Center for Embedded Computer Systems
University of California, Irvine

Design Exploration using Multiple ARM Instruction Set Simulators

- A Case Study on a JPEG Encoder

Kyoung Park Kim, Rainer Dömer

Technical Report CECS-09-08
May 28, 2009

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{kpkim, doemer}@uci.edu
http://www.cecs.uci.edu/

Design Exploration using Multiple ARM Instruction Set Simulators - A Case Study on a JPEG Encoder

Kyoung Park Kim, Rainer Dömer

Technical Report CECS-09-08
May 28, 2009

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{kpkim, doemer}@uci.edu
http://www.cecs.uci.edu

Abstract

Embedded software synthesis is an integral step in the embedded system design flow. To validate the generated software in the context of the entire system-on-chip, instruction set simulation is a critical task. In previous work [8], an instruction set simulator (ISS) for an ARM processor core has been integrated into the System-on-Chip Environment. Unfortunately, the existing integration is limited to a single processor unit in the entire system.

In this work, we extend the previous implementation so that multiple ARM ISS instances can be used concurrently within the system model. As a result, platform architectures with multiple ARM CPUs can now be accurately simulated at ISS level. This report demonstrates our multi-ARM ISS support by use of a case study on a JPEG encoder application.

Contents

List of Figures

List of Tables

Design Exploration using Multiple ARM Instruction Set Simulators

- A Case Study on a JPEG Encoder

Kyoung Park Kim, Rainer Dömer

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

{kpkim, doemer}@uci.edu
http://www.cecs.uci.edu

Abstract

Embedded software synthesis is an integral step in the embedded system design flow. To validate the generated software in the context of the entire system-on-chip, instruction set simulation is a critical task. In previous work [8], an instruction set simulator (ISS) for an ARM processor core has been integrated into the System-on-Chip Environment. Unfortunately, the existing integration is limited to a single processor unit in the entire system.

In this work, we extend the previous implementation so that multiple ARM ISS instances can be used concurrently within the system model. As a result, platform architectures with multiple ARM CPUs can now be accurately simulated at ISS level. This report demonstrates our multi-ARM ISS support by use of a case study on a JPEG encoder application.

1 Introduction

Hardware/software co-design is a set of methodologies and techniques specifically created to support the concurrent design of both systems, reducing the development time. Time to market is very important in chip business. There are so many companies to try to survive in SOC industry. The first company to release the chip to market takes most of profits. In addition to its critical role in the development of embedded systems, many experts believe that co-design will be a core

design methodology for Systems-on-a-Chip. Also, concurrent design, or co-design of hardware and software is extremely important for meeting design goals, such as high performance, that are the key to commercial competitiveness because designers can trade-off in the way hardware and software components work together to exhibit a specified behavior.

1.1 Top-Down System Design with SCE

In this project, top-down design methodology using SpecC[5][6][7] will be dealt with. Software should be developed in the early stage of chip design because software development time for embedded system takes more time than hardware development time. In Figure 1, top-down design methodology is briefly described by using SCE. It is composed of 6 steps. It starts from product specification. The specification model is generated from product specification. This specification model is untimed and has only the functional description of the design. Architecture refinement transforms this specification to an architecture model. It involves partitioning the design and mapping the partitions onto the selected components. The architecture model thus reflects the intended architecture for the design. The next scheduling refinement steps add RTOS to architecture model. Dynamic scheduling like Round-robin and priority-based scheduling and static scheduling are available for scheduling for each behavior in Spec-C model. Networking refinement is performed by adding buses to DUT and mapping all components under DUT to slaves and masters for buses. Communication refinement generates a timing accurate BFM. The final step is HW/SW synthesis which produces clock cycle accurate RTL model for hardware components and instruction specific assembly code for processors.

1.2 Multiple Instruction Set Simulator

This report contributes the support of multiple ARM instruction set simulators to the SCE design flow[2]. Previously there is a limitation in SCE to support at most one ARM instruction set simulator due to some global variables used in SWARM[3]. Now System-on Chip using multiple ARM7TDMI processors can be simulated with our extension of the SWARM. The multiple ARM simulation could be achieved by making those global variables local. The multiple ARM instruction set simulator is being developed, based on SpecC ARM ISS[8] currently integrated in SCE. More detailed information on this will be available later in this report.

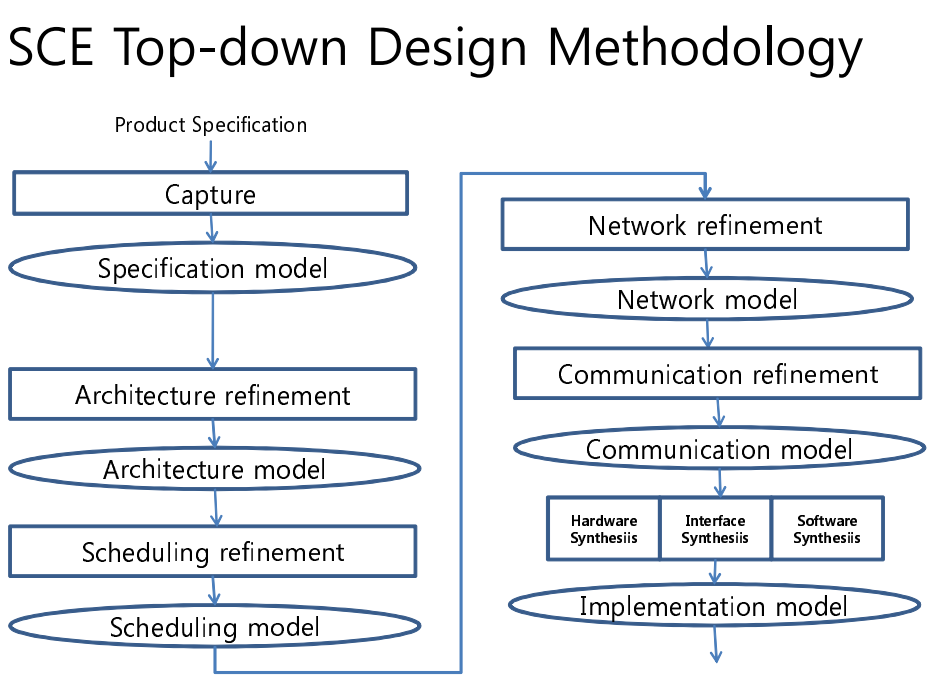

Figure 1: Embedded system design flow using SCE

2 Exploration of JPEG Encoder Application

To get through SCE top down design methodology, we start from JPEG encoder application[9]. Hardware synthesis is not covered at this time. Software synthesis is going to be dealt with to achieve the performance goals for JPEG encoder. In computing, JPEG is a commonly used method of compression for photographic images. The degree of compression can be adjusted, allowing a selectable tradeoff between storage size and image quality. JPEG typically achieves 10:1 compression with little perceptible loss in image quality.To be familiar with JPEG Encoder application, the structure of JPEG encoder is investigated. In Figure 2, the block diagram for JPEG encoder application[1] is shown. It consists of 8 blocks. `chendct1`, `chendct2`, `quantize`, `zigzag` and `huffencode` are the main parts of JPEG encoder. `Readbmpheader`, `InitGlobals` and `ReadBmpBlock` exist to generate inputs and initialize. Above JPEG encoder C model will be converted to Spec C model to do software synthesis. The interesting part of this JPEG application is `chendct1` and `chendct2` to support multi-processing. `chendct1` is covering odd block and `chendct2` is executed on even block.

