UC Irvine
ICS Technical Reports

Title
System level architecture exploration using the SpecC methodology

Permalink
https://escholarship.org/uc/item/0b98k3zv

Authors
Cai, Lukai
Olivarez, Mike
Gajski, Daniel D.

Publication Date
2000-09-07

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

ICS

TECHNICAL REPORT

System Level Architecture Exploration Using the SpecC Methodology

Technical Report ICS-00-36
Sept 07, 2000

Lukai Cai
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA
lcai@ics.uci.edu

Mike Olivarez
Architecture and System Platforms, Motorola
M.Olivarez@Motorola.com

Dr. Daniel D. Gajski
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA
gajski@ics.uci.edu

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Information and Computer Science
University of California, Irvine

System Level Architecture Exploration Using the SpecC Methodology

Technical Report ICS-00-36
Sept 07, 2000

Lukai Cai
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA
lcai@ics.uci.edu

Mike Olivarez
Architecture and System Platforms, Motorola
M.Olivarez@Motorola.com

Dr. Daniel D. Gajski
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA
gajski@ics.uci.edu

Contents

Abstract 1
1. Introduction 1
2. SpecC 1
 2.1 SpecC methodology 1
 2.2 SpecC language 2
3. JPEG encoder 2
4. System level architecture exploration 2
5. JPEG example 3
 5.1 System level execution time concept and simulation environment 3
 5.2 Component separation exploration 4
  5.2.1 Pure SW architecture model 4
  5.2.2 Component separation exploration process 5
  5.2.3 SW_HW_sequential model 5
 5.3 Parallel execution exploration 6
  5.3.1 Parallel execution exploration process 6
  5.3.2 SW_HW_parallel_model 6
 5.4 Architecture pipeline exploration 6
  5.4.1 Architecture pipeline exploration process 7
  5.4.2 SW_2HW_parallel model 7
 5.5 Shared global memory exploration 7
  5.5.1 Shared global memory exploration process 8
  5.5.2 SW_2HW_mem model 8
 5.6 Other HW solutions 8
 5.7 Result 8
 5.8 Other operations of system level architecture exploration 10
6. Conclusions 10
References 10
Appendix 12
 A. Simulation Environment 12
  A.1 th.sc 12
  A.2 time_counter.sc 12
 B. Pure SW model 12
  B.1 jpeg.sc 12
 C. SW_HW sequential model 13
  C.1 jpeg.sc 13
  C.2 sw.sc 13
  C.3 hw.sc 14
 D. SW_HW_parallel model 15
  D.1 jpeg.sc 15
  D.2 sw.sc 15
  D.3 hw.sc 17
 E. SW_2HW_parallel model 17
  E.1 jpeg.sc 17
  E.2 sw.sc 18
  E.3 hw.sc 20
 F. SW_2HW_mem model 20
  F.1 jpeg.sc 20
  F.2 sw.sc 21
  F.3 hw.sc 23
  F.4 memory.sc 24

List of Figures

1 Block diagram of the JPEG encoder 2
2 System level architecture exploration operation types 3
3 System level architecture exploration design flow of JPEG encoder 3
4 Simulation environment of the JPEG encoder design 4
5 Pure SW architecture model for JEPG encoder 5
6 SW_HW_sequential architecture model for JPEG encoder 5
7 SW_HW_parallel architecture model for JPEG encoder 7
8 SW_2HW_parallel architecture model for JPEG encoder 8
9 SW_2HW_mem architecture model for JPEG encoder 9
10 Execution time of JPEG Encoder for different hardware solutions and target architectures 10

System Level Architecture Exploration Using the SpecC Methodology

Lukai Cai, University of California, Irvine - lcai@ics.uci.edu
Mike Olivarez, Architecture, and System Platforms, Motorola - M.Olivarez@Motorola.com
Dr. Dan Gajski, University of California Irvine - gajski@ics.uci.edu

Abstract

To implement chip design on a satisfactory target architecture, more architecture exploration should be done at higher levels of abstraction, in the earliest design stages. Using the SpecC language, an executable system level specification language, architecture exploration can be processed easily and smoothly. A SpecC methodology of system level architecture exploration is introduced within this paper to illustrate this process. The design of a JPEG encoder is used as an example to illustrate the system level architecture exploration methodology.

1 Introduction

According to Moore's Law, the number of transistors on a chip will keep growing exponentially, pushing technology towards the System-On-Chip (SOC) era. To decrease the gap between designing chips of growing complexity and increased time-to-market pressures, it is commonly agreed that the design process should shift to higher levels abstraction and the reuse of pre-designed, complex system components known as intellectual property (IP) is necessary.

In the SoC design process, the work of mapping the functionality into a target architecture, called *architecture exploration*, is one of the main problems facing the SoC designers. In the traditional design methodology, architecture exploration is not very complex because it only maps the functional specification into the fixed target architecture chosen. However, with the increase in complexity of the algorithms used in the design and the availability of different target architectures and their components, architecture exploration is more important. Thus, more target architectures should be explored to find the best solution. In the RTL level models which reflect the target architectures, contains timing and/or pin information, performing architectural exploration is too cumbersome to satisfy the time to market requirement. Therefore, a system level architecture model is required by the industry to implement fast architecture exploration.

Using the SpecC language, a system specification language developed at UC Irvine, one can implement architecture exploration easily and efficiently[1]. Using the SpecC language and methodology, mapping functionality into different target architectures in the system level is straight forward. Furthermore, the final result of an implementation on a target architecture model at the system level, can be smoothly and consistently changed to an RTL level model by using the rest of the SpecC methodology.

In this paper, the system level architecture exploration methodology is introduced as part of the SpecC methodology. Unlike the existing SpecC methodology, which mainly focuses on refining specification level into architecture level[1], system level architecture exploration focuses on changing the implementation from one target architecture to another target architecture, at the architecture level of abstraction. This methodology allows the designers to compare implementations on different target architectures and to improve the current implementation by changing some part of target architecture. In this paper. a JPEG encoder example is used to illustrate this process.

The sections of this paper are organized as described here. Section 2 summarizes the SpecC language as well as the existing SpecC methodology which is described in [1]. In section 3, a description of the JPEG encoder algorithm, which is used for our tests, is given. Section 4 introduces the system level architecture exploration methodology. In section 5, a JPEG encoder model which has been architecturally explored, is iterated through typical changes as the change guidelines are introduced which make up the system level architecture exploration process. Finally, in section 6, conclusions and future work are described.

2. SpecC

2.1 SpecC methodology

The SpecC methodology is a design methodology to implement design from pure specification into full implementation[1]. It defines four levels of modeling, from

the most abstract level to the most detailed level. The first level *specification model* represents pure specification. The second model, which is the *architecture model*, represents the implementation on target architectures with the abstraction of computation and communication specifications. The *communication model*, which is third in the hierarchy, represents the implementation on target architectures with the abstraction of computation but detailed communication model. The fourth or *implementation model* is the synthesizable RTL model.

Besides the four specification models, the SpecC methodology also defines the method of transitioning between these models. *Architecture exploration* refines the specification model to architecture model. *Communication synthesis* is the refinement from the architecture model to the communication model. Finally, the refining work from *communication model* to *implementation model* is achieved via hardware synthesis tools and software compilation.

This paper is focus on the architecture model. The flexibility of SpecC architecture model makes the system level architecture exploration easy to implement.

2.2 SpecC language

The SpecC methodology is supported by abstracting at the system-level using a specification language called SpecC[1,2]. Within the first three stages of the SpecC methodology, the current state of the design is represented by a model described in the SpecC language. In the homogeneous approach, transformations are made on the SpecC description in contrast to a heterogeneous approach, where each step also transforms the design representation at different stages of the process.

SpecC is a super-set which extends ANSI-C to allow easy reuse of the existing algorithmic and behavioral C descriptions that are common in today's industrial practice. SpecC contains all the features required to support system-level design, including structural and behavioral hierarchy, concurrency, communication with explicit separation from computation, synchronization, exception handling, timing, and explicit state transitions.

3. JPEG encoder

JPEG is an image compression standard. It is designed for compressing either full-color or gray-scale images of natural scenes[3]. Figure 1 shows the block diagram of the DCT based encode for a gray scale image. It consists of four blocks: the image fragmentation block, the DCT block, the quantization block and the entropy coding block.

In the image fragmentation block, the image is divided into non-overlapping blocks, each of which contains an 8*8 matrix of pixels. Each block is then transformed into the frequency domain in the DCT block. The DCT output coefficients are then quantized in the quantization block before it is entropy-coded in the entropy coding block. The entropy coding block consists of two stages. The first stage is either a predictive encoder for the DC coefficients or a run-length encoder for the AC coefficients. The second stage is a Huffman encoder.

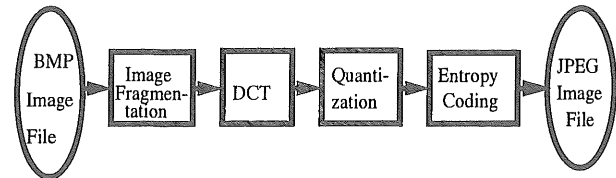

Figure 1 - Block diagram of the JPEG encoder

4.system level architecture exploration

Architecture exploration as mentioned in 2.2, is a method to refine the specification model into the architecture model. However, in the design process, different implementation models based on different target architectures are always compared, therefore, unbiased metrics are needed to be obtained. Furthermore, the most common way to improve the current design is to modify the current target architecture to achieve better performance. This need to explore between architecture models can be done using the methodology, which we call *system level architecture exploration*, introduced in this paper. The system level architecture exploration methodology includes four types of basic operations shown in Figure 2.

The first block includes three types of system level architecture exploration sub-operations: function moving exploration, component separation exploration, and component merging exploration. *Function moving* exploration moves one function block from one component into another component within the target architecture. *component separation* exploration moves one function block from one component into a new component in the target architecture. *Component merging* exploration merges two components in the target architecture into one component. In section 5.2, the guideline of *component separation* is given.

The second block includes two types of system level architecture exploration sub-operations: parallel execution and sequential execution exploration. Parallel execution exploration schedules two components, which communicate with each other, to make them execute in parallel. Sequential execution exploration schedules two components which communicate with each other to make them execute sequentially. In section 5.3, the guideline of parallel execution exploration is introduced.

