable and executable on the target microprocessor, users need to remove from the software written in SpecC all SpecC specific constructs (par, pipe, fsm, wait, etc.). The equivalent task in SystemC needs the additional step of converting the C++ code to C code.

7. Results and Conclusion

The effectiveness of the two methodologies has been evaluated with respect to many facets. SpecC and SystemC are comparable in terms of time required to complete the design, and time taken to execute the system model. Although both C and C++ design methodologies have many similarities, we observed that C based methodology is easier to use, and is better suited for architecture exploration. This results in smaller design time and better design quality, especially when there are many design alternatives to consider. On the other hand, currently, C++ methodology is better linked with commercial hardware synthesis tools.

During the implementation of this project, we were able to get a first hand experience of the design flow, shown in Figure 1. The first step was to come up with a design example, that offers scope for abstraction and manageable implementation complexity. Fortunately, the example of digital camera had enough space for including modules with almost all the models of computation. We consciously modeled the system to include parallel execution, sequential execution, pipelined execution, finite state machine execution, and communication using channels. This aided in understanding all the models of computation of SpecC. In order to avoid spending too much time in coding actual image processing algorithms, the behavior was substituted by a computation load of nearly same complexity.

The biggest academic and intellectual contribution of this project, System Design of Digital Camera Using SpecC, was to give the students an opportunity to learn a new SLDL. Since SpecC has the same syntax as C language, we were able to tackle design issues early on, instead of first trying to get used to a new syntax and grammar.

Though, some bugs in the tool prevented us from proceeding to the step of architecture refinement, we were able to simulate, profile, and even generate a C file for target Microprocessor from the specification model of the system. The fact that the tool is still under development justifies these errors. In future, new versions of the tool will be more versatile.

We also learnt that, synchronous communication between parallel blocks, is difficult to implement, without using an in-built communication protocol provided by SpecC.

SpecC should also include a debugging tool to help not only to debug the specification model, also for closer analysis of the system.

References

[1] K. Keutzer, S. Malik, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli, "System-Level Design: Orthogonalization of Concerns and Platform-Based Design," *IEEE Trans. CAD Integrated Circuits and Systems*, vol. 19, no. 12, 2000, pp. 1523-1543.

[2] C. Talarico, J.W. Rozenblit, A. Gupta, and E. Peter, "Performance Analysis of Embedded Systems with SystemC," *Proc. Int'l Conf. Computing, Communications and Control Technologies (CCCT 2004)*, IIIS Press, 2004, pp. 46-51.

[3] T. Givargis, F. Vahid, and J. Henkel, "System-Level Exploration for Pareto-Optimal Configurations in parameterized System-on a-Chip," *IEEE Trans. VLSI Systems*, vol. 10, no. 4, 2002, pp. 416-422.

[4] C. Talarico, A. Gupta, J.W. Rozenblit, and E. Peter, "Embedded System Engineering Using C/C++ Based Design Methodologies," *Proc. Int. Conf. on Engineering of Computer-Based Systems (IEEE-ECBS'05)*, April, 2005, accepted for publication.

[5] D.D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, 2000.

[6] T. Groetker, S. Liao, G. Martin, S. Swan:,*System Design with SystemC*, Kluwer Academic Publishers, 2002.

[7] A. Gerstlauer, R. Doemer, J. Peng, D.D. Gajski,*System Design: A Practical Guide with SpecC*, Kluwer Academic Publishers, 2001.

[8] Open SystemC Initiative, *Functional Specification for SystemC 2.0*, 2002, http://www.systemc.org

[9] SpecC Open Technology Consortium, *SpecC Language Reference Manual, version 2.0, 2004*, http://www.specc.org.

[10] F. Vahid, T. Givargis, *Embedded System Design, A Unified Hardware/Software Introduction*, John Wiley & Sons, New York, 2002.

[11] Joint Photographic Experts Group, http://www.jpeg.org/.

[12] C. Brown, B. Shepherd, *Graphics File Formats - Reference and Guide*, Manning Publication Company, Connecticut, 2002.

[13] L. Cai, D. Gajski, *C/C++ Based System Design Flow Using SpecC, VCC and SystemC*, CECS Technical Report 02-30, UC Irvine, CA, 2002.

[14] L. Cai, A. Gerstlauer, S. Abdi, J. Peng, D. Shin, H. Yu, R. Doemer, D. Gajski, *System-on-Chip Environment (SCE Version 2.2.0 Beta): Manual*, CECS Technical Report 03-45, UC Irvine, CA, 2003.

[15] Synopsys, *Design Compiler Reference Manual*, June 2003.