2.1 JPEG Specification

To specify jpegencoder application for system design using SCE[4], we went through many steps. In version 0, JPEG encoder can be compiled through SpecC compiler. In version 1, `chendct1`, `chendct2`, `quantize`, `zigzag` and `huffencode` are integrated under DUT. And other blocks

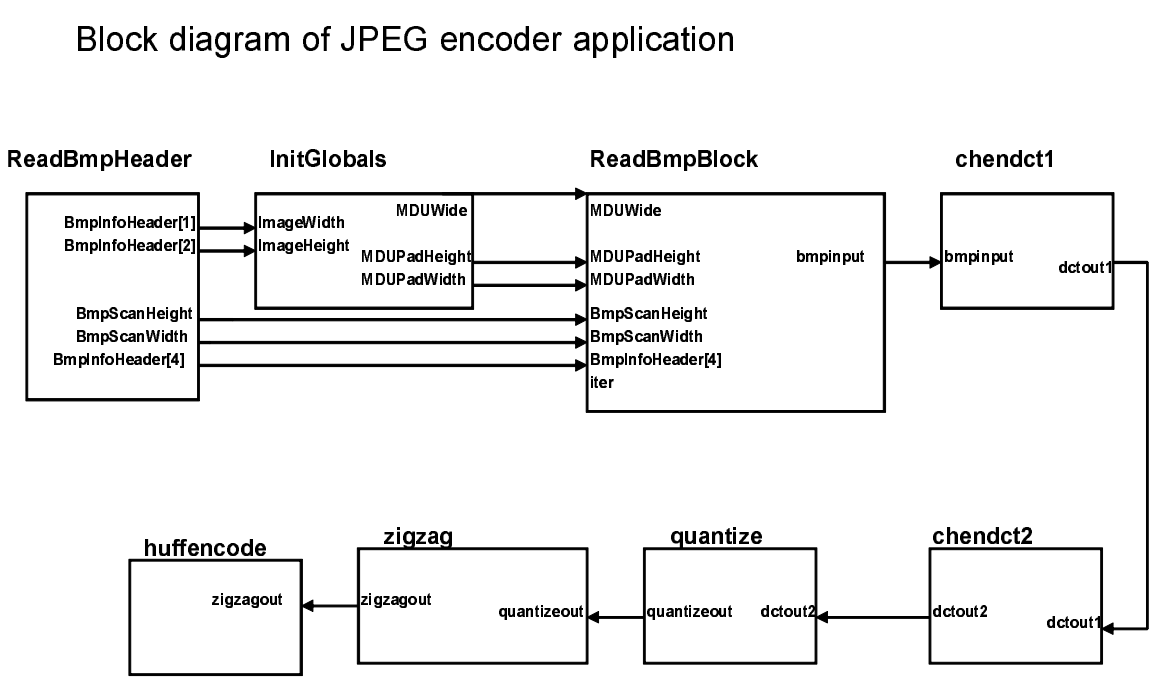

Figure 2: Block diagram of JPEG encoder application[1]

are divided into stimulus and monitor as it is seen in Figure 3. In version 1.1, printing statement for timing is added. In version 2.0, double_handshake channels are added to SpecC model to support parallel execution. In version 2.1, typed queue channel are introduced to support pipelined execution. This step was the most tricky part in this project. A lot of efforts are made to make it working[4].

2.2 JPEG Exploration

JPEG Encoder is created by improving version 2.1. It has zero warnings and clean hierarchy shown in Figure 4. There are no global variables and no global functions. They are merged into behaviors. It shows detailed timings for each encoded block. The reference picture, `test.jpg`, is moved into Monitor. For our convenience, this model is called `perfect model`.

More timing analysis is performed with newly created JPEG Encoder SpecC model. Computation profile for each block is shown on Figure 5. Timing for `chendct1` and `chendct2` are all same as 10.41 ms. `quantize` is consuming 7.84 ms of 180 block encoding time. `zigzag` is spending 2.32 ms of CPU time. `huffman` does use 8.88 ms. So total encoding time for 180 block is 39.86 ms. `chendct1` and `chendct2` are spending half of CPU time. And `huffman` is third. `quantize` is fourth. `zigzag` is the last.