The third block includes architecture pipeline exploration and component sharing exploration. Architecture pipeline exploration doubles some component as well as the function executed on the component in the target architecture, to make them run in parallel, thus, improving the performance. The component sharing exploration is reverse of architecture pipeline exploration. In section 5.4, the guideline of architecture pipeline exploration is introduced.

The fourth block includes two types of system level architecture exploration sub-operations: shared global memory exploration and message passing exploration. Shared global memory exploration changes the communication from a message passing method into a shared global memory method. Message passing exploration is the reverse process of shared global memory exploration. In 5.5, the guideline of shared global memory exploration is introduced.

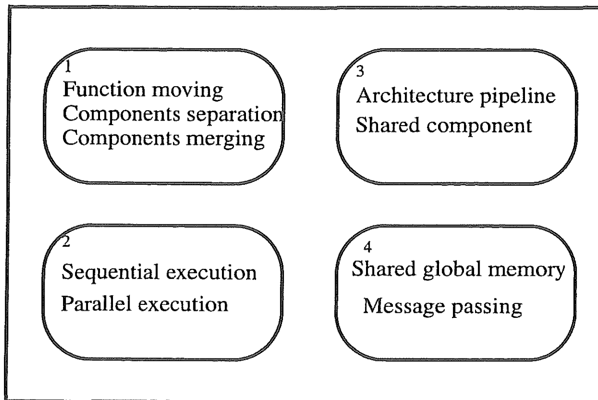

Figure 2 - system level architecture exploration operation types

Based on the SpecC language, the system level architecture exploration is implemented manually, but without difficulty. Furthermore, the guidelines of system level architecture exploration operations used in the JPEG encoder example and suggestion of automatic tools are given. Finally, system level architecture exploration can be merged into the SpecC methodology, which smoothly leads to the final stage of the design process, implementation.

5. JPEG example

In this section, a design of JPEG encoder is described to illustrate the methodology of system level architecture exploration. As shown in Figure 3, four main types of system level architecture exploration: components separation, parallel execution, architecture pipeline, and shared global memory exploration are utilized for the JPEG encoder. The guidelines of these explorations are described in the following sections. The resultant architecture models of these explorations, as well as the execution times of these models, illustrates this methodology.

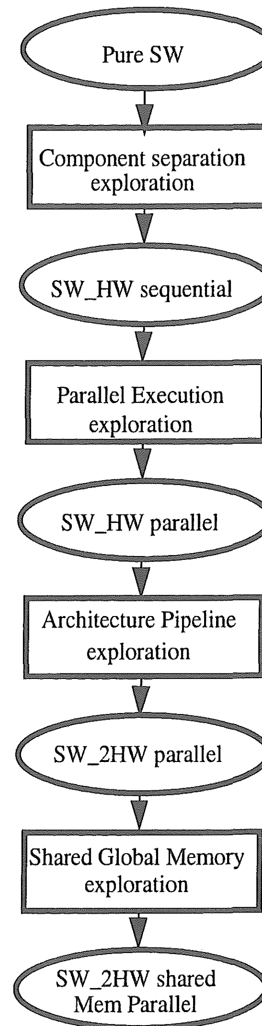

Figure 3 - system level architecture exploration design flow of JPEG encoder

5.1 System level execution time concept and simulation environment.

In system level design, leaf function nodes of the system level specification, such as the DCT block in JPEG encoder, are treated as the smallest unit. system level architecture exploration maps the different leaf nodes into different architecture components, and schedules leaf nodes either inside or among architecture components to get good performance. Therefore, the estimation time used in system level design should also be in a very abstract level, using the estimation time of leaf nodes as the smallest time unit. The total execution time of the design should be the dynamic sum of the estimation time of leaf nodes, without going into any more granularity of timing. This is the idea of system

Handledata (HD)	DCT (DCT_S)	Quantization (QZ)	Huffman (HF)
142us	745us	93us	162us

Table 1: Estimated execution time for leaf nodes on SW for each 8*8 pixel block

Solution 1	Solution 2	Solution 3	Solution 4	Solution 5	Solution 6	Solution 7
650us	600us	500us	400us	300us	200us	100us

Table 2: Timing constraint for DCT on 7HW solutions for each 8*8 pixel block

level execution time. The system level execution time is defined as the execution time which is calculated based on each leaf function node's execution time in the system level. It is a crude method, but very useful in the early stages of designing.

The execution time of each leaf node can be achieved by profiling tools of chosen processors, such as Motorola 68000 or estimation tools of ASIC's. After using profiling tools and estimation tools, the estimation time should be written back to SpecC model using the *waitfor* keyword. This step will take the *un-timed system level specification* into a new concept, *crude timed system level specification.* After rewriting, the executable specification can be executed to achieve the total estimation time of the design.

In the JPEG encoder example, only a DSP56600 processor with maximum clock frequency of 60MHz (which is called SW in this paper) and an ASIC to be designed (which is called HW in this paper) are chosen as architecture components, from the view of easy implementation. The estimation times of JPEG's four leaf nodes on SW, is given in table 1[4].

Since the DCT block causes most of the execution time, it is a good candidate to be run in HW. In some case, the ASIC (HW) is already designed and can be reused, but if not, it's execution time can be estimated. In either case, at this early stage of the design, estimation of HW execution time is unknown. In this section, the first case is tested using 650us as HW's DCT execution time for 8*8 pixel block from section 5.2 to 5.5. In section 5.6, for the second case, seven HW solutions, which have different time constraints for the DCT block, are tested to derive the trade-off between the HW performance and target architecture models. The time constraints for these HW solutions are shown in Table2. Estimates for communication, are also needed. Communication overhead between SW and HW is assumed as one pixel per SW cycle. For a 8*8 pixel block, the overhead is 1us.

In this paper, a bitmap (bmp) file which includes 180 8*8 pixel blocks, is used as input of the testbench, and the expected timing constraint is assumed to be 90ms. To count the overall JPEG encoder execution time, a time simulator, which is also written in the SpecC language, is added to the simulation environment. The time simulator runs parallel with the specification of the JPEG encoder to tabulate the total execution time, as shown in Figure 4.

5.2 Component separation exploration

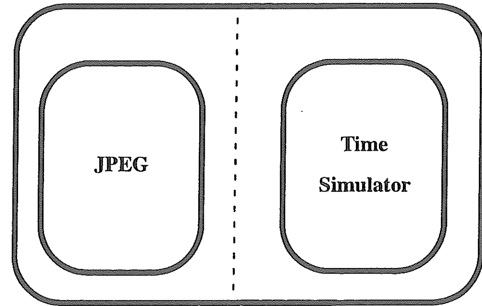

Figure 4 - Simulation environment of the JPEG encoder design

5.2.1 Pure SW architecture model

Assume in the beginning, the four basic blocks run in SW sequentially as shown in Figure 5. Compared with the specification model in [5], the architecture model uses a *behavior* named *SW* (*behavior* is the keyword in SpecC to represent architecture component or functional process) to encapsulate four leaf nodes to represent target architecture component inside the top level behavior, *JPEG Encoder behavior*. The four leaf nodes encapsulated within the *SW behavior* are HandleData block (which implements image fragmentation), DCT block, Quantization block and HuffmanEncode block (which implements entropy coding). There are also four variables in *SW behavior*: *eobmp* is the

integer which can indicates the end of the input file, hdata, ddata and qdata are immediate variables between blocks. It should be noted that the model only shows the encoder's core part, which does not involve execution related to input and output files. A reference of this part of JPEG design can be found in [5].

The execution time of the JPEG encoder in pure SW model can be estimated as follows:

```
For each 8*8 byte block,
T(block)=T(HD)+T(DCT_S)+T(QT)+T(EC)=162+745+93+162=11
42(us).

For the testbench which includes 180 blocks,
T(total)=Num(block)*T(block)=180*1142/1000=205.56(ms)
```

Using the simulation environment in figure 4, the time simulator shows the same execution timing result. To note among the results is the DCT block, which costs 134ms. This should be reduced to satisfy the timing constraint in the next step.

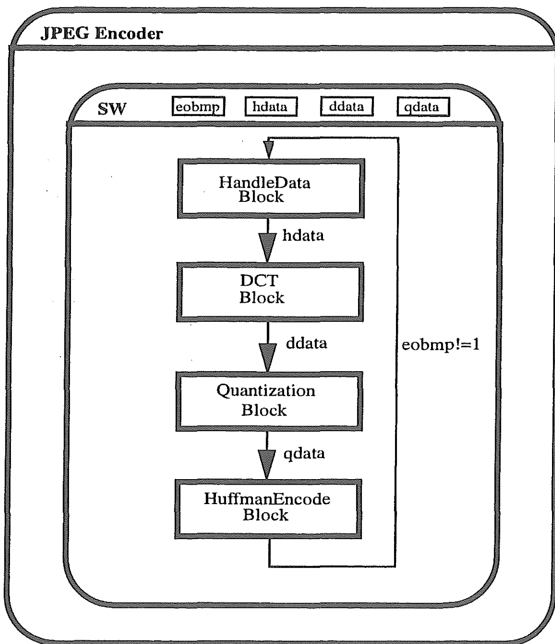

Figure 5 - Pure SW architecture model for JPEG encoder

5.2.2 Component separation exploration process

To reduce the DCT's execution time, a target architecture, which consists of HW and SW, is explored. The DCT is separated from other parts of code and executed in HW while the remaining part is executed in SW. The final target architecture is called SW_HW_sequ model, as shown in Figure 6. This modifying process is accomplished by *component separation,* which is defined as the movement of a function block from one component to a new component. Modifying the previously defined JPEG encoder from pure SW model to SW_HW_sequ model follows these guidelines of *component separation*:

1. Creation of the new component within the top level behavior. -- Create a *behavior* which represents the necessary new component(s), and make it/them run with the existing component(s). In our JPEG encoder example, the *HW behavior* is created and run in parallel with the *SW behavior,* within the *JPEG Encoder behavior.*

2. Move the function block. -- Move the function block which is being separated from the existing component(s) to new component(s). The variables used for this function are added in new component(s). In the JPEG encoder, move the DCT block from SW behavior to HW behavior. Variables eobmp, hdata, and ddata are added in HW behavior.