8. Appendix

A. Simulation Results

B. Code for Digital Camera Specification

C. Source file generated by SpecC for Motorola_68HC11

Appendix

A. Simulation Results

```
Time =    0 : Idle active...
 click  = 1 in stimuli
Time =  100 : CTick!
Time =  100 : MemoryCheck active...
Time =  200 : CTick!
Time =  200 : CCD Initialization ..
Time =  300 : CTick!
Time =  300 : CCD Capture ..
Time =  400 : CTick!
Time =  400 : CCD Popping Pixels ..
Time =  500 : CTick!
Time =  500 : Bias Adjustment  for two Rows
Time =  500 : Bias Adjustment  for two Rows
Time =  500 : Bias Adjustment  for two Rows
Time =  500 : Bias Adjustment  for two Rows
Time =  500 : DCT ...
Time =  600 : CTick!
Time =  600 : Quantize...
Time =  700 : CTick!
Time =  700 : Huffman Encoding...
Time =  800 : CTick!
Time =  800 : Saving to Memory...
Time =  900 : CTick!
Time = 1000 : CTick!
Time = 1100 : CTick!
Time = 1200 : CTick!
Time = 1300 : CTick!
Time = 1400 : CTick!
[aseemg@alpha ~]$
```

B. Code for Digital Camera Specification

```
//
// Digital.sc
// -------
//
// author:  Aseem Gupta
// last update: 25/11/2004

#include <sim.sh> #include <stdio.h>

// definition of states as child behaviors


interface IS {
    void Send(float);
};

interface IR {
    float Receive(void);
};

channel C() implements IS,IR {
    event Req;
    float Data;
    event Ack;

    void Send(float x)
    {
        Data = x;
        notify Req;
        wait Ack;
    }
    float Receive(void)
    {
        float y;
        wait Req;
        y = Data;
        notify Ack;
        return y;
    }
};


behavior S(IS Port, in bit[8] x) {
    //float x;
    void main(void)
    {   int i,z;
        z = (int)x;
        printf("Time =%5s : Saving to Memory...\n", time2str(now()));
        for(i=0;i<64;i++)
        {
            //printf(" SENDING x = %d \n",x);
            Port.Send(x);
            z = z + 1;
            // Right now, i am sending x as if x is data, bt I could send real data, ix x was address of data
        }
    }
};

behavior R(IR Port) {
    int y;
    void main(void)
    {
        int i;
        for(i=0;i<64;i++)
        {
            y = Port.Receive();
            //printf(" RECEIVING y = %d \n",y);
        }
    }
};

behavior CCDInitialize (event clk) {
    void main(void)
    {
    int i,a = 1;
    printf("Time =%5s : CCD Initialization ..\n", time2str(now()));
    // Assume that the way to initialize the 8 X 10 CCD is to send a bit to each one pixel, so execute a statement 80 times
    for (i=0;i<82;i++)
```

```
    {
        a = a + a;
    }
    wait(clk);
    }
}; behavior CCDCapture (event clk) {
    void main(void)
    {
    int i,a = 1;
    printf("Time =%5s : CCD Capture ..\n", time2str(now()));
    // Assume that the way to capture the 8 X 10 CCD is to send a bit to each one pixel, so execute a statement 80 times
    for (i=0;i<(80+8);i++)
    {
        a = a + a;
    }
    wait(clk);
    }
};

behavior CCDPopPixel (out bit[8] Add1, event clk) {
    void main(void)
    {
    int i,a = 1;
    printf("Time =%5s : CCD Popping Pixels ..\n", time2str(now()));
    // Assume that the way to pop the 8 X 10 CCD is to read a Byte from, so execute a statement 640 times
    for (i=0;i<(640+64);i++)
    {
        a = a + 2;
    }
    Add1 = 10101010b;
    wait(clk);
    }
};

behavior GetImage (out bit[8] Add1, event clk) {
    CCDInitialize CCDInit(clk);
    CCDCapture CCDCap(clk);
    CCDPopPixel CCDPop(Add1, clk);
    void main(void)
    {
    CCDInit.main();
    CCDCap.main();
    CCDPop.main();
    }
};


behavior ZBOneRow(
        in bit[8] Row,
        out bit[8] Add2,
        event clk)
{
    void main(void)
    {
    int a,i;
    printf("Time =%5s : Bias Adjustment  for two Rows \n", time2str(now()));
    // It is for one row, so first compute Bias
    a = 2;
    a = a * 2;
    a = a / 2;
    // in 8 operations & then do biasing in other elements
    for (i = 0;i<10;i++)
    {
        a = a+2;
    }
    }
};

behavior ZeroBiasing (in bit[8] Add1, out bit[8] Add2, event clk) {
    ZBOneRow ZBR0_an_1(Add1, Add2, clk);
    ZBOneRow ZBR2_an_3(Add1, Add2,clk);
    ZBOneRow ZBR4_an_5(Add1, Add2,clk);
    ZBOneRow ZBR6_an_7(Add1, Add2,clk);
    void main(void)
    {
    par
    {
        ZBR0_an_1.main();
        ZBR2_an_3.main();
```

```
        ZBR4_an_5.main();
        ZBR6_an_7.main();
    }
    }
    // Takes in data stored in Add1
    // Does Zero Biasing
    // Puts data at address Add2
};


behavior Quantize (in bit[8] Add3, out bit[8] Add4,event clk) {
    void main(void)
    {
    int a,i;
    printf("Time =%5s : Quantize...\n", time2str(now()));
    //includes a division step, one for each pixel, so 64 divisions
    // 2 loops one inside another have more overhead than just one loop, because of comparisons etc.
    for (i=0;i<(64+6);i++)
    {
        a = a / 2;
        a = a * 2;
    }
    wait(clk);
    // Takes in data stored in Add2
    // Does Quantization