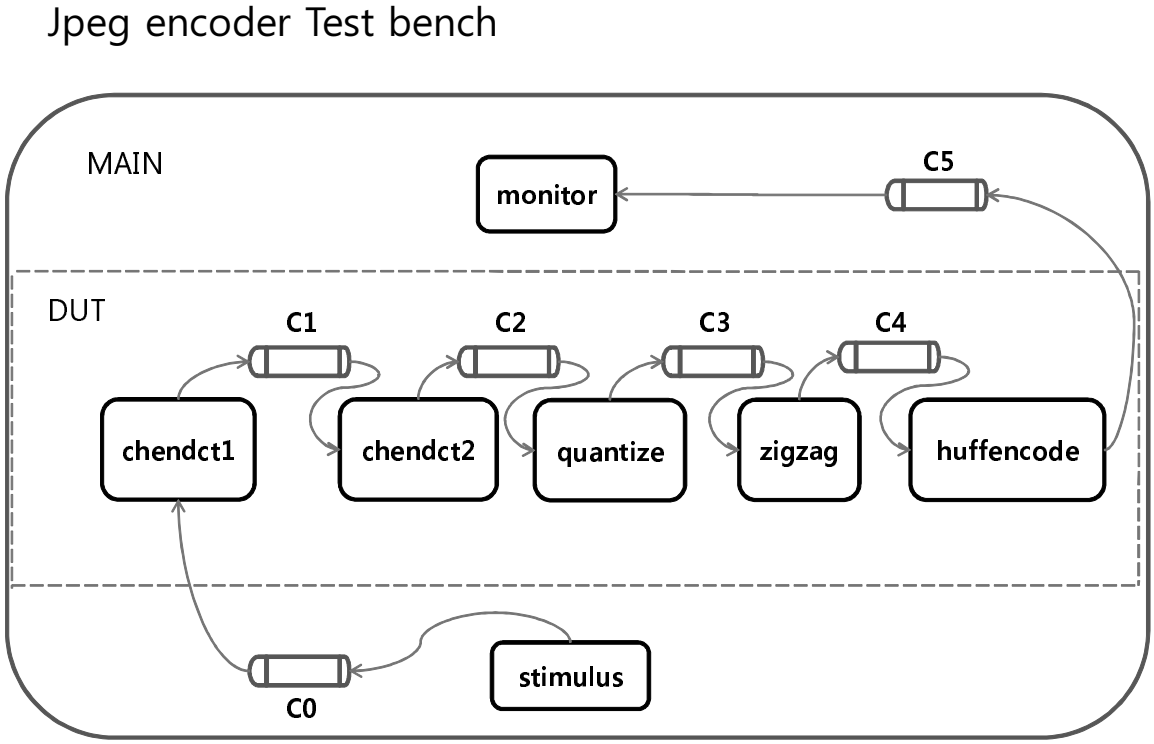

Figure 3: JPEG encoder test bench

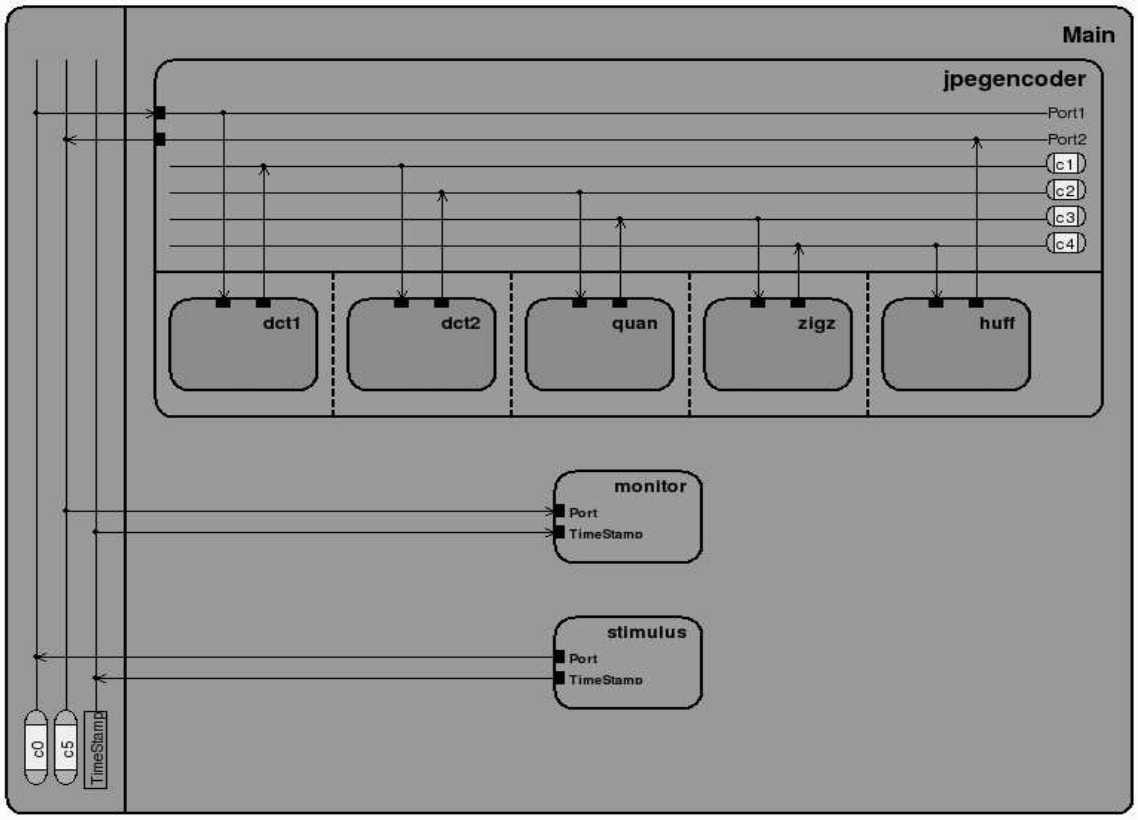

Figure 4: Exploration model of JPEG encoder in SCE

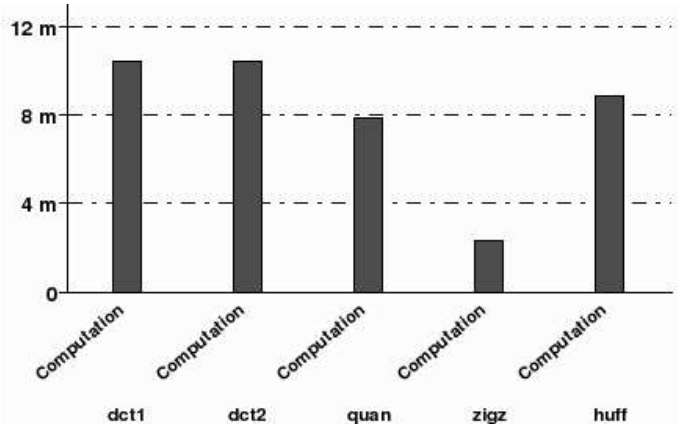

Figure 5: Computation profile of JPEG encoding

From the computation profile, `dct1`, `dct2` and `huff` are consuming most part of computation time. They are possible candidates for improving performance.

2.3 JPEG Design Analysis

To do architecture refinement, timing calculated from perfect model is back annotated. Estimated computation delays are inserted manually. Timing delays are added by using waitfor statement. The timings from perfect model are divided by 180 which means the number of input stimulus. That timing is added to each block.

Architecture refinements are performed and various system architecture is tested to get best architecture. From many tests and experiments, results show that performance is increased according to the increase of number of CPUs. But there is some saturation of performance at more than 3 CPUs. So 3 CPUs are the optimal number in terms of cost and performance. Table 1 shows some test results depending on number of CPUs and scheduling method of RTOS. Also blocks in models are randomly allocated to each CPU. Some RTOS is doing scheduling statically, and others are scheduling dynamically. Eventually, a graph for cost and speed trade off is drawn in Figure 6.

In reality, our JPEG encoder is still too slow to be used. It takes about 40 ms for encoding a 116X96 pixel image in black and white. 116X96 pixel is only 0.011136 mega-pixels. It needs about 1000 times performance improvements to cover 11.1 mega pixels.

Number of CPU	DCT1	DCT2	Quantize	Zigzag	Huffman	RTOS Scheduling	Execution time(us)
1	CPU1	CPU1	CPU1	CPU1	CPU1	CPU1:Priority	39860
2	CPU1	CPU1	CPU2	CPU2	CPU2	CPU1:No OS CPU2:Priority	19153
3	CPU1	CPU1	CPU2	CPU3	CPU3	CPU1:No OS CPU2:No OS CPU3:Round Robin	11358
4	CPU1	CPU2	CPU3	CPU4	CPU4	CPU1:No OS CPU2:No OS CPU3:No OS CPU4:Round Robin	11358
5	CPU1	CPU2	CPU3	CPU4	CPU5	CPU1:Round Robin CPU2:Round Robin CPU3:Round Robin CPU4:Round Robin CPU5:Round Robin	10574

Table 1: Architecture refinement from perfect model

2.4 Single CPU Implementation

To support bus functional model for CPU, platform model for JPEG encoder is created in Figure 8. Communications in jpegencoder can be refined to actual CPU bus. IO units datain and dataout are added to platform model to support BFM communication via CPU bus. Channels from c1 to c4 are going to be real system AMBA bus. They are using totally different protocols compared to c1 and c5 which means channels for c1 and c2 are TLM bus model and more abstract bus model than BFM bus model. So there should be some kind of wrapper to convert channel to real hardware bus. IO_Unit1 is inserted into JPEG platform and mapped to datain to transfer the inputs from stimulus to jpegencoder. Also IO_Unit2 is added to JPEG Platform and mapped to dataout to send output of JPE encoder to monitor which compares the results to check if those results are matched to the golden data of `test.jpg`.

Network refinement is performed to map all of PEs under JPEG Plaform to slaves and masters of buses.Bus0 are renamed to "CPU_BUS". Double handshake bus is added as HW_Bus. Additional Port1 is created for HW. Port1 is eventually connected to stimulus through IO_Unit1. H2 is connected to CPU_Bus as slave4 through port1. Hierarchy chart of BFM is drawn in Figure 7.

Communication refinement is performed to generate timing accurate BFM model of JPEG encoder. Addresses are assigned to HW and IO_Unit2 for memory mapped IO. C codes for execution for ARM7TDMI are generated from communication model.