3. Add channels -- Add channels between the previously existing component(s) and newly created component(s). The communication functions which implements channels are added in each side of channels. In our JPEG encoder, channels ceobmp, chdata, cddata are added in the *JPEG Encoder behavior* to transfers eobmp, hdata and ddata variables. Furthermore, functions OEOBmp, OHData, IDData are added in *SW behavior* and IEOBmp, IHData, ODData are added in *HW behavior.*

4. Add necessary exit condition(s) for the created component. -- Add an outer loop in the created component to encapsulate moved function and communication functions and set an exit condition for it. In the JPEG encoder, add a loop and set eobmp!=1 as the while's loop condition.

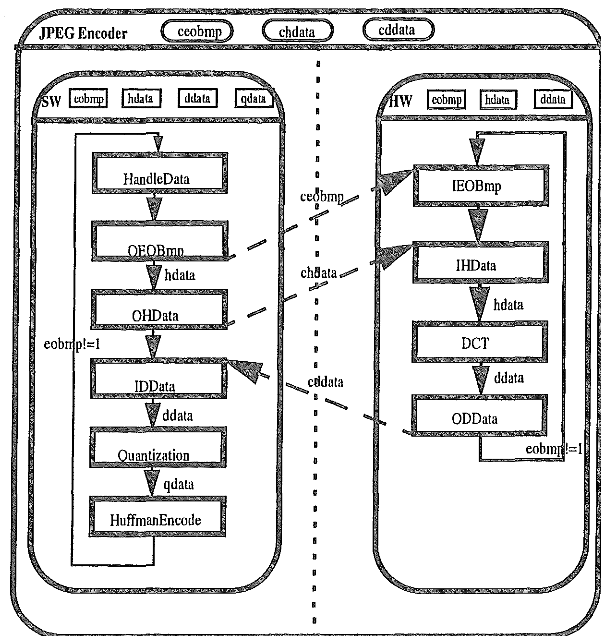

Figure 6 - SW_HW_sequential architecture model for JPEG encoder

5.2.3 SW_HW_sequential model

The resulting architecture model is called SW_HW_sequential model which is shown in Figure 6. The communications between the blocks are implemented using

the channels' synchronization functions. Since the SW and HW parts can not run until the other part finishes its execution, SW and HW are run sequentially, although, they are parallel in the architecture view.

From this model, the execution time of the JPEG encoder also can be estimated as follows:

```
For each 8*8 pixel block,
T(block)=T(HD)+T(DCT_H)+T(QT)+T(EC)+2*T(comm)=142+650
+93+162+2*1=1049(us).

For the testbench which includes 180 blocks,
T(total)=Num(block)*T(block)=180*1049/1000=188.82(ms)
```

Using the simulation environment as shown in Figure 4, the time simulator reinforces the calculated execution time. Compared with 205.56ms in the pure SW model, the execution time decreases to 188.82ms, giving a 8.1% throughput increase.

5.3 Parallel execution exploration

As mentioned in 5.2.3, in SW_HW_sequential model, SW and HW are run sequentially in the functional view. To reduce the execution time to satisfy the 90ms's timing constraint, the SW should be rescheduled to make SW and HW run in parallel. It is possible because they can execute the different 8*8 pixel blocks in the same time. This process is implemented by "parallel execution" exploration.

5.3.1 Parallel execution exploration process

The *parallel execution* exploration process is defined as the scheduling of two or more components in the system which communicate each other to make them execute in parallel. To modify the JPEG encoder, *parallel execution* exploration guidelines are created. These guidelines can solve the easy case of parallel execution exploration. In this case, one component (called child component) is the *sub-function* of another component (called parent component). In the JPEG encoder example, since the HW component is *called* by the SW component, HW is the "child component" while SW is the "parent component". The resulting architecture model, called *SW_HW_parallel model*, is shown in Figure 7.

The parallel execution exploration guidelines that have been created are as follows:

1. Group blocks into four visual blocks in the parent component. -- Group the function and communication blocks in the *parent component* into four visual blocks, named S_1, S_2, S_3 and S_4. S_1 includes all the blocks above the input communication blocks for the *child component* HW; S_2 includes input communication blocks for the *child component*; S_3 includes output communication blocks for the *child component*; and finally, S_4 includes all the blocks after S_3. In the JPEG encoder, the visual blocks are shown in Figure 7.

2. Add the control variable for each visual block. --Add control variables (control_s1 to control_s4) for each visual block to indicate if visual blocks can be executed in current time. The control variables are valid only when the output of previous visual blocks are available (For the first visual block, control variable is valid only when its output buffer is empty).

3. Add the exit condition for each visual block -- For S_1, the exit condition is the end of the input file. For other visual blocks, the exit conditions are the number of their execution times are the same as S_1.

4. Link four visual blocks. -- Use *if/else* statements to link visual blocks.

5. Change the *while* loop condition of the *parent component*'s outer loop. -- The while loop condition is true only when S_4 is not exit.

6. Add an idle visual block in the *parent component*. -- An idle visual block, which is just to add time counter, is added at the end of the *parent component*'s loop body. If in one loop execution, no other visual block is executed, the idle visual block is executed.

After exploration, the structure of the code of the *parent component* is as follows:

```
while (not exit S_4){
    if(control_s1==1 and not exit S_1){S_1}
    else if(control_s2==1 and not exit S_2){S_2}
    else if(control_s3==1 and not exit S_3){S_3}
    else if(control_s4==1 and not exit S_4){S_4}
    else{idle block};
};
```

5.3.2 SW_HW_parallel model.

After the parallel execution exploration, a SW_HW_parallel model is developed. Since the execution time of this model is difficult to estimate, the result of the time simulator is used. The time simulator shows the execution time is 117.94ms. Compared with 188.82ms in SW_HW_sequential model, it gives 37.5% throughput increase.

```
For each 8*8 pixel block,
T(block)=T(total)/Num(block)=117.94*1000/180=655us
```

It should be noted that during the exploration process some computations are added. But the system level leaf node's execution time, should be big enough, that the added execution time can be ignored.

5.4 Architecture pipeline exploration

Because SW_HW_parallel model can not satisfy the timing constraint, more system level architecture

exploration should be tried. For each 8*8 pixel block, the execution time of function blocks in SW without *waiting* is:

```
T(SW)=T(HD)+T(QT)+T(HE)+2*T(comm)=399us<<T(block)=
655us
```

The execution time of function block in HW without *waiting* is:

```
T(HW)=T(DCT_H)+2*T(comm)=652~T(block)=655(ATU)
```

As the data shows, the HW component is the bottleneck. To reduce the execution time, two HW components should be run, processing different input data, concurrently. Therefore, the target architecture which consists of two HWs and one SW running in parallel, called the *SW_2HW_parallel* model is explored. The process to change from *SW_HW_parallel* model to *SW_2HW_parallel* models is accomplished by the *architecture pipeline* exploration.

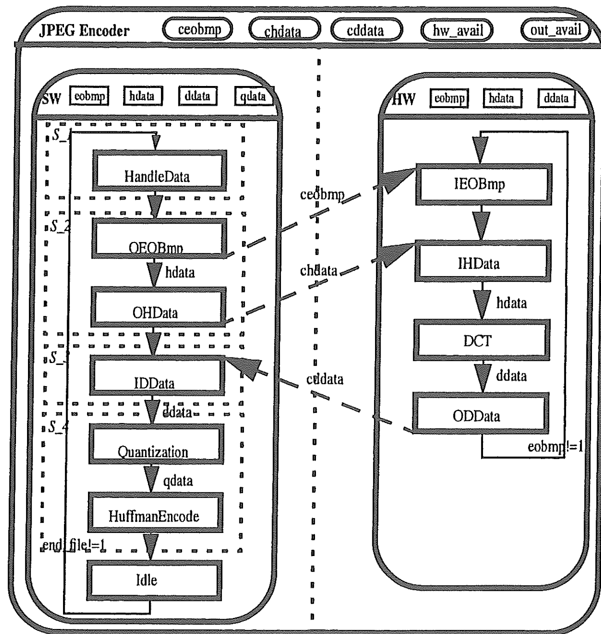

Figure 7 - SW_HW_parallel architecture model for JPEG encoder

5.4.1 Architecture pipeline exploration process

Architecture pipeline exploration is defined as the changes in the target architecture model by doubling some component(s) to make them run in parallel on different blocks of data, thus, improving the performance of the system. The exploration process from SW_HW_parallel model to SW_2HW_parallel model follows these guidelines of *architecture pipeline* exploration. In this section, the component being doubled is to be called *doubled component*. The component communicating with *doubled component* is called *communicated component*. The *architecture pipeline* exploration guideline is shown as follows:

1. Doubling of the component and related channels. -- In top level behavior (*JPEG Encoder behavior* in JPEG Encoder example), double the target component and run with the communicated component in parallel. Double the channels between *doubled component* and *communicated component*, half of which are used for communication between *communicated component* and one *doubled component*. In the JPEG Encoder example, two HWs are run in parallel with SW. The five channels of JPEG encoder are doubled as shown in Figure 8.

2. Make changes in the communicated component. -- In the *communicated component*, double visual blocks S_2 and S_3. The new visual blocks S_2_1, S_3_1 and S_2_2, S_3_2 are used to communicate with doubled component_1 and doubled component_2 respectively. The channels used in each visual block are replaced by new doubled channels respectively.

5.4.2 SW_2HW_parallel model

After *architecture pipeline* exploration, the SW_2HW_parallel model is developed as shown in Figure 8. Since the execution time is difficult to estimate. The result of the time simulator is used once again. The time simulator shows the execution time is 72.33ms. Compared with 117.94ms in the SW_HW_sequential model, it gives a 38.7% throughout increase.

```
For each 8*8 byte block,
T(block)=T(total)/Num(block)=72.33*1000/180=402us
```

This result can satisfy the timing constraint requirement, but more refinement can still be tried. This may give a solution which will achieve more desired results later in the design.