    // Puts data at address Add3
    }
};

behavior DCTblock (in bit[8] Add2, out bit[8] Add3,event clk) {
    void main(void)
    {
    int i,a=2;
    printf("Time =%5s : DCT ...\n", time2str(now()));
    // DCT is just a lookup out of 64 values. The number to lookup is generated by multilplication and rounding off
    // Assume it takes 1 operation to multiply, 3 operations to round off, and log_2(64) = 6 operations to lookup
    //Total 10 operations, add overhead of 2, so total 12 operations per pixel, so total 64 X 12 = 786
    for (i=0;i<(786+78);i++)
    {
        a = a + 2;
    }
    wait(clk);
    // Takes in data stored in Add3
    // Does DCT
    // Puts data at address Add4
    }
};

behavior Huffman (in bit[8] Add4, out bit[8] Add5,event clk) {
    void main(void)
    {
    int a,i;
    printf("Time =%5s : Huffman Encoding...\n", time2str(now()));
    // Complexity of Huffman is O(nlgn) = 64(lg 64) = 384
    // The complexity hides all constants etc. lets say they were 5, hence 384*5 = 1920
    // lets make it computationally expensive
    for (i=0;i<(1920+192);i++)
    {
    for (i=0;i<(1920+192);i++)
    {
    for (i=0;i<(1920+192);i++)
    {
    for (i=0;i<(1920+192);i++)
    {
    for (i=0;i<(1920+192);i++)
    {
    for (i=0;i<(1920+192);i++)
    {
            a = a + 2;
            a = a - 2;
            a = a * 2;
            a = a / 2;

    } } }  } } }
    wait(clk);
    // Takes in data stored in Add4
    // Does Huffman encoding
    // Puts data at address Add5
    }
```

```
};


behavior SaveMemory (in bit[8] Add5,event clk) {
    C Chan1;
    S Send2Mem(Chan1,Add5);
    R MemGet(Chan1);
    void main(void)
    {
    //printf("Time =%5s : Saving to Memory...\n", time2str(now()));
    //  Assume each pixel takes up 1 byte or 8 bits, so total 512 bits
    // Let us use a channel to send send data to memory
    // Why?, because memories are slow and processors are fast.
    // Lets send 4 sample values 9,10,11,12
    par
    {
        Send2Mem.main();
        MemGet.main();
    }
    //wait(clk);
    }
};


behavior Process (
    event clk)
{
    int i;
    piped bit[8] Add1;
    piped bit[8] Add2;
    piped bit[8] Add3;
    piped bit[8] Add4;
    piped bit[8] Add5;
    GetImage GetImg(Add1, clk);
    ZeroBiasing ZBA(Add1, Add2, clk);
    Quantize QTZ(Add3, Add4,clk);
    DCTblock DCT(Add2, Add3,clk);
    Huffman HFM(Add4, Add5,clk);
    SaveMemory SAVE(Add5,clk);
    void main(void)
    {
    pipe(i=0;i<1;i++)
    {
        GetImg.main();
        ZBA.main();
        DCT.main();
        QTZ.main();
        HFM.main();
            SAVE.main();
    }
    }
};

behavior CardFull (event clk) {
    void main(void)
    {
        printf("Time =%5s : Change Memory Card!! \n", time2str(now()));
        wait(clk);
    }
};

behavior MemoryCheck (
    event clk,
    out bit[1] MemFul,
    out bit[1] click
    )
{
    void main(void)
    {
    // Here, you undo click, so that FSM does not fire up again
    click = 0;
    //let us assume that 1 image takes 1000KB
    printf("Time =%5s : MemoryCheck active...\n", time2str(now()));
    // Assume that for now, space is available
    // if (MemoryStaus > 1000)
    MemFul = 0;
    wait(clk);
    }
};
```

```
behavior Idle (event clk) {
    void main(void)
    {
        printf("Time =%5s : Idle active...\n", time2str(now()));
        wait(clk);
    }
};

behavior FSM (
    inout bit[1]    click,
    event clk)
{
    bit[1]  MemFul;
    Idle    Idl(clk);
    CardFull    CFull(clk);
    MemoryCheck MemChk(clk,MemFul,click);
    Process Procs(clk);

    void main(void)
    {
    fsm
    {
        Idl:
            {
                if (click)
                goto MemChk;
                if(!click)
                    goto Idl;
            }
            MemChk:
            {
                if (MemFul)
                goto CFull;
            if (!MemFul)
                goto Procs;
            }
            CFull:
            {
                break;
            }
            Procs:
        {
            break;
        }
    }
    }
};

behavior Stimuli(
    inout bit[1] click,
    event clk)
{
    void main(void)
    {
        printf(" click  = 1 in stimuli \n");
        click = 1;
        wait(clk);
        //printf(" click  = 0 in stimuli \n");
        //click = 0;
    }
};

// definition of the clock generator

behavior Clock (event clk) {
    void main(void)
    {
    int i;
    for(i=1; i<15; i++) // the demo shouldn't run forever
    {
        waitfor(100);
        printf("Time =%5s : CTick!\n", time2str(now()));
        notify(clk);
    }
    }
};
```

```
// the testbench

behavior Main(void) {
    bit[1]  click;
    event   SystemClock;
    Clock   ClockGen(SystemClock);
    FSM DigiCam(click,SystemClock);
    Stimuli Stimulus(click,SystemClock);

    int main(void)
    {

    par
    {
        ClockGen.main();
        DigiCam.main();
        Stimulus.main();
    }
    }
};
```