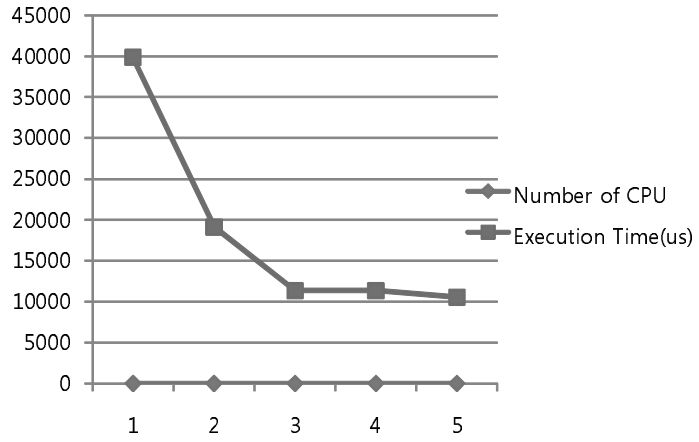

Figure 6: Cost/Speed trade off graph

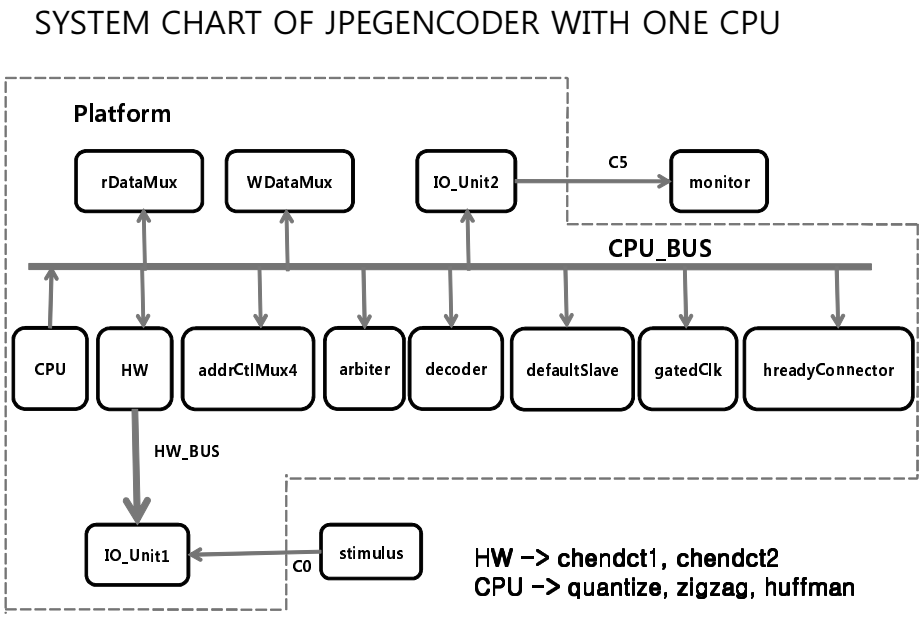

Figure 7: Hierarchy chart of bus functional model

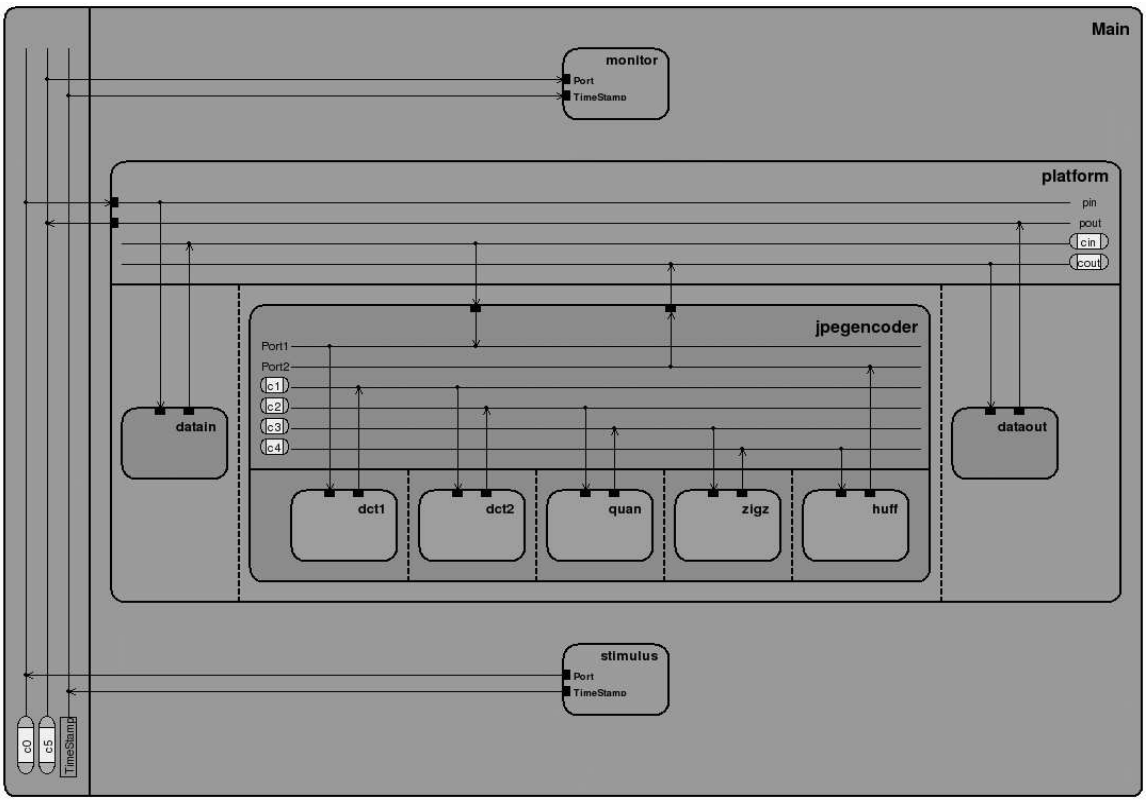

Figure 8: Platform model for JPEG encoder

To run C codes generated from communication model, instruction set simulator model is created. System architecture consists of 4 processing elements. Those PEs are including main ARM CPU for quantize, zigzag, and Huffman whose using a priority based scheduling using microC-OS2 RTOS, a hardware accellarator for the DCTs, 2 I/O units for data in and output. Those PEs are connected through AMBA-AHB bus which is the built-in CPU bus and a custom double-handshake bus. But some problems exists when C code generator inserts unwanted TaskDelay(). Bus Functional models get stuck after encoding for block 177. Encoding is still too slow. It takes 142 milliseconds for 177 blocks. Our goal is to achieve 0.013 ms to support 1.1 mega pixels for color photograph. The target speed is far away from the result regardless of some limitation. The detailed steps in SCE for this single CPU implementation are listed in `Appendix A.1`.

3 Multi-ARM Instruction Set Simulation

We will now describe out work in extending the SWARM ISS in SCE to support multiple ARM ISSs instances.

3.1 Problem definition

The reason why the original ARM instruction simulation does not support the multiple instances is that there are nine global variables to be referenced in some local and global functions in instruction simulator. When JPEG simulation is run with multiple ARM ISSs, those extern variables should not be shared and commonly used in multiple ARM ISSs. Eventually those extern variables cause simulation to stop and prevent multiple ARM ISSs from running concurrently. To instance the multiple ARM ISSs properly, all global functions should not touch global variables and classes. Moreover, global variables and classes should be localized to support multiple ARM ISSs. All extern variables that need to be localized are shown in Figure 9 as an example to show what variables are directly used in the source code of ARM ISS.

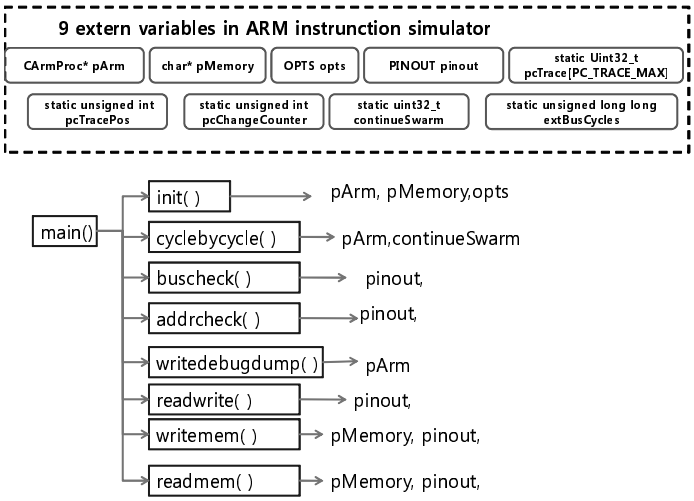

Figure 9: Extern variables and classes used in funtions in SWARM ISS

3.2 Approach and Solution

Also some global functions to use those external variables are described. To protect the functions in the source code of ISS from referencing the global variables directly, they are passed to the functions as one of the parameters. `CArmProc pArm` is first selected to be localized because it is most frequently used in ARM ISS. SpecC is ANSI-C based system level description language and does not recognize class object. So to make the extern variables localized, structure object will be used instead of class object. All extern variables are packed into one structure called `SArmProc`.

CArmProc pArm is hidden as an member of struct SArmProc to make the SpecC wrapper of ARM ISS compiled.

struct SarmProc is declared as an empty struct in swarm_sim.h which means that it does not have any members in it because ANSI-C compiler does not understand the class declaration and creates compiler error. So the real struct SArmProc is defined again in swarm_sim.cpp like below.

```
struct SArmProc
{
CArmProc* pArm;
};
```

And then it is declared as struct SArmProc* pArm. To hide all of detailed information related to Class, pArm will be declared as a pointer of SArmProc. So &pArm is passed to init function like result = init(argc,argv,&pArm); And then all of objects like pArm and pMemory necessary to run simulation will be created in init() function.

Now most of function calls in SWARM source code have CArmProc* pArm as a function parameter. While modifying source code, atexit() is updated different from the way to modify other functions. atexit(PrintISSCycles) is executed when simulation ended. atexit() has function pointer as a parameter which points to the function. That function should have void pointer as a parameter. But PrintISSCycles does have pArm as a parameter due to localization of pArm. Now PrintISSCylces() is inserted into class CArmProc as the member function of class CArmProc.

3.3 Implementation

All extern variables are removed from the original source code for ARM instruction simulator one by one. Finally all of extern variables are merged into structure like below.

```
struct SArmProc
{
CArmProc* pArm;
char* pMemory;
OPTS opts;
PINOUT pinout;
uint32_t pcTrace[PC_TRACE_MAX];
unsigned int pcTracePos;
unsigned int pcChangeCounter;
uint32_t continueSwarm;
};
```

To verify all updates for new ARM instruction simulator, JPEG simulation with one ARM ISS is run and the results match the original ARM ISS.

3.4 Installation

Source codes for multiple SWARM instruction set simulators and SpecC wrappers for multiple SWARM ISSs are imported into CVS repository under `/home/lecs/cvs/multi_swarm`. They are also checked out under `/home/lecs/chkout/multi_swarm`. `multi_swarm` directory consists of two directories. One is `arm7tdmiwrapper` to include SpecC wrapper for multiple SWARM ISS and the other is `swarm` which contains source codes for multiple SWARM ISSs. Actually `src` directory under `swarm` contains all source codes for multiple SWARM ISSs.