5.5 Shared global memory exploration

In the SW_HW model, the data communication between HW and SW is by message passing method. In some cases, a shared memory is needed. Therefore, the JPEG encoder example will be implemented in a shared global memory communication model. This architecture model is called a *shared memory model*, which is implemented by *shared global memory* exploration.

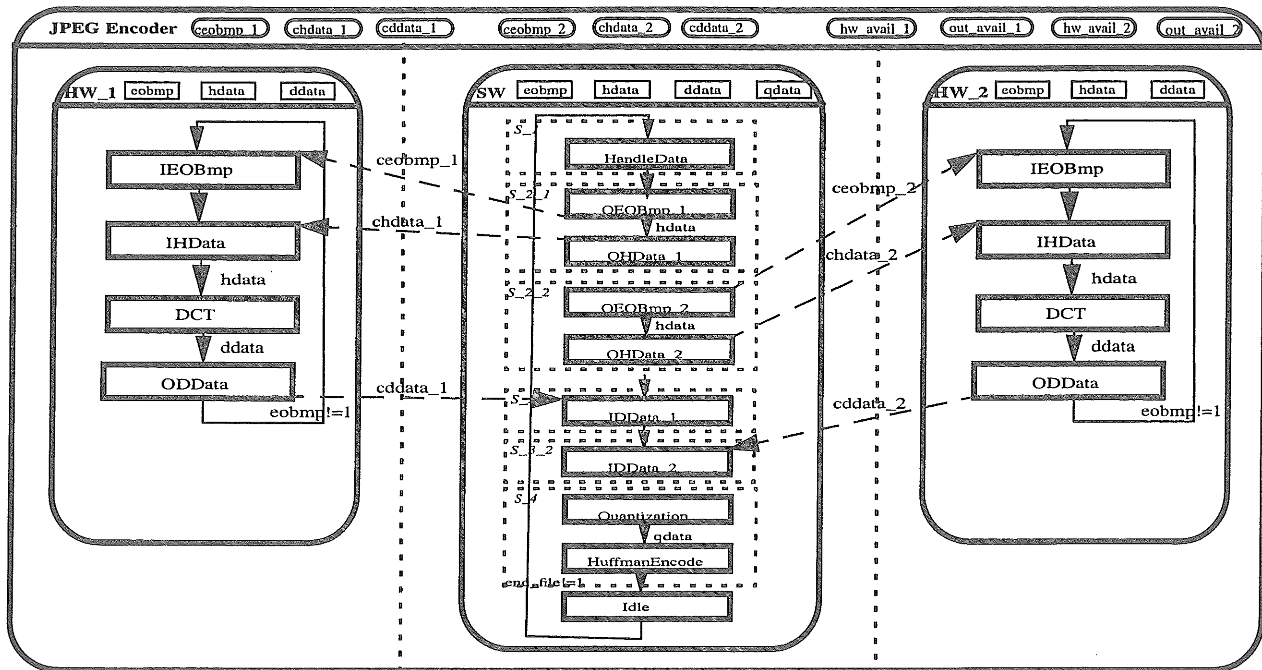

Figure 8- SW_2HW_parallel architecture model for JPEG encoder

5.5.1 Shared global memory exploration process

Shared global memory exploration is defined as the changes needed for data communication message passing to a shared global memory. In our JPEG encoder example, the data communicated between HW and SW, which includes 8*8 pixel block inputs and 8*8 pixel block outputs of the DCT, will be written/read through a global memory. The guideline of *shared global memory* exploration is as follows:

1.Create MEM component. -- Create a component which is called *MEM behavior* to run parallel with other components. In the JPEG encoder, MEM will be run in parallel with one SW and two HW components.

2.Create Channels for MEM. -- Use four channels to replace one previous channel between *communicated components*, called A and B. These four channels are 1) channel for memory enable variable from Component A to Memory, 2) channel for transferred data from component A to memory, 3) channel for memory enable variable form component B to memory, and 4) channel for transferred data from component B to memory. For our JPEG example, components SW and HW have been updated as such.

3.Implement MEM component. -- In the memory component, a flag is assigned for each *memory storage variable*. Use this flag to indicate if the data in memory is valid to read or valid to write. When the flag is 1, the data can be read, but can not be written. If flag is 0, the data can be written, but can not be read.

5.5.2 SW_2HW_mem model

After the *shared global memory* exploration, the SW_2HW_mem model is developed as shown in figure 9. When running the testbench, two HW components execute the different 8*8 pixel input block alternatively. Since the execution time is difficult to estimate. The result of the time simulator is needed once again. The time simulator shows the execution time is 72.69ms, compared with 72.33ms in SW_2HW_parallel model. The difference between two model's execution time is very little, but comes from the memory channels.

```
For each 8*8 pixel block,
T(block)=T(total)/Num(block)=72.69*1000/180=404us
```

This result will also satisfy the timing constraint, and shows how various configurations can be achieved using this methodology.

5.6 Other HW solutions

system level architecture exploration from 5.2 to 5.5 is implemented based on HW solution 1 which has 650us as the time constraint of DCT block when running 8*8 pixel block. Using other HW solutions with SW as the target architecture components, the execution times are as shown in Table 3. The data with underline symbol represents the solution which can satisfy the timing constraint.

5.7 Result

From Figure 10, which is derived on Table3, the solutions of the two cases mentioned in 5.1 can be achieved.

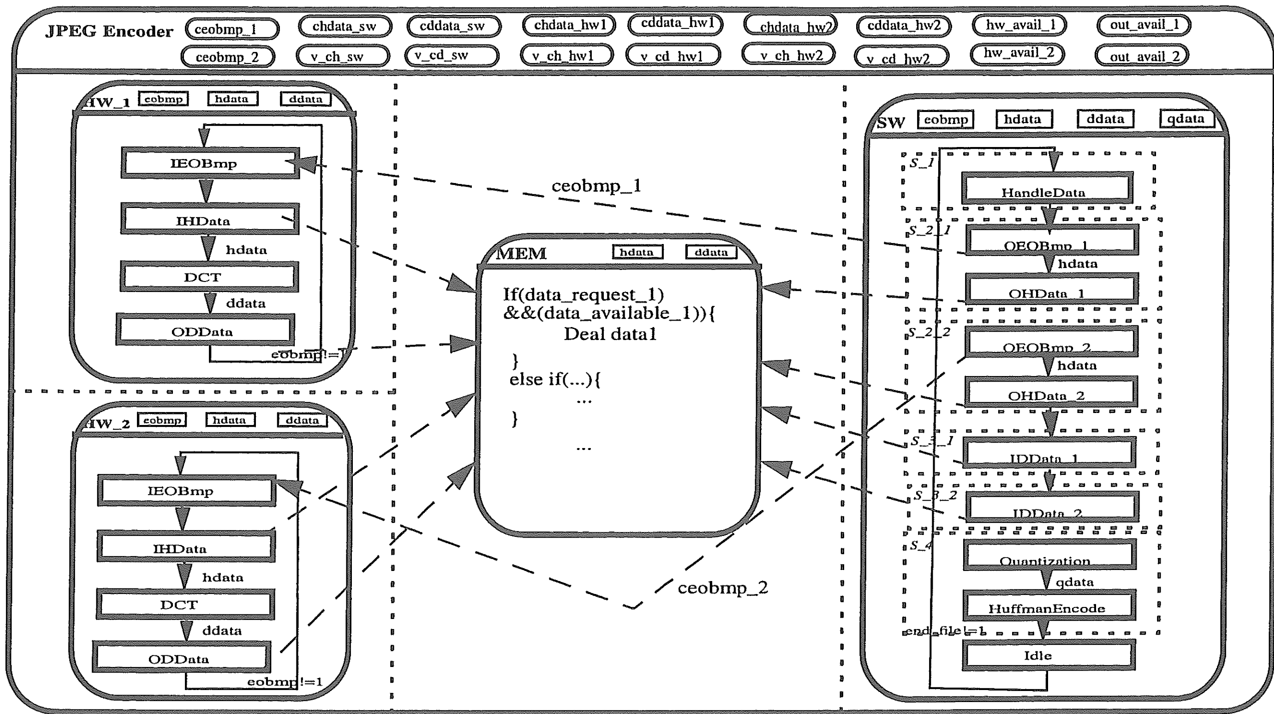

Figure 9- SW_2HW_mem architecture model for JPEG encoder

	Pure_SW model	SW_HW_sequential	SW_HW_parallel	SW_2HW_parallel	SW_2HW_memory
HW1-650us	205.56ms	188.82ms	117.94ms	72.33ms	72.69ms
HW2-600us	205.56ms	179.82ms	108.94ms	72.23ms	72.59ms
HW3-500us	205.56ms	161.82ms	90.94ms	72.04ms	72.40ms
HW4-400us	205.56ms	143.82ms	72.94ms	71.94ms	72.30ms
HW5-300us	205.56ms	125.82ms	72.03ms	71.84ms	72.20ms
HW6-200us	205.56ms	107.82ms	71.88ms	71.82ms	72.18ms
HW7-100us	205.56ms	89.82ms	71.82ms	71.82ms	72.18ms

Table 3- Execution time of JPEG Encoder for different Hardware solutions

In case 1, which the execution time of HW can be estimated as 650us for executing the DCT block before the architecture exploration. The system level architecture exploration is processed and the performance of JPEG is improved. Two target architecture models, SW_2HW_parallel and SW_2HW_memory can be chosen as the final result, as shown in the oval in Figure 10.

In case 2, where the execution time of HW is unknown, several HWs which have different time constraint are tested. In this case, system level architecture exploration can be used to make trade-offs between the target architectures and the different HW solutions. As shown in Figure 10, Pure_SW target architecture can not satisfy the time constraint. SW_HW_sequential model can be chosen only if HW has a 100us time constraint on the DCT block, such as HW solution 7, and can be implemented (as shown in cycle 1). If SW_HW_parallel model is chosen, the timing constraint for DCT on HW has to less than 500us, such as HW solution 4 (as shown in cycle 2). If SW_2HW_parallel model or SW_2HW_memory model is chosen, the time constraint of HW can be very low, therefore the time constraint of HW at 650us is enough, such as HW solution 1 (as shown in cycle 3).