C. C-Source file Generated by SpecC for Motorola_68HC11

```
/*
   C source file generated by SpecC V2.2.b
   Design: Digital
   Root  : Motorola_68HC11
   File:   Motorola_68HC11.c
   Time:   Sun Dec 22 23:44:48 1935

*/

#include "/home/aseemg/Motorola_68HC11.h"

/* Global Variables */

/* Global Function Definitions */

/* Global Functions */

struct Idle {
    event (*clk) /* port(data) */;

};

struct CCDInitialize {
    bit<_BITLEN(0,0),false> (*CCDInitDone) /* port(data) */;
    event (*clk) /* port(data) */;

};

struct CCDCapture {
    bit<_BITLEN(0,0),false> (*CCDCapDone) /* port(data) */;
    event (*clk) /* port(data) */;

};

struct CCDPopPixel {
    bit<_BITLEN(0,0),false> (*CCDPopDone) /* port(data) */;
    event (*clk) /* port(data) */;

};

struct Default {

 int a; /* just not to compiler complain */
};

struct GetImage {
    event (*clk) /* port(data) */;
    bit[0:0] CCDCapDone;
    bit[0:0] CCDInitDone;
    bit[0:0] CCDPopDone;

    struct  CCDCapture CCDCap;
    struct  CCDInitialize CCDInit;
    struct  CCDPopPixel CCDPop;
    struct  Default This;
};

struct ZBInitial {
    bit<_BITLEN(7,0),false> (*Row) /* port(data) */;
    bit<_BITLEN(7,0),false> (*Col) /* port(data) */;
    event (*clk) /* port(data) */;

};

struct ZBCompBias {
    bit<_BITLEN(7,0),false> (*Row) /* port(data) */;
    bit<_BITLEN(7,0),false> (*Col) /* port(data) */;
    bit<_BITLEN(7,0),false> (*Bias) /* port(data) */;
    event (*clk) /* port(data) */;

};

struct ZBNextRow {
    bit<_BITLEN(7,0),false> (*Row) /* port(data) */;
    event (*clk) /* port(data) */;

};
```

```
struct BiasAdjustment {
    bit<_BITLEN(7,0),false> (*Row) /* port(data) */;
    bit<_BITLEN(7,0),false> (*Col) /* port(data) */;
    event (*clk) /* port(data) */;

};

struct ZeroBiasing {
    event (*clk) /* port(data) */;
    bit[7:0] Bias;
    bit[7:0] Col;
    bit[7:0] Row;

    struct  BiasAdjustment BiasAdj;
    struct  ZBCompBias CompBias;
    struct  ZBInitial ZBInit;
    struct  ZBNextRow ZBNxtRow;
};

struct DCTblock {
    event (*clk) /* port(data) */;

};

struct Quantize {
    event (*clk) /* port(data) */;

};

struct Huffman {
    event (*clk) /* port(data) */;

};

struct SaveMemory {
    event (*clk) /* port(data) */;

};

struct Stage1 {
    int (*p1) /* port(data) */;
    int (*p2) /* port(data) */;
    int (*p3) /* port(data) */;
    int Stage1ExecCount;

};

struct Stage2 {
    int (*p1) /* port(data) */;
    int (*p2) /* port(data) */;
    int Stage2ExecCount;

};

struct Stage3 {
    int (*p1) /* port(data) */;
    int (*p2) /* port(data) */;
    int (*p3) /* port(data) */;
    int Stage3ExecCount;

};

struct CardFull {
    event (*clk) /* port(data) */;

};

struct Clock_PE1 {

 int a; /* just not to compiler complain */
};

struct END_OF_FSM {

 int a; /* just not to compiler complain */
};

struct MemoryCheck {
    event (*clk) /* port(data) */;
```

```
    bit<_BITLEN(0,0),false> (*MemFul) /* port(data) */;
    bit<_BITLEN(0,0),false> (*click) /* port(data) */;

};

struct Process {
    int (*a) /* port(data) */;
    int (*b) /* port(data) */;
    event (*clk) /* port(data) */;
    int From1To2;
    int From1To3;
    int From2To3;
    int i;

    struct  DCTblock DCT;
    struct  GetImage GetImg;
    struct  Huffman HFM;
    struct  Quantize QTZ;
    struct  SaveMemory SAVE;