4 Multiple CPU Implementation

We will now describe two implementations of the JPEG encoder with multiple ARM CPUs.

4.1 JPEG Architecture using 2 ARM CPUs

To run JPEG simulation with 2 CPUs, new architecture platform with 2 CPUs needs to be devised.

4.1.1 Communication via Hardware Block

`HW2` is used as an intermediary processor element to connect `CPU1_BUS1` and `CPU2_BUS` each other because there is no bridge currently available in SCE library. `chendct1` and `chendct2` are assigned to `HW1` like JPEG platform with one CPU. `CPU1` handles `quantize` behavior. And `HW2` is taking care of `zigzag`. `CPU2` performs `huffman`. The detailed JPEG platform with 2 CPUs is shown in Figure 10. The detailed steps in SCE for this implementations are listed in `Appendix A.2`.

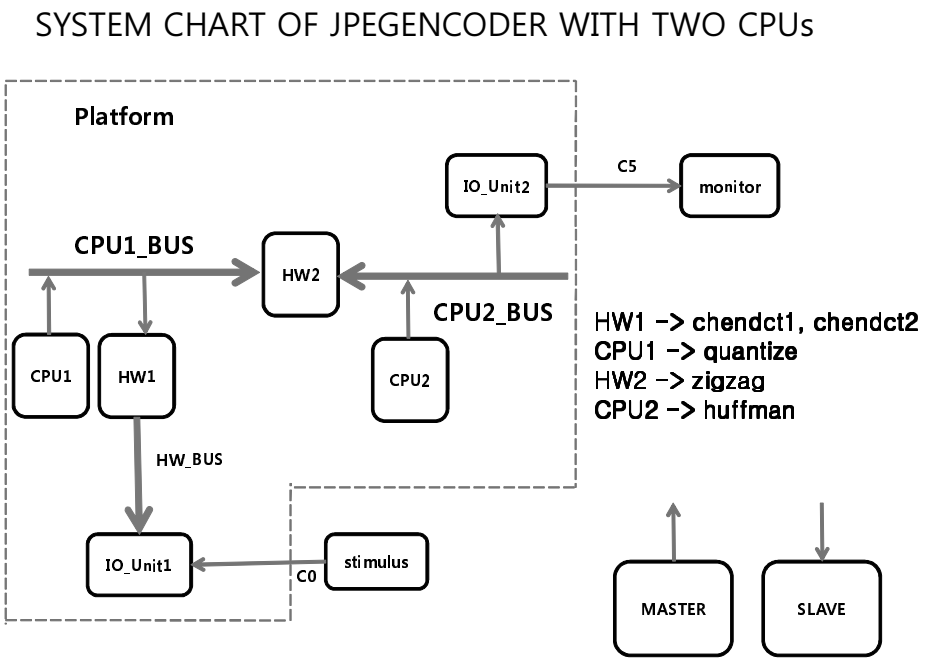

Figure 10: System architecture for JPEG encoder using 2 CPUs

4.1.2 Communication via Bridge

If the bridge HW is available, JPEG platform will be drawn like in Figure 11. `HW2` is replaced with bridge HW and then `HW2` is now the slave of `CPU2_BUS`. There are some advantages in this JPEG platform because `CPU1` can access `CPU2_BUS` through bridge. When bridge is used to connect the buses each other, it is safe to disallow the bidirectional access because it might cause the deadlock and then the whole system might get stuck. The JPEG platform with bridge allows more flexibility in the architecture of Platform than the architecture without bridge. Also it will be better to support the multiple masters in each bus because the bridge hardware usually is the slave of one bus and also it is the master of the other bus. Eventually there exist two masters on the bus. Also it is very important for CPU to be able to access all of slaves on the platform due to the memory mapped I/0 to control slaves. In Figure 11, bridge is the slave of `CPU1_BUS` and it is also the master of `CPU2_BUS`.

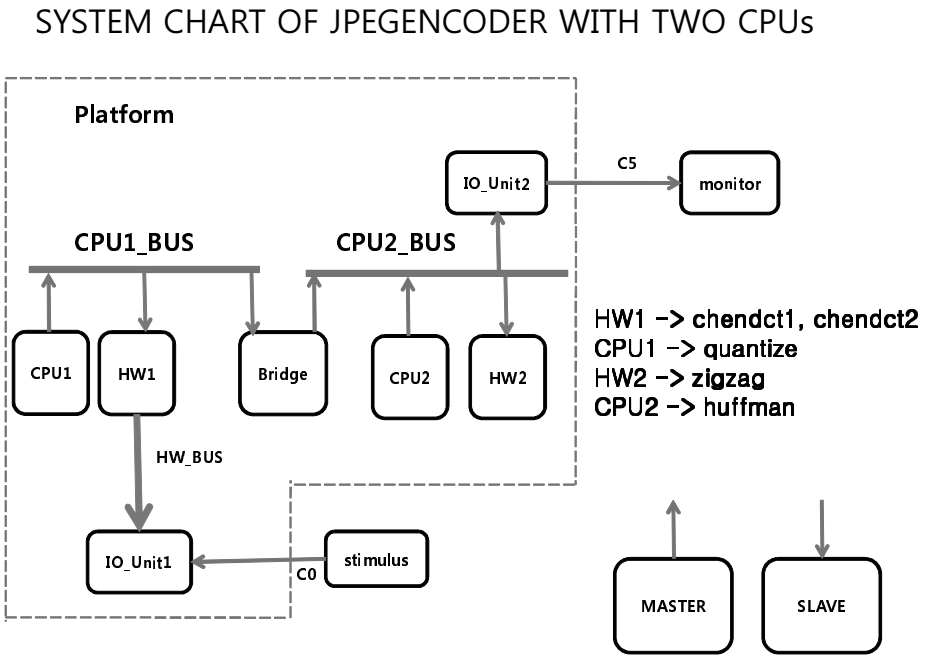

Figure 11: System architecture for JPEG encoder using 2 CPUs with bridge

4.2 JPEG Architecture using 3 ARM CPUs

4.2.1 Communication via Hardware Block

To run JPEG simulation with 3 CPUs, new architecture platform with 3 CPUs needs to be devised. `HW1` still needs to be used as intermediary processor elements to connect `CPU_BUS1` and `CPU_BUS2`

together because there is no bridge currently available in SCE library. HW2 is used to connect CPU_BUS2 and CPU_BUS3. chendct1 is assigned to CPU1. HW1 handles chendct2 different for JPEG platform with one CPU and 2 CPUs. And CPU2 is taking care of quantize. HW2 executes zigzag. CPU3 performs huffman. The detailed JPEG platform with 3 CPUs is shown in Figure 12.

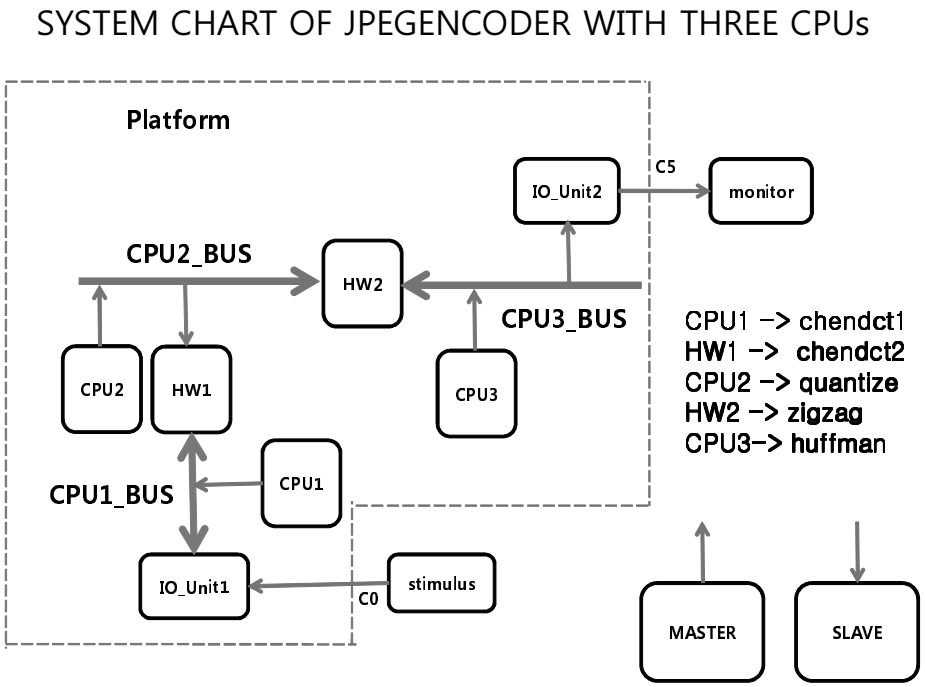

Figure 12: System architecture for JPEG encoder using 3 CPUs

4.2.2 Communication via Bridge

If the bridge HW is available, JPEG platform will be drawn like in Figure 13. HW1 is replaced with bridge HW and then HW1 is now the slave of CPU2_BUS.