Using the SpecC language and the system level architecture exploration guideline, the manually explored JPEG example mentioned in this paper can be implemented

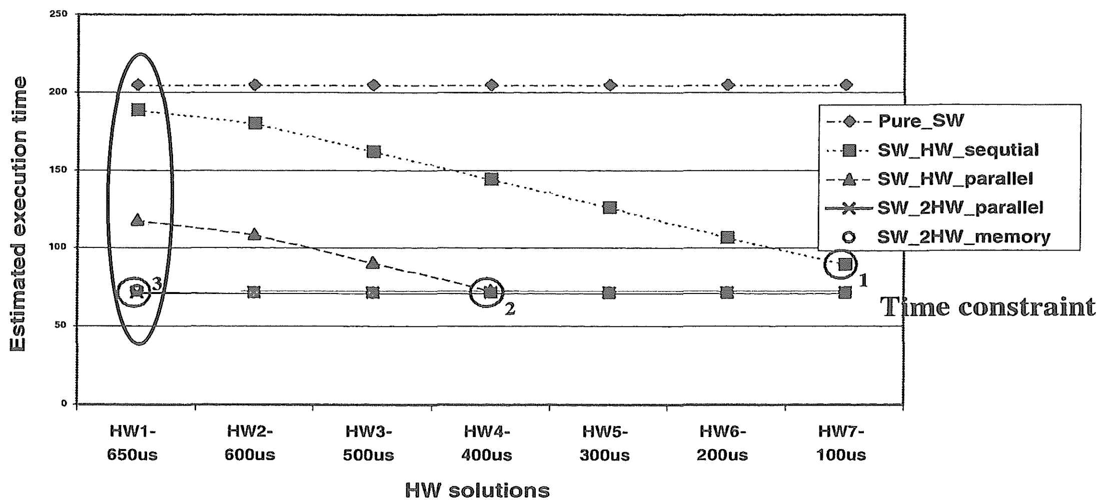

Figure 10- Execution time of JPEG Encoder for different hardware solutions and target architectures

in few hours. Although the resulting architecture models are not as complete as the ones tested within the methodology mentioned in [1], this paper shows techniques which can refine the existing SpecC methodology. *Channel partitioning* which should be implemented to optimize the channels in the top level have been left out, but would be the next step in the SpecC Methodology.

5.8. Other operations of system level architecture exploration

In section 5, the guidelines of four types of exploration are described. For the exploration operation types that were not detailed, implementation can be accomplished in one of two ways. This is true because most are reverse operations of the four exploration operations described in section 5 (function moving exploration is quite similar to component separation exploration). The first method is to design the guideline for each operation independently. The second method is to use the existing four operations of exploration described in section 5.2 to 5.5. The second method can be accomplished since the four methods implemented in section 5.2 to 5.5 are to change from a simple situation to a complex situation. While the reverse operations are to change from the complex situation to simple situation, the changes from the simple to the complex solutions are more difficult to implement. Furthermore, since the initial specification is sequentially executed in one component and uses a message passing communication method, the path of system level architecture exploration operations can be traced. It is easy to go back to the immediate previous step, which contains the needed simple situation. In the second method, system level architecture exploration is then restarted using the four operations mentioned in section 5.2 to 5.5 to continue its exploration from this immediate step. Therefore, the second method can be implemented more easily.

6 Conclusions

This paper introduces a new methodology in system level design, system level architecture exploration. system level architecture exploration is the methodology of modifying a design implementation from one target architecture model to another target architecture model, within the architecture level of the SpecC methodology[1]. Since the architecture level model of the SpecC methodology is in an abstract level, the process of system level architecture exploration is fast. Furthermore, a new timing model, "timed system level specification model" is clearly defined and first used in the SpecC methodology.

This paper uses a JPEG encoder as an example to show the process of system level architecture exploration and guidelines for these operations are given. These guidelines show that system level architecture exploration can be implemented easily with manual modification. To implement these modifications more easily, automated tools can be designed for system level architecture exploration.

system level architecture exploration is one part of the SpecC methodology. It refined the architecture level of the SpecC methodology. Using this methodology within the SpecC design flow, the fast estimation of implementation, and finding a suitable target architecture earlier in the design cycle, between can be achieved.

References

[1] [GZDGZ] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, S. Zhao, *SpecC: specification language and methodology,* Kluwer Academic Publishers, March, 2000

[2] [GZDH] A. Gerstlauer, S. Zhao, D. Gajski, A. Horak, *Design of a GSM Vocoder using SpecC methodology,* University of California, Irvine, Technical Report ICS-99-xx, February 1999.

[3] [DCT] V. Bhaskaran, K. Konstantinides, *Image and Video compression standards,* Second Edition, Kluwer Academic Publisher, 1997

[4] [CPCG] L. Cai, J. Peng, C. Chang, A. Gerstlauer, H. Li, A. Selka, C. Siska, L. Sun, S. Zhao, D. Gajski, *Design of a JPEG Encoding System*, University of California, Irvine, Technical Report ICS-99-xx, September 1999.

[5] [YDLG] H. Yin, H. Du, T. Lee, D,. Gajski. *Design of a JPEG Encoder using SpecC methodology.* University of California, Irvine, Technical Report ICS-00-xx, July 2000

Appendix

A. Simulation Environment

A.1 tb.sc

```
/*******************************************
 Project: system level architecture exploration
  Stage: Simulation environment
  Filename: tb.sc
  Last change: 08/17/00
  Author: Lukai Cai
*******************************************/


import "io";
import "jpeg";
import "time_counter";

behavior Main
{
 char* ifname;
 char* ofname;

 // Channels
 cSyncInt  header;
 cSyncByte pixel;
 cSyncByte data;
 cFlagInt      end_file;

 Input   input(ifname, header, pixel);
 Jpeg    jpeg(header, pixel, data, end_file);
 Output  output(ofname, data);
 time_countertc(end_file);

 int main (int argc, char** argv)
 {
  // Command line arguments
  if (argc < 2) {
   error("Usage: %s infile [ outfile ]\n", argv[0]);
  }
  ifname = argv[1];
  if (argc >= 3) {
   ofname = argv[2];
  } else {
   ofname = 0;
  }

  // And now run the stuff...
  par {
   input.main();
   jpeg.main();
   output.main();
   tc.main();
  }

  return 0;
 }

};
```

A.2 time_counter.sc

```
/*******************************************
 Project: system level architecture exploration
  Stage: Simulation enviroment
  Filename: time_counter.sc
  Last change: 08/17/00
  Auther: Lukai Cai
*******************************************/

#include "const.sh"

import "chann";
import "global";

behavior time_counter(iBlckFlagRev end_file){

void main(void){

static int run_time;


run_time=0;
end_file.Invalid_flag();


while(end_file.Check_flag()==0){
waitfor(1);
run_time++;
}

printf("\n Timing simulater report: run_time is %d\n",
run_time-1);

}
};
```

B Pure SW model

B.1jpeg.sc

```
/*******************************************
 Project: system level architecture exploration
  Stage: Pure Software model
  Filename: jpeg.sc
  Last change: 08/17/00
  Auther: Lukai Cai
*******************************************/


import "handle";
import "dct";
import "quant";
import "huff";
```

```
#include "const.sh"

behavior SW(iBlckRecvInt header_ch, iBlckRecvByte pixel_ch,
iBlckSendByte data_ch, iBlckFlagSend end_file )
{
int block_no ;
inthdata[64];
intddata[64];
intqdata[64];
inteobmp;

HandleDatahandledata(header_ch, pixel_ch, data_ch,
hdata, eobmp);
DCTdct(hdata, ddata);
Quantizationquantization(ddata, qdata);
HuffmanEncodehuffmanencode(qdata, data_ch);

  void main(void) {
printf("*********************\n");
printf("JPEG Encoder Begin...\n");
printf("*********************\n");
block_no = 0 ;
eobmp = 0;
do
{
block_no ++ ;
printf("Processing Block %dth...\n", block_no);

handledata.main();
dct.main();
quantization.main();
huffmanencode.main();
}while(eobmp!=1);
WriteBits(-1, 0, data_ch);
WriteMarker(M_EOI, data_ch);
end_file.Valid_flag();
printf("******************\n");
printf("JPEG Encoder End...\n");
printf("******************\n");
  }
};


behavior Jpeg(iBlckRecvInt header_ch, iBlckRecvByte pixel_ch,
iBlckSendByte data_ch, iBlckFlagSend end_file ){

SW  sw_exec(header_ch, pixel_ch, data_ch, end_file);

void main(){
sw_exec.main();
}
};
```

C SW_HW sequential model

C.1 jpeg.sc

```
/*******************************************
 Project: system level architecture exploration
   Stage: SW_HW_sequential model
   Filename: jpeg.sc
   Last change: 08/17/00
   Auther: Lukai Cai
*******************************************/

import "sw";
import "hw";
import "time_counter";

#include "const.sh"

behavior Jpeg(iBlckRecvInt header_ch, iBlckRecvByte pixel_ch,
iBlckSendByte data_ch, iBlckFlagSend end_file_tc)
{

cSyncBlockchdata, cddata;
cSyncIntceobmp;


SWsw(header_ch, pixel_ch, data_ch,
chdata, ceobmp, cddata);
HWhw(chdata, ceobmp, cddata);



  void main(void) {
printf("*********************\n");
printf("JPEG Encoder Begin...\n");
printf("*********************\n");
par
{

sw.main();
hw.main();

}
end_file_tc.Valid_flag();
printf("******************\n");
printf("JPEG Encoder End...\n");
printf("******************\n");
  }
};
```