    struct  Default This;
    struct  ZeroBiasing ZBA;
    struct  Stage1 s1;
    struct  Stage2 s2;
    struct  Stage3 s3;
};

struct FSM {
    bit<_BITLEN(0,0),false> (*click) /* port(data) */;
    event (*clk) /* port(data) */;
    bit[0:0] MemFul;
    bit[0:0] ProcDone;
    int i;
    int o;

    struct  CardFull CFull;
    struct  Idle Idl;
    struct  MemoryCheck MemChk;
    struct  Process Procs;
    struct  END_OF_FSM ar_end;
};

struct Stimuli {
    bit<_BITLEN(0,0),false> (*click) /* port(data) */;
    event (*clk) /* port(data) */;

};

struct C_Motorola_68HC11 {
    event (*SystemClock) /* port(data) */;
    bit[0:0] click;

    struct  Clock_PE1 ClockGen;
    struct  FSM DigiCam;
    struct  Stimuli Stim;
};


void Idle_main(struct Idle *This) {
    printf("Time =%5s : Idle active...\n", time2str(now()));
    WAIT(clk);
}

void CCDInitialize_main(struct CCDInitialize *This) {
    int a = 1;
    int i;

    printf("Time =%5s : CCD Initialization ..\n", time2str(now()));
    for(i = 0; i < 82; i++ )
    {
    a = a + a;
    }
    (*(This->CCDInitDone)) = 1;
    WAIT(clk);
}

void CCDCapture_main(struct CCDCapture *This) {
    int a = 1;
    int i;
```

```
    printf("Time =%5s : CCD Capture ..\n", time2str(now()));
    for(i = 0; i < (80 + 8); i++ )
    {
    a = a + a;
    }
    (*(This->CCDCapDone)) = 1;
    WAIT(clk);
}

void CCDPopPixel_main(struct CCDPopPixel *This) {
    int a = 1;
    int i;

    printf("Time =%5s : CCD Popping Pixels ..\n", time2str(now()));
    for(i = 0; i < (640 + 64); i++ )
    {
    a = a + 2;
    }
    (*(This->CCDPopDone)) = 1;
    WAIT(clk);
}

void Default_main(struct Default *This) { }

void GetImage_main(struct GetImage *This) {
    CCDInitialize_main(&(This->CCDInit));
    CCDCapture_main(&(This->CCDCap));
    CCDPopPixel_main(&(This->CCDPop));
}

void ZBInitial_main(struct ZBInitial *This) {
    printf("Time =%5s : ZBInitialize...\n", time2str(now()));
    (*(This->Row)) = 1;
    (*(This->Col)) = 1;
    WAIT(clk);
}

void ZBCompBias_main(struct ZBCompBias *This) {
    printf("Time =%5s : ZBCompute Bias for Row %d    .\n", time2str(now()),
    (int)(*(This->Row)));
    (*(This->Bias)) = 2;
    (*(This->Bias)) = 3;
    (*(This->Bias)) = 4;
    (*(This->Bias)) = 5;
    (*(This->Col)) = 1;
    WAIT(clk);
}

void ZBNextRow_main(struct ZBNextRow *This) {
    (*(This->Row)) = (*(This->Row)) + 1;
    printf("Time =%5s : ZB Next Row  %d   ..\n", time2str(now()), (int)(*(This->Row)));
    WAIT(clk);
}

void BiasAdjustment_main(struct BiasAdjustment *This) {
    int a;

    printf("Time =%5s : Bias Adjustment  .for Row %d    .\n", time2str(now()),
    (int)(*(This->Row)));
    a = a + 2;
    (*(This->Col)) = (*(This->Col)) + 1;
    WAIT(clk);
}

void ZeroBiasing_main(struct ZeroBiasing *This) {
    int i;

    ZBInitial_main(&(This->ZBInit));
    for(i = 0; i < 8; i++ )
    {
    PAR
    {
        ZBCompBias_main(&(This->CompBias));
        BiasAdjustment_main(&(This->BiasAdj));
        ZBNextRow_main(&(This->ZBNxtRow));
    }
    }
}
```

```
void DCTblock_main(struct DCTblock *This) {
    int a = 2;
    int i;

    printf("Time =%5s : DCT ...\n", time2str(now()));
    for(i = 0; i < (786 + 78); i++ )
    {
    a = a + 2;
    }
    WAIT(clk);
}

void Quantize_main(struct Quantize *This) {
    int a;
    int i;