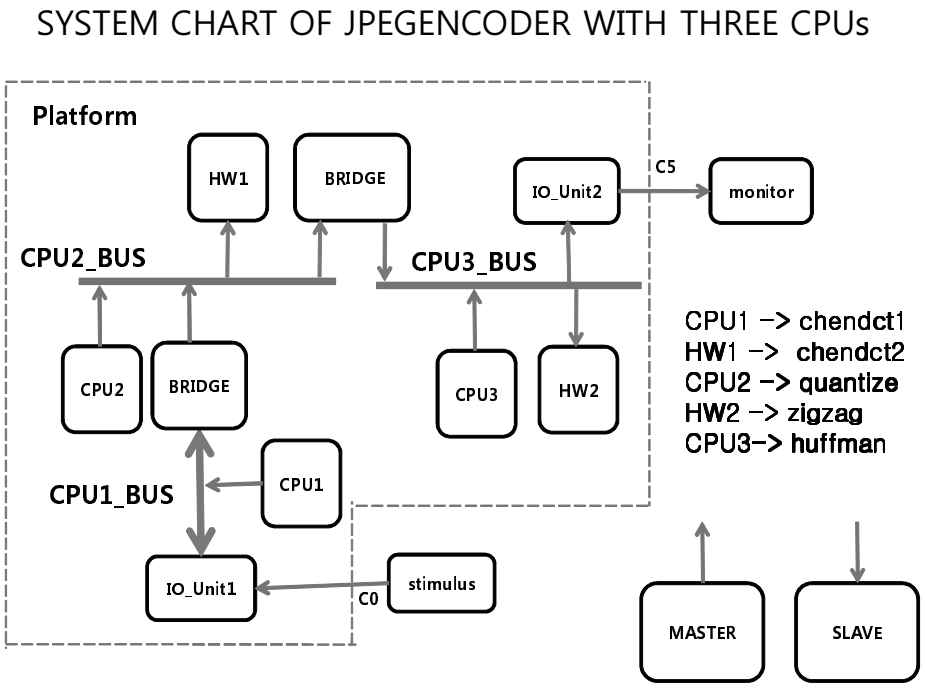

Figure 13: System architecture for JPEG encoder using 3 CPUs with bridge

5 Experimental Results

5.1 Platform Execution Times

JPEG encoder software synthesis is explored. It starts from JPEG encoder application. To obtain our performance goal, many architecture variations are analyzed and revealed. If there is a bridge available in SCE library, JPEG platform having more than 3 ISS will be able to be tested. Due to the current limitation of SCE library, the results for simulation could be obtained up to 3 ARM ISS. The results are shown in Table 2. This result is reasonable to take more time in 3 CPUs than 2 CPUs because DCT1 is processed in ARM ISS in 3 CPU case differently from 2 CPU case. Usually hardware performs faster than software.

Number of CPU	PE Mapping	Execution time(us)
1	DCT1,DCT2 $\mapsto$ HW Quantize,Zigzag,Huffman $\mapsto$ CPU	142043
2	DCT1,DCT2 $\mapsto$ HW1 Quantize $\mapsto$ CPU1 Zigzag $\mapsto$ HW2 Huffman $\mapsto$ CPU2	92979
3	DCT1 $\mapsto$ CPU1 DCT2 $\mapsto$ HW1 Quantize $\mapsto$ CPU2 Zigzag $\mapsto$ HW2 Huffman $\mapsto$ CPU3	93467

Table 2: Execution time depending on the number of CPUs

5.2 Multi-ARM Simulation Times

Simulation times are obtained by using `/usr/bin/time` command. Simulation times are shown in Table 3 according to the number of CPU. As assumed, simulation time is increasing proportional to the number of CPU.

	Simulation Time		
Number of CPU	User(sec)	System(sec)	Total(User+System)
1	554.59	91.06	651.96
2	837.70	164.33	1002.33
3	1250.63	232.01	1482.62

Table 3: Simulation time depending on the number of ARM ISS

6 Conclusions and Future Work

JPEG encoder software synthesis is explored. It starts from JPEG encoder application. SWARM ISS library in SCE has been successfully patched such that multiple ARM ISS instances of the simulator can run independently within the same platform model. Our JPEG example runs now fine with 1, 2, or 3 ARM CPUs.

Now SCE does not support the multiple C code generation concurrently. So C codes for multiple ARM ISS need to be generated separately until the next SCE is released. Regarding to HW IPs, if the bridge HW is available, JPEG platform might be able to have better architecture than now. When bridge is used to connect the buses each other, it is safe to disallow the bidirectional access because it might cause the deadlock and then the whole system might get stuck. The JPEG platform with bridge allows more flexibility in the architecture of Platform than the architecture without bridge. Due to unavailability of bridge, JPEG ISS model having more than 3 ARM ISSs is impossible to test. Also it will be better to support the multiple masters in each bus because the bridge hardware usually is the slave of one bus and also it is the master of the other bus. Eventually there exist two masters on the bus. Also it is very important for CPU to be able to access all of slaves on the platform due to the memory mapped I/0 to control slaves.

References

[1] Samar Abdi. *JPEG Encoder Software Synthesis,Talks in Class*, Fall 2008.

[2] Samar Abdi, Junyu Peng, Haobo Yu, Dongwan Shin, Andreas Gerstlauer, Rainer Dömer, and Daniel Gajski. *System-on-Chip Environment (SCE Version 2.2.0 Beta): Tutorial*. Technical Report CECS-TR-03-41, Center for Embedded Computer Systems, University of California, Irvine, July 2003.

[3] Michael Dales. *SWARM, ARM instruction simulator,GNU software* , 2000.

[4] Rainer Dömer. *SOC Software Synthesis, Lecture Notes*, Fall 2008.

[5] Rainer Dömer, Andreas Gerstlauer, and Daniel Gajski. *SpecC Language Reference Manual, Version 2.0*. SpecC Technology Open Consortium, http://www.specc.org, December 2002.

[6] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[7] Andreas Gerstlauer, Rainer Dömer, Junyu Peng, and Daniel D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[8] G.Schirner, G.Sachdeva, A. Gerstlauer, and R. Dömer. *Modeling, Simulation and Synthesis in an Embedded Software Design Flow for an ARM Processor*. Technical Report CECS-TR-06-06, Center for Embedded Computer Systems, University of California, Irvine, May 2006.

[9] International Telecommunication Union (ITU). *Digital Compression and Coding of Continous-Tone Still Images*, September 1992. ITU Recommendation T.81.