C.2 sw.sc

```
/*******************************************
 Project: system level architecture exploration
   Stage: SW_HW_sequential model
   Filename: sw.sc
   Last change: 08/17/00
   Auther: Lukai Cai
```

```
*******************************************/

import "handle";
import "quant";
import "huff";

#include "const.sh"


behavior BOHData(in int hdata[64], iBlckSendBlock CHData)
{
void main ( void )
{
// send item hdata over channel
CHData.send ( hdata ) ;
waitfor(1);
}
};

behavior BOEOBmp(in int eobmp, iBlckSendInt CEOBmp)
{
void main ( void )
{
// send item eobmp over channel
CEOBmp.send ( eobmp ) ;
}
};

behavior BIDData(out int ddata[64], iBlckRecvBlock CDData)
{
void main ( void )
{
// receive item ddata over channel
CDData.receive ( ddata ) ;
waitfor(1);
}
};

behavior SW(iBlckRecvInt header_ch, iBlckRecvByte
pixel_ch,
iBlckSendByte data_ch,
iBlckSendBlock CHData,
iBlckSendInt CEOBmp,
iBlckRecvBlock CDData)
{
inthdata[64];
intddata[64];
intqdata[64];
inteobmp;
intblock_no ; //for display
inti ;

HandleDatahandledata(header_ch, pixel_ch, data_ch,
hdata, eobmp);
Quantizationquantization(ddata, qdata);
HuffmanEncodehuffmanencode(qdata, data_ch);

BOEOBmpOEOBmp(eobmp, CEOBmp);
BOHDataOHData(hdata, CHData);
BIDDataIDData(ddata, CDData);

void main(void) {

eobmp = 0 ;
block_no = 0 ;

do
{
block_no ++ ;
printf("Processing Block %dth...\n", block_no);

handledata.main();// original behavior

//send data from ColdFire to DCT
OEOBmp.main() ;// send eobmp ouput
OHData.main() ;// send hdata ouput

//Receive data from DCT
IDData.main() ;// receive ddata ouput

quantization.main();// original behavior
huffmanencode.main();// original behavior


}while(eobmp!=1); // end of while

WriteBits(-1, 0, data_ch);
WriteMarker(M_EOI, data_ch);

}// end of main
}; // end of behavior
```

C.3 hw.sc

```
/*******************************************
Project: system level architecture exploration
Stage: SW_HW_sequential model
Filename: hw.sc
Last change: 08/17/00
Auther: Lukai Cai
*******************************************/

import "dct";

#include "const.sh"

behavior BIHData(out int hdata[64], iBlckRecvBlock CHData)
{
void main ( void )
{
CHData.receive ( hdata ) ;
waitfor(1);
}
};

behavior BIEOBmp(out int eobmp, iBlckRecvInt CEOBmp)
```

```
{
void main ( void )
{
eobmp = CEOBmp.receive() ;
}
};

behavior BODData(in int ddata[64], iBlckSendBlock CDData)
{
void main ( void )
{
CDData.send ( ddata ) ;
waitfor(1);
}
};

behavior HW(iBlckRecvBlock CHData,
iBlckRecvInt CEOBmp,
      iBlckSendBlock CDData)
{
inthdata[64];
intddata[64];
inteobmp;

DCTdct(hdata, ddata);
BIHData IHData(hdata, CHData);
BIEOBmp IEOBmp(eobmp, CEOBmp);
BODData ODData(ddata, CDData);

   void main(void) {
do
{
//Receive data from ColdFire
IEOBmp.main();// receive eobmp output
IHData.main();// receive hdata output

dct.main();// original behavior

//send data from DCT to ColdFire
ODData.main();// send ddata output
}while(eobmp!=1) ;  // end of while
   }  // end of main
};  // end of behavior
```

D. SW_HS_parallel model

D.1 jpeg.sc

```
/*********************************************
 Project: system level architecture exploration
  Stage: SW_HW_parallel model
  Filename: jpeg.sc
  Last change: 08/17/00
  Auther: Lukai Cai
*********************************************/

import "sw";
import "hw";
import "time_counter";

#include "const.sh"

behavior Jpeg(iBlckRecvInt header_ch, iBlckRecvByte
pixel_ch,
  iBlckSendByte data_ch,iBlckFlagSend end_file_tc)
{

cSyncBlockchdata, cddata;
cSyncIntceobmp;
cFlagInt valid_hw;
cFlagInt valid_result;


SWsw(header_ch, pixel_ch, data_ch,
chdata, ceobmp, cddata, valid_hw, valid_result);
HWhw(chdata, ceobmp, cddata, valid_hw, valid_result);



   void main(void) {
printf("*********************\n");
printf("JPEG Encoder Begin...\n");
printf("*********************\n");
par
{
sw.main();
hw.main();
}
end_file_tc.Valid_flag();
printf("*******************\n");
printf("JPEG Encoder End...\n");
printf("*******************\n");
   }
};
```

D.2 sw.sc

```
/*********************************************
 Project: system level architecture exploration
  Stage: SW_HW_parallel model
  Filename: hw.sc
  Last change: 08/17/00
  Auther: Lukai Cai
*********************************************/
import "handle";
import "quant";
import "huff";

#include "const.sh"


behavior BOHData(in int hdata[64], iBlckSendBlock CHData)
{
void main ( void )
{
```

```
// send item hdata over channel
CHData.send ( hdata ) ;
waitfor(1);
}
};

behavior BOEOBmp(in int eobmp, iBlckSendInt CEOBmp)
{
void main ( void )
{
// send item eobmp over channel
CEOBmp.send ( eobmp ) ;
}
};

behavior BIDData(out int ddata[64], iBlckRecvBlock CDData)
{
void main ( void )
{
// receive item ddata over channel
CDData.receive ( ddata ) ;
waitfor(1);
}
};

behavior SW(iBlckRecvInt header_ch, iBlckRecvByte
pixel_ch,
iBlckSendByte data_ch,
iBlckSendBlock CHData,
iBlckSendInt CEOBmp,
iBlckRecvBlock CDData,
iBlckFlagRev valid_hw,
iBlckFlagRev valid_result)
{
inthdata[64];
intddata[64];
intqdata[64];
inteobmp;
intblock_no ; //for display
inti ;

HandleDatahandledata(header_ch, pixel_ch, data_ch,
hdata, eobmp);
Quantizationquantization(ddata, qdata);
HuffmanEncodehuffmanencode(qdata, data_ch);

BOEOBmpOEOBmp(eobmp, CEOBmp);
BOHDataOHData(hdata, CHData);
BIDDataIDData(ddata, CDData);

void main(void) {

int handle_count, valid_after_hw;
int end_file;
int a, b, c, d;

eobmp = 0 ;
block_no = 0 ;
handle_count=0;
valid_after_hw=0;
end_file=0;
a=0;
b=0;
c=0;
d=0;


do
{

/*prepare for the hardware */
if((handle_count==0)&& (eobmp!=1)){
handledata.main();// original behavior
handle_count=1;
a++;
}
else if((valid_hw.Check_flag()==1)&&(b<a)){
valid_hw.Invalid_flag();
//send data from ColdFire to DCT
OEOBmp.main() ;// send eobmp ouput
OHData.main() ;// send hdata ouput
block_no ++ ;
handle_count--;
b++;
printf("Processing Block %dth...\n", block_no);
}


else if(valid_after_hw==1){
quantization.main();// original behavior
huffmanencode.main();// original behavior
valid_after_hw=0;
c++;

if(c==a){
end_file=1;
}


}

else
if((valid_result.Check_flag()==1)&&(d<a)&&(valid_after_hw==
0)){
//Receive data from DCT
valid_result.Invalid_flag(); //tell sw can receive data
IDData.main() ;// receive ddata ouput
valid_after_hw=1;
d++;


}


else {
waitfor(1);
```

```
}


}while(end_file!=1); // end of while

WriteBits(-1, 0, data_ch);
WriteMarker(M_EOI, data_ch);
/* printf("\n a=%d, b=%d, c=%d, d=%d", a, b,c, d);
printf("\n time=%d", run_time); */


   }// end of main
}; // end of behavior
```

D.3 hw.sc

```
/*********************************************
 Project: system level architecture exploration
   Stage: SW_HW_parallel model
   Filename: hw.sc
   Last change: 08/17/00
   Auther: Lukai Cai
*********************************************/

import "dct";

#include "const.sh"

behavior BIHData(out int hdata[64], iBlckRecvBlock CHData)
{
void main ( void )
{
CHData.receive ( hdata ) ;
waitfor(1);

}
};

behavior BIEOBmp(out int eobmp, iBlckRecvInt CEOBmp)
{
void main ( void )
{
eobmp = CEOBmp.receive() ;
}
};

behavior BODData(in int ddata[64], iBlckSendBlock CDData)
{
void main ( void )
{
CDData.send ( ddata ) ;
waitfor(1);
}
};

behavior HW(iBlckRecvBlock CHData,
iBlckRecvInt CEOBmp,
    iBlckSendBlock CDData,
iBlckFlagSend valid_hw,
iBlckFlagSend valid_result)
{
inthdata[64];
intddata[64];
inteobmp;

DCTdct(hdata, ddata);
BIHData IHData(hdata, CHData);
BIEOBmp IEOBmp(eobmp, CEOBmp);
BODData ODData(ddata, CDData);

   void main(void) {
valid_hw.Valid_flag();
valid_result.Invalid_flag();

do
{
//Receive data from ColdFire
IEOBmp.main();// receive eobmp output
IHData.main();// receive hdata output

dct.main();// original behavior

valid_result.Valid_flag(); //tell sw can receive data

//send data from DCT to ColdFire
ODData.main();// send ddata output
valid_hw.Valid_flag(); //tell sw can deal other imput


}while(eobmp!=1) ; // end of while

   } // end of main
}; // end of behavior
```

E SW_2HW_parallel model

E.1 jpeg.sc

```
/*********************************************
 Project: system level architecture exploration
   Stage: SW_2HW_parallel model
   Filename: jpeg.sc
   Last change: 08/17/00
   Auther: Lukai Cai
*********************************************/
import "sw";
import "hw";

#include "const.sh"

behavior   Jpeg(iBlckRecvInt   header_ch,   iBlckRecvByte
pixel_ch,
iBlckSendByte data_ch, iBlckFlagSend end_file_tc)
{

cSyncBlockchdata, cddata;
```

```
cSyncIntceobmp;
cSyncBlockchdata_2, cddata_2;
cSyncIntceobmp_2;
cFlagInt valid_hw_1, valid_hw_2;
cFlagInt valid_result_1, valid_result_2;
event end_SW;


SWsw(header_ch, pixel_ch, data_ch,
chdata, ceobmp, cddata,chdata_2, ceobmp_2, cddata_2,
valid_hw_1, valid_result_1, valid_hw_2, valid_result_2,
end_SW);
HWhw_1(chdata, ceobmp, cddata, valid_hw_1,
valid_result_1, end_SW);
HWhw_2(chdata_2, ceobmp_2, cddata_2, valid_hw_2,
valid_result_2, end_SW);



void main(void) {
printf("********************\n");
printf("JPEG Encoder Begin...\n");
printf("********************\n");

par
{

sw.main();
hw_1.main();
hw_2.main();

}
end_file_tc.Valid_flag();
printf("*******************\n");
printf("JPEG Encoder End...\n");
printf("*******************\n");

}
};
```