    printf("Time =%5s : Quantize...\n", time2str(now()));
    for(i = 0; i < (64 + 6); i++ )
    {
    a = a / 2;
    a = a * 2;
    }
    WAIT(clk);
}

void Huffman_main(struct Huffman *This) {
    int a;
    int i;

    printf("Time =%5s : Huffman Encoding...\n", time2str(now()));
    for(i = 0; i < (1920 + 192); i++ )
    {
    for(i = 0; i < (1920 + 192); i++ )
    {
        for(i = 0; i < (1920 + 192); i++ )
        {
        for(i = 0; i < (1920 + 192); i++ )
        {
            for(i = 0; i < (1920 + 192); i++ )
            {
            for(i = 0; i < (1920 + 192); i++ )
            {
                a = a / 2;
                a = a * 2;
            }
            }
        }
        }
    }
    }
    WAIT(clk);
}

void SaveMemory_main(struct SaveMemory *This) {
    int a;
    int i;

    printf("Time =%5s : Saving to Memory...\n", time2str(now()));
    for(i = 0; i < (512); i++ )
    {
    a = a / 2;
    a = a * 2;
    }
    WAIT(clk);
}

void Stage1_main(struct Stage1 *This) {
    int t1;
    int t2;

    (This->Stage1ExecCount)++ ;
    printf("Stage1 execution #%d at time %s\n", (This->Stage1ExecCount),
    time2str(now()));
    printf("Stage1 input:  p3 = %d\n", (*(This->p3)));
    t1 = (*(This->p3)) + 1000 + 10000 * (This->Stage1ExecCount);
    t2 = (*(This->p3)) + 2000 + 10000 * (This->Stage1ExecCount);
    printf("Stage1 output: p1 = %d, p2 = %d\n", t1, t2);
    WAITFOR((5));
    (*(This->p1)) = t1;
```

```
    (*(This->p2)) = t2;
}

void Stage2_main(struct Stage2 *This) {
    int t;

    (This->Stage2ExecCount)++ ;
    printf("Stage2 execution #%d at time %s\n", (This->Stage2ExecCount),
    time2str(now()));
    printf("Stage2 input:  p1 = %d\n", (*(This->p1)));
    t = (*(This->p1)) + 100000 * (This->Stage2ExecCount);
    printf("Stage2 output: p2 = %d\n", t);
    WAITFOR((10));
    (*(This->p2)) = t;
}

void Stage3_main(struct Stage3 *This) {
    int t;

    (This->Stage3ExecCount)++ ;
    printf("Stage3 execution #%d at time %s\n", (This->Stage3ExecCount),
    time2str(now()));
    printf("Stage3 input:  p1 = %d, p2 = %d\n", (*(This->p1)), (*(This->p2)));
    t = (*(This->p1)) + (*(This->p2));
    printf("Stage3 output: p3 = %d\n", t);
    WAITFOR((7));
    (*(This->p3)) = t;
}

void CardFull_main(struct CardFull *This) {
    printf("Time =%5s : Change Memory Card!! \n", time2str(now()));
    WAIT(clk);
}

void Clock_PE1_main(struct Clock_PE1 *This) { }

void END_OF_FSM_main(struct END_OF_FSM *This) { }

void MemoryCheck_main(struct MemoryCheck *This) {
    (*(This->click)) = 0;
    printf("Time =%5s : MemoryCheck active...\n", time2str(now()));
    (*(This->MemFul)) = 0;
    WAIT(clk);
}

void Process_main(struct Process *This) {
    pipe((This->i) = 0; (This->i) < 1; (This->i)++ )
    {
    GetImage_main(&(This->GetImg));
    ZeroBiasing_main(&(This->ZBA));
    DCTblock_main(&(This->DCT));
    Quantize_main(&(This->QTZ));
    Huffman_main(&(This->HFM));
    SaveMemory_main(&(This->SAVE));
    }
}

void FSM_main(struct FSM *This) {
    FSM {
    goto Idl;
    Idl:
        Idle_main(&(This->Idl));
        if ((*(This->click)))       goto MemChk;
        if ( !click) goto Idl;
        goto MemChk;
    MemChk:
        MemoryCheck_main(&(This->MemChk));
        if ((This->MemFul))         goto CFull;
        if ( !MemFul) goto Procs;
        goto CFull;
    CFull:
        CardFull_main(&(This->CFull));
        goto ar_end;
    Procs:
        Process_main(&(This->Procs));
        if ((This->ProcDone))       goto ar_end;
        goto ar_end;
    ar_end:
        END_OF_FSM_main(&(This->ar_end));
```

```
        break;
    }
}

void Stimuli_main(struct Stimuli *This) {
    printf(" click  = 1 in stimuli \n");
    (*(This->click)) = 1;
    WAIT(clk);
}

int Motorola_68HC11_main(struct C_Motorola_68HC11 *This) {
    PAR
    {
    Clock_PE1_main(&(This->ClockGen));
    FSM_main(&(This->DigiCam));
    Stimuli_main(&(This->Stim));
    }
}




/* port of the software ROOT Behavior */ bool _scc_true  = 1; bool
_scc_false = 0;

#ifdef __SPECC__ /* SpecC structure initilization,for simulation
only*/ /* BIG structure initilization generated from behavior
instance tree */ struct C_Motorola_68HC11  c_PE1= {0/*click*/,
{0}/*ClockGen*/, {0/* port click*/,0/* port
SystemClock*/,0/*MemFul*/,0/*ProcDone*/,0/*i*/,0/*o*/, {0/* port
clk*/}/*CFull*/, {0/* port clk*/}/*Idl*/, {0/* port clk*/,0/* port
MemFul*/,0/* port click*/}/*MemChk*/, {0/* port i*/,0/* port o*/,0/*
port clk*/,0/*From1To2*/,0/*From1To3*/,0/*From2To3*/,0/*i*/, {0/*
port clk*/}/*DCT*/, {0/* port
clk*/,0/*CCDCapDone*/,0/*CCDInitDone*/,0/*CCDPopDone*/, {0/* port
CCDCapDone*/,0/* port clk*/}/*CCDCap*/, {0/* port CCDInitDone*/,0/*
port clk*/}/*CCDInit*/, {0/* port CCDPopDone*/,0/* port
clk*/}/*CCDPop*/, {0}/*This*/}/*GetImg*/, {0/* port clk*/}/*HFM*/,
{0/* port clk*/}/*QTZ*/, {0/* port clk*/}/*SAVE*/, {0}/*This*/, {0/*
port clk*/,0/*Bias*/,0/*Col*/,0/*Row*/, {0/* port Row*/,0/* port
Col*/,0/* port clk*/}/*BiasAdj*/, {0/* port Row*/,0/* port Col*/,0/*
port Bias*/,0/* port clk*/}/*CompBias*/, {0/* port Row*/,0/* port
Col*/,0/* port clk*/}/*ZBInit*/, {0/* port Row*/,0/* port
clk*/}/*ZBNxtRow*/}/*ZBA*/, {0/* port From1To2*/,0/* port
From1To3*/,0/* port a*/,0/*Stage1ExecCount*/}/*s1*/, {0/* port
From1To2*/,0/* port From2To3*/,0/*Stage2ExecCount*/}/*s2*/, {0/*
port From1To3*/,0/* port From2To3*/,0/* port
b*/,0/*Stage3ExecCount*/}/*s3*/}/*Procs*/,
{0}/*ar_end*/}/*DigiCam*/, {0/* port click*/,0/* port
SystemClock*/}/*Stim*/};

void _scc_port_mapping(void) { c_PE1.DigiCam.click=&(c_PE1.click);
c_PE1.DigiCam.clk=&(c_PE1.SystemClock);
c_PE1.DigiCam.CFull.clk=&(c_PE1.SystemClock);
c_PE1.DigiCam.Idl.clk=&(c_PE1.SystemClock);
c_PE1.DigiCam.MemChk.clk=&(c_PE1.SystemClock);
c_PE1.DigiCam.MemChk.MemFul=&(c_PE1.DigiCam.MemFul);
c_PE1.DigiCam.MemChk.click=&(c_PE1.click);
c_PE1.DigiCam.Procs.a=&(c_PE1.DigiCam.i);
c_PE1.DigiCam.Procs.b=&(c_PE1.DigiCam.o);
c_PE1.DigiCam.Procs.clk=&(c_PE1.SystemClock);
c_PE1.DigiCam.Procs.DCT.clk=&(c_PE1.SystemClock);
c_PE1.DigiCam.Procs.GetImg.clk=&(c_PE1.SystemClock);
c_PE1.DigiCam.Procs.GetImg.CCDCap.CCDCapDone=&(c_PE1.DigiCam.Procs.GetImg.CCDCapDone);
c_PE1.DigiCam.Procs.GetImg.CCDCap.clk=&(c_PE1.SystemClock);
c_PE1.DigiCam.Procs.GetImg.CCDInit.CCDInitDone=&(c_PE1.DigiCam.Procs.GetImg.CCDInitDone);
c_PE1.DigiCam.Procs.GetImg.CCDInit.clk=&(c_PE1.SystemClock);
c_PE1.DigiCam.Procs.GetImg.CCDPop.CCDPopDone=&(c_PE1.DigiCam.Procs.GetImg.CCDPopDone);
c_PE1.DigiCam.Procs.GetImg.CCDPop.clk=&(c_PE1.SystemClock);
c_PE1.DigiCam.Procs.HFM.clk=&(c_PE1.SystemClock);
c_PE1.DigiCam.Procs.QTZ.clk=&(c_PE1.SystemClock);
c_PE1.DigiCam.Procs.SAVE.clk=&(c_PE1.SystemClock);
c_PE1.DigiCam.Procs.ZBA.clk=&(c_PE1.SystemClock);