A Appendix

A.1 SCE Design Steps for Single-CPU Implementation

```
Here are some detailed instructions to run JPEG simulation with one ARM ISS.
* Specification step
- new project
- import JpegPlatform.sc
- add to project as "PlatformSpec.sir"
- view the new hierarchy chart (entirely, incl. connectivity)
- compile and simulate

* Architecture Refinement step
- choose "platform" as top-level
- allocate two ARM_7TDMI as "CPU"
- allocate two HW_Standard as "HW"
- allocate two HW_Virtual as "IO_Unit1" and "IO_Unit2"
- map datain to IO_Unit1
- map dataout to IO_Unit2
- map cin to IO_Unit1
- map cout to IO_Unit2
- map dct1 and dct2 to HW
- map quantize,zigzag and huffman to CPU
- perform architecture refinement (no timing back annotation)
- rename generated model as "PlatformArch"
- compile and simulate

* Scheduling Refinement
- use priority-based scheduling for CPU
  (priorities 1, 2, 3 for quan, zigz, huff, respectively)
- leave IO_Units alone
- leave HW alone
- perform scheduling refinement (both static and dynamic)
- rename generated model as "PlatformSched"
- compile and simulate

* Network Refinement
- rename "Bus0" to "CPU_Bus"
- add DblHndShkBus as "HW_Bus"
- create additional "Port1" for HW
- connect HW (Port0) to CPU_Bus as "slave4"
- connect HW (Port1) to HW_Bus as "Master"
- connect IO_Unit2 (Port0) to CPU_Bus as "slave5"
- connect IO_Unit1 (Port0) to HW_Bus as "Slave"
- perform network refinement
- rename generated model as "PlatformNet"
- compile and simulate

* Communication Refinement
- press "CPU_Bus"tab
- select start address 0x50000000 for c_link_CPU__IO_Unit2
- select start address 0x40000000 for c_link_HW__CPU
- press "Bus0"tab
- select start address 0x0000 for c_link_IO_Unit1__HW
- perform communication refinement (pin-accurate model)
- rename generated model as "PlatformComm"
- compile and simulate
- Simulation gets stuck after monitor receives block 177.
- press CTRL-C to stop simulation
```

```
* Code generation for CPU
- close PlatformComm.sir
- sir_note PlatformComm ARM_7TDMI_Core_20000_0_CPU '_PE_HAL_MODEL="
    ARM_7TDMI_HAL_20000_0_CPU"'
- re-open PlatformComm.sir
- perform code generation for CPU1 (for ARM_7TDMI_OS_20000_0_CPU1_NET)
  (store in files "CPU/CPU.c" and "CPU/CPU.h")
- rename generated model as "PlatformCommC"
- compile and simulate
- Simulation gets stuck after monitor receives block 177.
- press CTRL-C to stop simulation

* Cross-compilation for CPU
- we will cross-compile the generated C code in the CPU directory
  cd CPU
- inspect the generated ANSI-C code in files "CPU.c" and "CPU.h"
  It appears that the generated C code converts our back-annotated timing
estimates into TaskDelay statements that suspend a task for the given
period of time. Thus, the software tasks are actually put to sleep for
our back-annotated waitfor() statements!!

After the C code generation step, take a look at the file CPU1.h.
At the top of the file, you will find the following definition:
#define WAITFOR(X) TaskDelay((unsigned long)((X)/1000))
Please change this to the following:
#define WAITFOR(X) // nothing!
This will make the bogus waiting disappear. This changes should be done again in CPU2.h
- to compile the generated code together with the microC-OS-II
  (and some other files), we'll use a prepared Makefile
- update Makefile for reference path for SWARM ISS and user source code like below.
    # reference to SWARM ISS
    #ISS_DIR = /opt/pkg/sw/swarm
     ISS_DIR = /home/lecs/chkout/project/multi_swarm/swarm

   # USER specific source file
      USR_SRC     := CPU.c
      USR_HDR     := CPU.h
- type "make" after modification of Makefile
- the generated ARM-executable is found in file "userCode"
  ls
- copy the generated executable userCode into your SCE working directory
  (so that the SCE simulation can find it)
  cp userCode ..

* Insertion of ISS model for CPU
- select Comm.sir (in SCE Project window)
- Edit->ImportDesign
  "/home/lecs/chkout/project/multi_swarm/arm7tdmiwrapper/arm7tdmiiss.sir"
  (this will show up in the list of unused behaviors in the working window
  as behavior "ARM_7TDMI_ISS")
- locate the instance "CPU" of behavior "ARM_7TDMI_Core_20000_0_CPU"
  in the hierarchy browser
- replace this abstract model with the ISS model
(right-click "ARM_7TDMI_Core_20000_0_CPU", ChangeType to "ARM_7TDMI_ISS")
- go to Project menu -> Settings
- change Import path to ".:/home/lecs/chkout/project/multi_swarm/arm7tdmiwrapper"
- change Library path to "/home/lecs/chkout/project/multi_swarm/swarm"
- set Verbosity level and Warning level to 2 .
- save this design model (as a new model in the project)
```

```
File->SaveAs "PlatformISS.sir" (in your working directory!)
- compile and simulate.
- Simulation gets stuck after monitor receives block 177.
  139632: Monitor received block 174 (with 3 bytes).
  139632: Encoding for block 174 took   21154 micro seconds.
  140444: Monitor received block 175 (with 15 bytes).
  140444: Encoding for block 175 took   21179 micro seconds.
  141251: Monitor received block 176 (with 9 bytes).
  141251: Encoding for block 176 took   21199 micro seconds.
  142043: Monitor received block 177 (with 6 bytes).
  142043: Encoding for block 177 took   21188 micro seconds.
```