E.2 sw.sc

```
/*******************************************
Project: system level architecture exploration
Stage: SW_2HW_parallel model
Filename: sw.sc
Last change: 08/17/00
Auther: Lukai Cai
*******************************************/

import "handle";
import "quant";
import "huff";

#include "const.sh"

behavior BOHData(in int hdata[64], iBlckSendBlock CHData)
{
void main ( void )
{
// send item hdata over channel
CHData.send ( hdata ) ;
waitfor(1);
}
};

behavior BOEOBmp(in int eobmp, iBlckSendInt CEOBmp)
{
void main ( void )
{
// send item eobmp over channel
CEOBmp.send ( eobmp ) ;
}
};

behavior BIDData(out int ddata[64], iBlckRecvBlock CDData)
{
void main ( void )
{
// receive item ddata over channel
CDData.receive ( ddata ) ;
waitfor(1);
}
};

behavior SW(iBlckRecvInt header_ch, iBlckRecvByte
pixel_ch,
iBlckSendByte data_ch,
iBlckSendBlock CHData,
iBlckSendInt CEOBmp,
iBlckRecvBlock CDData,
iBlckSendBlock CHData_2,
iBlckSendInt CEOBmp_2,
iBlckRecvBlock CDData_2,
iBlckFlagRev valid_hw_1,
iBlckFlagRev valid_result_1,
iBlckFlagRev valid_hw_2,
iBlckFlagRev valid_result_2,
event end_SW)
{
inthdata[64];
intddata[64];
intqdata[64];
inteobmp;
intblock_no ; //for display
inti ;

HandleDatahandledata(header_ch, pixel_ch, data_ch,
hdata, eobmp);
Quantizationquantization(ddata, qdata);
HuffmanEncodehuffmanencode(qdata, data_ch);

BOEOBmpOEOBmp(eobmp, CEOBmp);
BOHDataOHData(hdata, CHData);
```

```
BIDDataIDData(ddata, CDData);

BOEOBmpOEOBmp_2(eobmp, CEOBmp_2);
BOHDataOHData_2(hdata, CHData_2);
BIDDataIDData_2(ddata, CDData_2);


   void main(void) {

int handle_count, valid_after_hw;
int end_file;
int a, b, c, d;

eobmp = 0 ;
block_no = 0 ;
handle_count=0;
valid_after_hw=0;
end_file=0;
a=0;
b=0;
c=0;
d=0;


do
{



/*prepare for the hardware */
if((handle_count==0)&& (eobmp!=1)){
handledata.main();// original behavior
handle_count=1;
a++;
}
else if((valid_hw_1.Check_flag()==1)&&(b<a)){
valid_hw_1.Invalid_flag();
//send data from ColdFire to DCT
OEOBmp.main() ;// send eobmp ouput
OHData.main() ;// send hdata ouput
block_no ++ ;
handle_count--;
b++;
printf("Processing Block %dth in hw_1...\n", block_no);
}

else if((valid_hw_2.Check_flag()==1)&&(b<a)){
valid_hw_2.Invalid_flag();
//send data from ColdFire to DCT
OEOBmp_2.main() ;// send eobmp ouput
OHData_2.main() ;// send hdata ouput
block_no ++ ;
handle_count--;
b++;
printf("Processing Block %dth in hw_2...\n", block_no);
}

 else
if((valid_result_1.Check_flag()==1)&&(d<a)&&(valid_after_hw
==0)){
 valid_result_1.Invalid_flag();
 //Receive data from DCT
 IDData.main() ;// receive ddata ouput
 valid_after_hw=1;
 d++;


 }

 else
if((valid_result_2.Check_flag()==1)&&(d<a)&&(valid_after_hw
==0)){

 valid_result_2.Invalid_flag();
 //Receive data from DCT
 IDData_2.main() ;// receive ddata ouput
 valid_after_hw=1;
 d++;


 }

 else if(valid_after_hw==1){
 quantization.main();// original behavior
 huffmanencode.main();// original behavior
 valid_after_hw=0;
 c++;
 if(c==a){
 end_file=1;
 notify(end_SW);
 }


 }

 else {

 waitfor(1);
 }


 }while(end_file!=1); // end of while


 WriteBits(-1, 0, data_ch);
 WriteMarker(M_EOI, data_ch);



   }// end of main
 }; // end of behavior
```

E.3 hw.sc

```
/*******************************************
 Project: system level architecture exploration
  Stage: SW_2HW_parallel model
  Filename: hw.sc
  Last change: 08/17/00
  Auther: Lukai Cai
*******************************************/
import "dct";

#include "const.sh"

behavior BIHData(out int hdata[64], iBlckRecvBlock CHData)
{
void main ( void )
{
CHData.receive ( hdata ) ;
waitfor(1);
}
};

behavior BIEOBmp(out int eobmp, iBlckRecvInt CEOBmp)
{
void main ( void )
{
eobmp = CEOBmp.receive() ;

}
};

behavior BODData(in int ddata[64], iBlckSendBlock CDData)
{
void main ( void )
{
CDData.send ( ddata ) ;
waitfor(1);
}
};

behavior HW_main(iBlckRecvBlock CHData,
iBlckRecvInt CEOBmp,
     iBlckSendBlock CDData,
iBlckFlagSend valid_hw,
iBlckFlagSend valid_result)
{
inthdata[64];
intddata[64];
inteobmp;

DCTdct(hdata, ddata);
BIHData IHData(hdata, CHData);
BIEOBmp IEOBmp(eobmp, CEOBmp);
BODData ODData(ddata, CDData);

  void main(void) {
valid_hw.Valid_flag();
valid_result.Invalid_flag();
do
{
//Receive data from ColdFire
IEOBmp.main();// receive eobmp output
IHData.main();// receive hdata output

dct.main();// original behavior

valid_result.Valid_flag(); //tell sw can receive data

//send data from DCT to ColdFire
ODData.main();// send ddata output
valid_hw.Valid_flag(); //tell sw can deal other imput


}while(eobmp!=1) ; // end of while

  } // end of main
}; // end of behavior

behavior IDLE(){
void main(){}
};

behavior HW(iBlckRecvBlock CHData,
iBlckRecvInt CEOBmp,
     iBlckSendBlock CDData,
iBlckFlagSend valid_hw,
iBlckFlagSend valid_result,
event end_SW){

HW_main HW_main_exec(CHData, CEOBmp, CDData,
valid_hw, valid_result);
IDLE idle_exec();

void main(){
try{
HW_main_exec.main();}
trap(end_SW){idle_exec.main();}
}
};
```

F Sw_2HW_mem model

F.1 jpeg.sc

```
/*******************************************
 Project: system level architecture exploration
  Stage: SW_HW_parallel mem model
  Filename: jpeg.sc
  Last change: 08/17/00
  Auther: Lukai Cai
*******************************************/

import "sw";
import "hw";
import "mem";
```

```
#include "const.sh"

behavior Jpeg(iBlckRecvInt header_ch, iBlckRecvByte pixel_ch,
iBlckSendByte data_ch, iBlckFlagSend end_file_tc)
{

cSyncBlockchdata_sw, cddata_sw;
cSyncIntceobmp;
cSyncBlockchdata_hw1, cddata_hw1;
cSyncBlockchdata_hw2, cddata_hw2;
cSyncIntceobmp_2;
cFlagInt valid_hw_1, valid_hw_2;
cFlagInt valid_result_1, valid_result_2;
cFlagInt ch_sw, cd_sw, ch_hw1, cd_hw1, ch_hw2, cd_hw2;
event end_SW;


SWsw(header_ch, pixel_ch, data_ch,
chdata_sw, ceobmp, cddata_sw, ceobmp_2, valid_hw_1, valid_result_1, valid_hw_2, valid_result_2, ch_sw, cd_sw, end_SW);
HWhw_1(chdata_hw1, ceobmp, cddata_hw1, valid_hw_1, valid_result_1, ch_hw1, cd_hw1, end_SW);
HWhw_2(chdata_hw2, ceobmp_2, cddata_hw2, valid_hw_2, valid_result_2, ch_hw2, cd_hw2, end_SW);

memmemory(chdata_sw, cddata_sw, chdata_hw1, cddata_hw1,
chdata_hw2, cddata_hw2, ch_sw, cd_sw, ch_hw1, cd_hw1, ch_hw2, cd_hw2, end_SW);


void main(void) {
printf("********************\n");
printf("JPEG Encoder Begin...\n");
printf("********************\n");

par
{

sw.main();
hw_1.main();
hw_2.main();
memory.main();
}
end_file_tc.Valid_flag();
printf("******************\n");
printf("JPEG Encoder End...\n");
printf("******************\n");
}
};
```