c_PE1.DigiCam.Procs.ZBA.BiasAdj.Row=&(c_PE1.DigiCam.Procs.ZBA.Row);
c_PE1.DigiCam.Procs.ZBA.BiasAdj.Col=&(c_PE1.DigiCam.Procs.ZBA.Col);
c_PE1.DigiCam.Procs.ZBA.BiasAdj.clk=&(c_PE1.SystemClock);
c_PE1.DigiCam.Procs.ZBA.CompBias.Row=&(c_PE1.DigiCam.Procs.ZBA.Row);
c_PE1.DigiCam.Procs.ZBA.CompBias.Col=&(c_PE1.DigiCam.Procs.ZBA.Col);
c_PE1.DigiCam.Procs.ZBA.CompBias.Bias=&(c_PE1.DigiCam.Procs.ZBA.Bias);
```

```
c_PE1.DigiCam.Procs.ZBA.CompBias.clk=&(c_PE1.SystemClock);
c_PE1.DigiCam.Procs.ZBA.ZBInit.Row=&(c_PE1.DigiCam.Procs.ZBA.Row);
c_PE1.DigiCam.Procs.ZBA.ZBInit.Col=&(c_PE1.DigiCam.Procs.ZBA.Col);
c_PE1.DigiCam.Procs.ZBA.ZBInit.clk=&(c_PE1.SystemClock);
c_PE1.DigiCam.Procs.ZBA.ZBNxtRow.Row=&(c_PE1.DigiCam.Procs.ZBA.Row);
c_PE1.DigiCam.Procs.ZBA.ZBNxtRow.clk=&(c_PE1.SystemClock);
c_PE1.DigiCam.Procs.s1.p1=&(c_PE1.DigiCam.Procs.From1To2);
c_PE1.DigiCam.Procs.s1.p2=&(c_PE1.DigiCam.Procs.From1To3);
c_PE1.DigiCam.Procs.s1.p3=&(c_PE1.DigiCam.i);
c_PE1.DigiCam.Procs.s2.p1=&(c_PE1.DigiCam.Procs.From1To2);
c_PE1.DigiCam.Procs.s2.p2=&(c_PE1.DigiCam.Procs.From2To3);
c_PE1.DigiCam.Procs.s3.p1=&(c_PE1.DigiCam.Procs.From1To3);
c_PE1.DigiCam.Procs.s3.p2=&(c_PE1.DigiCam.Procs.From2To3);
c_PE1.DigiCam.Procs.s3.p3=&(c_PE1.DigiCam.o);
c_PE1.Stim.click=&(c_PE1.click);
c_PE1.Stim.clk=&(c_PE1.SystemClock); }

#else /* BIG structure initilization generated from behavior
instance tree */ struct C_Motorola_68HC11  c_PE1= {0/*click*/,
{0}/*ClockGen*/,
{&(c_PE1.click),&(c_PE1.SystemClock),0/*MemFul*/,0/*ProcDone*/,0/*i*/,0/*o*/,
{&(c_PE1.SystemClock)}/*CFull*/, {&(c_PE1.SystemClock)}/*Idl*/,
{&(c_PE1.SystemClock),&(c_PE1.DigiCam.MemFul),&(c_PE1.click)}/*MemChk*/,
{&(c_PE1.DigiCam.i),&(c_PE1.DigiCam.o),&(c_PE1.SystemClock),0/*From1To2*/,0/*From1To3*/,0/*From2To3*/,0/*i*/,
{&(c_PE1.SystemClock)}/*DCT*/,
{&(c_PE1.SystemClock),0/*CCDCapDone*/,0/*CCDInitDone*/,0/*CCDPopDone*/,
{&(c_PE1.DigiCam.Procs.GetImg.CCDCapDone),&(c_PE1.SystemClock)}/*CCDCap*/,
{&(c_PE1.DigiCam.Procs.GetImg.CCDInitDone),&(c_PE1.SystemClock)}/*CCDInit*/,
{&(c_PE1.DigiCam.Procs.GetImg.CCDPopDone),&(c_PE1.SystemClock)}/*CCDPop*/,
{0}/*This*/}/*GetImg*/, {&(c_PE1.SystemClock)}/*HFM*/,
{&(c_PE1.SystemClock)}/*QTZ*/, {&(c_PE1.SystemClock)}/*SAVE*/,
{0}/*This*/, {&(c_PE1.SystemClock),0/*Bias*/,0/*Col*/,0/*Row*/,
{&(c_PE1.DigiCam.Procs.ZBA.Row),&(c_PE1.DigiCam.Procs.ZBA.Col),&(c_PE1.SystemClock)}/*BiasAdj*/,
{&(c_PE1.DigiCam.Procs.ZBA.Row),&(c_PE1.DigiCam.Procs.ZBA.Col),&(c_PE1.DigiCam.Procs.ZBA.Bias),&(c_PE1.SystemClock)}/*CompBias*/,
{&(c_PE1.DigiCam.Procs.ZBA.Row),&(c_PE1.DigiCam.Procs.ZBA.Col),&(c_PE1.SystemClock)}/*ZBInit*/,
{&(c_PE1.DigiCam.Procs.ZBA.Row),&(c_PE1.SystemClock)}/*ZBNxtRow*/}/*ZBA*/,
{&(c_PE1.DigiCam.Procs.From1To2),&(c_PE1.DigiCam.Procs.From1To3),&(c_PE1.DigiCam.i),0/*Stage1ExecCount*/}/*s1*/,
{&(c_PE1.DigiCam.Procs.From1To2),&(c_PE1.DigiCam.Procs.From2To3),0/*Stage2ExecCount*/}/*s2*/,
{&(c_PE1.DigiCam.Procs.From1To3),&(c_PE1.DigiCam.Procs.From2To3),&(c_PE1.DigiCam.o),0/*Stage3ExecCount*/}/*s3*/}/*Procs*/,
{0}/*ar_end*/}/*DigiCam*/,
{&(c_PE1.click),&(c_PE1.SystemClock)}/*Stim*/};

#endif



#ifdef __SPECC__ int main(void) {
    _scc_port_mapping();
    Motorola_68HC11_main(&c_PE1);
} #else int main() {
    Motorola_68HC11_main(&c_PE1);
    return  1;
} #endif

/* End of file Motorola_68HC11.c*/
```