F.2 sw.sc

```
/*******************************************
Project: system level architecture exploration
Stage: SW_HW_parallel mem model
Filename: sw.sc
Last change: 08/17/00
Auther: Lukai Cai
*******************************************/


import "handle";
import "quant";
import "huff";

#include "const.sh"


behavior BOHData(in int hdata[64], iBlckSendBlock CHData, iBlckFlagSend ch_sw)
{
void main ( void )
{
ch_sw.Valid_flag();
// send item hdata over channel
CHData.send ( hdata ) ;
waitfor(1);
}
};

behavior BOEOBmp(in int eobmp, iBlckSendInt CEOBmp)
{
void main ( void )
{
// send item eobmp over channel
CEOBmp.send ( eobmp ) ;
}
};

behavior BIDData(out int ddata[64], iBlckRecvBlock CDData, iBlckFlagSend cd_sw)
{
void main ( void )
{
cd_sw.Valid_flag();
// receive item ddata over channel
CDData.receive ( ddata ) ;
waitfor(1);


}
};

behavior SW(iBlckRecvInt header_ch, iBlckRecvByte pixel_ch,
iBlckSendByte data_ch,
iBlckSendBlock CHData,
iBlckSendInt CEOBmp,
iBlckRecvBlock CDData,
iBlckSendInt CEOBmp_2,
iBlckFlagRev valid_hw_1,
```

```
iBlckFlagRev valid_result_1,
iBlckFlagRev valid_hw_2,
iBlckFlagRev valid_result_2,
iBlckFlagSend ch_sw,
iBlckFlagSend cd_sw,
event end_SW)
{
inthdata[64];
intddata[64];
intqdata[64];
inteobmp;
intblock_no ; //for display
inti ;

HandleDatahandledata(header_ch, pixel_ch, data_ch,
hdata, eobmp);
Quantizationquantization(ddata, qdata);
HuffmanEncodehuffmanencode(qdata, data_ch);

BOEOBmpOEOBmp(eobmp, CEOBmp);
BOHDataOHData(hdata, CHData,ch_sw );
BIDDataIDData(ddata, CDData, cd_sw);

BOEOBmpOEOBmp_2(eobmp, CEOBmp_2);



  void main(void) {

int handle_count, valid_after_hw;
int end_file;
int a, b, c, d1, d2;

eobmp = 0 ;
block_no = 0 ;
handle_count=0;
valid_after_hw=0;
end_file=0;
a=0;
b=0;
c=0;
d1=0;
d2=0;


do
{



/*prepare for the hardware */
if((handle_count==0)&& (eobmp!=1)){
handledata.main();// original behavior
handle_count=1;
a++;
}
else if((valid_hw_1.Check_flag()==1)&&(b<a)){
valid_hw_1.Invalid_flag();
//send data from ColdFire to DCT
OEOBmp.main() ;// send eobmp ouput
OHData.main() ;// send hdata ouput
block_no ++ ;
handle_count--;
b++;
printf("Processing Block %dth in hw_1...\n", block_no);
}

else if((valid_hw_2.Check_flag()==1)&&(b<a)){
valid_hw_2.Invalid_flag();
//send data from ColdFire to DCT
OEOBmp_2.main() ;// send eobmp ouput
OHData.main() ;// send hdata ouput
block_no ++ ;
handle_count--;
b++;
printf("Processing Block %dth in hw_2...\n", block_no);
}



else
if((valid_result_1.Check_flag()==1)&&(d1+d2<a)&&(valid_afte
r_hw==0)){
valid_result_1.Invalid_flag();

//Receive data from DCT
IDData.main() ;// receive ddata ouput

valid_after_hw=1;
d1++;


}

else
if((valid_result_2.Check_flag()==1)&&(d1+d2<a)&&(valid_afte
r_hw==0)){
//Receive data from DCT
valid_result_2.Invalid_flag();
IDData.main() ;// receive ddata ouput
valid_after_hw=1;
d2++;


}

else if(valid_after_hw==1){
quantization.main();// original behavior
huffmanencode.main();// original behavior
valid_after_hw=0;
c++;

if(c==a){
end_file=1;
notify(end_SW);
}
```

```
    }
    else {

    waitfor(1);
    }


    }while(end_file!=1); // end of while


    WriteBits(-1, 0, data_ch);
    WriteMarker(M_EOI, data_ch);
    /* printf("\n a=%d, b=%d, c=%d, d1=%d, d2=%d", a, b,c, d1,
d2);
    printf("\n    run_time=%d,    wait_time=%d",    run_time,
wait_time); */


      }// end of main
    }; // end of behavior
```

F.3 hw.sc

```
/*******************************************
 Project: system level architecture exploration
  Stage: SW_HW_parallel mem model
  Filename: hw.sc
  Last change: 08/17/00
  Auther: Lukai Cai
*******************************************/

import "dct";

#include "const.sh"

behavior BIHData(out int hdata[64], iBlckRecvBlock CHData,
iBlckFlagSend ch_hw)
{
void main ( void )
{
ch_hw.Valid_flag();
CHData.receive ( hdata ) ;
waitfor(1);

}
};

behavior BIEOBmp(out int eobmp, iBlckRecvInt CEOBmp)
{
void main ( void )
{
eobmp = CEOBmp.receive() ;
}
};

behavior BODData(in int ddata[64], iBlckSendBlock CDData,
iBlckFlagSend cd_hw)
{
void main ( void )
{
cd_hw.Valid_flag();
CDData.send ( ddata ) ;
waitfor(1);
}
};

behavior HW_main(iBlckRecvBlock CHData,
iBlckRecvInt CEOBmp,
     iBlckSendBlock CDData,
iBlckFlagSend valid_hw,
iBlckFlagSend valid_result,
iBlckFlagSend ch_hw,
iBlckFlagSend cd_hw)
{
inthdata[64];
intddata[64];
inteobmp;

DCTdct(hdata, ddata);
BIHData IHData(hdata, CHData,ch_hw);
BIEOBmp IEOBmp(eobmp, CEOBmp);
BODData ODData(ddata, CDData, cd_hw);

   void main(void) {
valid_hw.Valid_flag();
valid_result.Invalid_flag();

do
{
//Receive data from ColdFire
IEOBmp.main();// receive eobmp output
IHData.main();// receive hdata output

dct.main();// original behavior



//send data from DCT to ColdFire
ODData.main();// send ddata output
valid_result.Valid_flag(); //tell sw can receive data
valid_hw.Valid_flag(); //tell sw can deal other imput

}while(eobmp!=1) ; // end of while

  } // end of main
}; // end of behavior


behavior IDLE(){
void main(){ }
};

behavior HW(iBlckRecvBlock CHData,
```

```
iBlckRecvInt CEOBmp,
      iBlckSendBlock CDData,
iBlckFlagSend valid_hw,
iBlckFlagSend valid_result,
iBlckFlagSend ch_hw,
iBlckFlagSend cd_hw,
event end_SW){

HW_main   HW_main_exec(CHData,   CEOBmp,   CDData,
valid_hw, valid_result,  ch_hw, cd_hw);
IDLE idle_exec();

void main(){
try{
HW_main_exec.main();}
trap(end_SW){idle_exec.main();}
}
};
```

F.4 memory.sc

```
/*******************************************
 Project: system level architecture exploration
  Stage: SW_HW_parallel mem model
  Filename: mem.sc
  Last change: 08/17/00
  Auther: Lukai Cai
*******************************************/

#include "const.sh"

import "global";
import "chann";

behavior MEM_main(iBlckRecvBlock chdata_sw,
   iBlckSendBlock cddata_sw,
   iBlckSendBlock chdata_hw1,
   iBlckRecvBlock cddata_hw1,
   iBlckSendBlock chdata_hw2,
   iBlckRecvBlock cddata_hw2,
   iBlckFlagRev ch_sw,
   iBlckFlagRev cd_sw,
   iBlckFlagRev ch_hw1,
   iBlckFlagRev cd_hw1,
   iBlckFlagRev ch_hw2,
   iBlckFlagRev cd_hw2){

void main(void){

inthdata[64];
intddata[64];
int hd_valid;
int dd_valid;
static int run_time;
static int a, b, c, d;

run_time=0;
a=0;
b=0;
c=0;
d=0;


ch_sw.Invalid_flag();
cd_sw.Invalid_flag();
ch_hw1.Invalid_flag();
cd_hw1.Invalid_flag();
ch_hw2.Invalid_flag();
cd_hw2.Invalid_flag();
hd_valid=0;
dd_valid=0;



while(1){

/* send first, then receive */
if((ch_sw.Check_flag()==1)&&(hd_valid==0)){
ch_sw.Invalid_flag();
chdata_sw.receive ( hdata ) ;
hd_valid=1;
/* a++;
printf("sw-->mem(%d)", a); */
}
else if(( cd_hw1.Check_flag()==1)&&(dd_valid==0)){
cd_hw1.Invalid_flag();
cddata_hw1.receive ( ddata ) ;
dd_valid=1;
/* b++;
printf("hw1-->mem(%d)", b); */
}
else if(( cd_hw2.Check_flag()==1)&&(dd_valid==0)){
cd_hw2.Invalid_flag();
cddata_hw2.receive ( ddata ) ;
dd_valid=1;
/* b++;
printf("hw2-->mem(%d)", b); */
}
else if((cd_sw.Check_flag()==1)&&(dd_valid==1)){
cd_sw.Invalid_flag();
cddata_sw.send ( ddata ) ;
dd_valid=0;
/* c++;
printf("mem-->sw(%d)", c); */

}
else if((ch_hw1.Check_flag()==1)&&(hd_valid==1)){
ch_hw1.Invalid_flag();
chdata_hw1.send ( hdata ) ;
hd_valid=0;
/* d++;
printf("mem-->hw1(%d)", d); */
}
else if((ch_hw2.Check_flag()==1)&&(hd_valid==1)){
ch_hw2.Invalid_flag();
chdata_hw2.send ( hdata ) ;
```

```
hd_valid=0;
/* d++;
printf("mem-->hw2(%d)", d); */
}
waitfor(1);
run_time=run_time+1;

}


printf("\n real run time is %d", run_time-1);
}
};

behavior IDLE2(){
void main(){ }
};

behavior mem(iBlckRecvBlock chdata_sw,
    iBlckSendBlock cddata_sw,
    iBlckSendBlock chdata_hw1,
    iBlckRecvBlock cddata_hw1,
    iBlckSendBlock chdata_hw2,
    iBlckRecvBlock cddata_hw2,
    iBlckFlagRev ch_sw,
    iBlckFlagRev cd_sw,
    iBlckFlagRev ch_hw1,
    iBlckFlagRev cd_hw1,
    iBlckFlagRev ch_hw2,
    iBlckFlagRev cd_hw2,
    event end_SW){

MEM_main mem_main_exec( chdata_sw, cddata_sw,
chdata_hw1, cddata_hw1, chdata_hw2, cddata_hw2,
ch_sw,cd_sw, ch_hw1, cd_hw1, ch_hw2, cd_hw2);
IDLE2 idle_exec();

void main(){
try{
mem_main_exec.main();}
trap(end_SW){idle_exec.main();}
}
};
```