A.2 SCE Design Steps for Two-CPU Implementation

```
Here are some detailed instructions to run JPEG simulation with 2 ARM ISSs which are quite
     similar to the ones with one CPU.
* Specification step
- new project
- import JpegPlatform.sc
- add to project as "PlatformSpec.sir"
- view the new hierarchy chart (entirely, incl. connectivity)
- compile and simulate

* Architecture Refinement step
- choose "platform" as top-level
- allocate two ARM_7TDMI as "CPU1" and "CPU2" respectively
- allocate two HW_Standard as "HW1" and "HW2"
- allocate two HW_Virtual as "IO_Unit1" and "IO_Unit2"
- map datain to IO_Unit1
- map dataout to IO_Unit2
- map cin to IO_Unit1
- map cout to IO_Unit2
- map dct1 and dct2 to HW1
- map quantize to CPU1
- map zigzag to HW2
- map Huffman to CPU2
- perform architecture refinement (no timing back annotation)
- rename generated model as "PlatformArch"
- compile and simulate

* Scheduling Refinement
- use priority-based scheduling for CPU1 and CPU2
  (no priorities needed for CPU1 and CPU2 because only one behavior assigned)
- leave IO_Units alone
- leave HW1 and HW2 alone
- perform scheduling refinement (both static and dynamic)
- rename generated model as "PlatformSched"
- compile and simulate

* Network Refinement
- rename "Bus0" to "CPU1_Bus"
- rename "Bus1" to "CPU2_Bus"
- add DblHndShkBus as "HW_Bus"
- create additional "Port1" for HW1
- create additional "Port1" for HW2
- connect HW1 (Port0) to CPU1_Bus as "slave4"
- connect HW1 (Port1) to HW_Bus as "Master"
- connect HW2 (Port0) to CPU1_Bus as "slave5"
- connect HW2 (Port1) to CPU2_Bus as "slave4"
- connect IO_Unit2 (Port0) to CPU2_Bus as "slave5"
```

```
- connect IO_Unit1 (Port0) to HW_Bus as "Slave"
- perform network refinement
- rename generated model as "PlatformNet"
- compile and simulate

* Communication Refinement
- press "CPU1_Bus"tab
- select start address 0x50000000 for c_link_CPU1__HW2
- select start address 0x40000000 for c_link_HW1__CPU1
- press "CPU2_Bus"tab
- select start address 0x50000000 for c_link_CPU2__IO_Unit2
- select start address 0x40000000 for c_link_HW2__CPU2
- press "Bus0"tab
- select start address 0x0000 for c_link_IO_Unit1__HW
- perform communication refinement (pin-accurate model)
- rename generated model as "PlatformComm"
- compile and simulate
- Simulation should be working fine. It does not get stuck different from one CPU case.

* Code generation for CPU1 and CPU2
Now SCE does not support the multiple C Code generation on the fly.
So C Codes for both CPU1 and CPU2 needs to be generated separately until the next version
    of SCE is released.
- close PlatformComm.sir
- sir_note PlatformComm ARM_7TDMI_Core_20000_0_CPU1 '_PE_HAL_MODEL="
    ARM_7TDMI_HAL_20000_0_CPU1"'
- sir_note PlatformComm ARM_7TDMI_Core_20000_0_CPU2 '_PE_HAL_MODEL="
    ARM_7TDMI_HAL_20000_0_CPU2"'
- re-open PlatformComm.sir
- perform code generation for CPU1 (for ARM_7TDMI_OS_20000_0_CPU1_NET)
  (store in files "CPU1/CPU1.c" and "CPU1/CPU1.h")
- rename generated model as "PlatformCommC1"
- compile and simulate
- perform code generation for CPU2 (for ARM_7TDMI_OS_20000_0_CPU2_NET)
  (store in files "CPU2/CPU2.c" and "CPU2/CPU2.h")
- rename generated model as "PlatformCommC2"
- compile and simulate
- Simulation should be working fine. It does not get stuck different from one CPU case.

* Cross-compilation for CPU1
Now SCE does not support the multiple ARM ISS. Only one ARM ISS simulation is allowed.
- we will cross-compile the generated C code in the CPU1 directory
  cd CPU1
- inspect the generated ANSI-C code in files "CPU1.c" and "CPU1.h"
  It appears that the generated C code converts our back-annotated timing
estimates into TaskDelay statements that suspend a task for the given
period of time. Thus, the software tasks are actually put to sleep for
our back-annotated waitfor() statements!!

After the C code generation step, take a look at the file CPU1.h.
At the top of the file, you will find the following definition:
#define WAITFOR(X) TaskDelay((unsigned long)((X)/1000))
Please change this to the following:
#define WAITFOR(X) // nothing!
This will make the bogus waiting disappear. This changes should be done again in CPU2.h
- to compile the generated code together with the microC-OS-II
  (and some other files), we'll use a prepared Makefile
- update Makefile for reference path for SWARM ISS and user source code like below.
    # reference to SWARM ISS
    #ISS_DIR = /opt/pkg/sw/swarm
```

```
      ISS_DIR = /home/lecs/chkout/project/multi_swarm/swarm

    # USER specific source file
       USR_SRC     := CPU1.c
       USR_HDR     := CPU1.h
- type "make" after modification of Makefile
- the generated ARM-executable is found in file "userCode"

* Cross-compilation for CPU2
Now SCE does not support the multiple ARM ISS. Only one ARM ISS simulation is allowed.

- we will cross-compile the generated C code in the CPU1 directory
  cd CPU2
- inspect the generated ANSI-C code in files "CPU2.c" and "CPU2.h"
  more CPU2.c
  more CPU2.h
- to compile the generated code together with the microC-OS-II
  (and some other files), we'll use a prepared Makefile
- update Makefile for reference path for SWARM ISS and user source code like below.
     # reference to SWARM ISS
     #ISS_DIR = /opt/pkg/sw/swarm
      ISS_DIR = /home/lecs/chkout/project/multi_swarm/swarm

    # USER specific source file
       USR_SRC     := CPU2.c
       USR_HDR     := CPU2.h
- type "make" after modification of Makefile
- the generated ARM-executable is found in file "userCode"
* Insertion of ISS model for CPU1 and CPU2
- select Comm.sir (in SCE Project window)
- Edit->ImportDesign
  "/home/lecs/chkout/project/multi_swarm/arm7tdmiwrapper/arm7tdmiiss1.sir"
  (this will show up in the list of unused behaviors in the working window
  as behavior "ARM_7TDMI_ISS1")
- Edit->ImportDesign
  "/home/lecs/chkout/project/multi_swarm/arm7tdmiwrapper/arm7tdmiiss2.sir"
  (this will show up in the list of unused behaviors in the working window
  as behavior "ARM_7TDMI_ISS2")
- locate the instance "CPU1" of behavior "ARM_7TDMI_Core_20000_0_CPU1"
  in the hierarchy browser
- replace this abstract model with the ISS model
(right-click "ARM_7TDMI_Core_20000_0_CPU1", ChangeType to "ARM_7TDMI_ISS1")
- locate the instance "CPU2" of behavior "ARM_7TDMI_Core_20000_0_CPU2"
  in the hierarchy browser
- replace this abstract model with the ISS model
(right-click "ARM_7TDMI_Core_20000_0_CPU2", ChangeType to "ARM_7TDMI_ISS2")
- go to Project menu -> Settings
- change Import path to ".:/home/lecs/chkout/project/multi_swarm/arm7tdmiwrapper"
- change Library path to "/home/lecs/chkout/project/multi_swarm/swarm"
- set Verbosity level and Warning level to 2.
- save this design model (as a new model in the project)
File->SaveAs "PlatformISS.sir" (in your working directory!)
- compile and simulate.
- Simulation should end like below.
        0: Stimulus sends block 0.
        0: Stimulus sends block 1.
        0: Stimulus sends block 2.
        0: Stimulus sends block 3.
        0: Stimulus sends block 4.
        0: Stimulus sends block 5.
```

```
       0: Stimulus sends block 6.
       0: Stimulus sends block 7.
       0: Stimulus sends block 8.
       0: Stimulus sends block 9.
       0: Stimulus sends block 10.
UARTCTRL: Serial device Slave pts [/dev/pts/5]
Note: Uploaded the Program-Binary: ./CPU1/userCode
UARTCTRL: Serial device Slave pts [/dev/pts/6]
Note: Uploaded the Program-Binary: ./CPU2/userCode
       0: Stimulus sends block 11.
       2: Stimulus sends block 12.
      62: Stimulus sends block 13.
     123: Stimulus sends block 14.
Register IRQ Register IRQ 00 pFunc  pFunc      183: Stimulus sends block 15.
0x0x5e05e0 pArg  pArg 0x0x00
243: Stimulus sends block 16.
.
.
.
.
.
.....
   89338: Monitor received block 172 (with 4 bytes).
   89338: Encoding for block 172 took   13623 micro seconds.
   89834: Monitor received block 173 (with 4 bytes).
   89834: Encoding for block 173 took   13607 micro seconds.
   90329: Monitor received block 174 (with 3 bytes).
   90329: Encoding for block 174 took   13599 micro seconds.
   90863: Monitor received block 175 (with 15 bytes).
   90863: Encoding for block 175 took   13629 micro seconds.
   91392: Monitor received block 176 (with 9 bytes).
   91392: Encoding for block 176 took   13653 micro seconds.
   91904: Monitor received block 177 (with 6 bytes).
   91904: Encoding for block 177 took   13640 micro seconds.
   92406: Monitor received block 178 (with 6 bytes).
   92406: Encoding for block 178 took   13604 micro seconds.
   92979: Monitor received block 179 (with 305 bytes).
   92979: Encoding for block 179 took   13686 micro seconds.
   92979: Monitor exits simulation.
```

A.3 SCE Design Steps for Three-CPU Implementation

```
Here are some detailed instructions to run JPEG simulation with three ARM ISSs which are
    quite similar to the ones with one CPU.
* Specification step
- new project
- import JpegPlatform.sc
- add to project as "PlatformSpec.sir"
- view the new hierarchy chart (entirely, incl. connectivity)
- compile and simulate

* Architecture Refinement step
- choose "platform" as top-level
- allocate three ARM_7TDMI as "CPU1" ,"CPU2" and "CPU3" respectively
- allocate two HW_Standard as "HW1" and "HW2"
- allocate two HW_Virtual as "IO_Unit1" and "IO_Unit2"
- map datain to IO_Unit1
- map dataout to IO_Unit2
- map cin to IO_Unit1
- map cout to IO_Unit2
```

```
- map dct1 to CPU1
- map dct2 to HW1
- map quantize to CPU2
- map zigzag to HW2
- map Huffman to CPU3
- perform architecture refinement (no timing back annotation)
- rename generated model as "PlatformArch"
- compile and simulate

* Scheduling Refinement
- use priority-based scheduling for CPU1,CPU2 and CPU3
  (no priorities needed for CPU1, CPU2 and CPU3 because only one behavior assigned)
- leave IO_Units alone
- leave HW1 and HW2 alone
- perform scheduling refinement (both static and dynamic)
- rename generated model as "PlatformSched"
- compile and simulate

* Network Refinement
- rename "Bus0" to "CPU1_Bus"
- rename "Bus1" to "CPU2_Bus"
- rename "Bus2" to "CPU3_Bus"
- create additional "Port1" for HW1
- create additional "Port1" for HW2
- connect HW1 (Port0) to CPU1_Bus as "slave4"
- connect HW1 (Port1) to CPU2_Bus as "slave4"
- connect HW2 (Port0) to CPU2_Bus as "slave5"
- connect HW2 (Port1) to CPU3_Bus as "slave4"
- connect IO_Unit2 (Port0) to CPU3_Bus as "slave5"
- connect IO_Unit1 (Port0) to CPU1_Bus as "slave5"
- perform network refinement
- rename generated model as "PlatformNet"
- compile and simulate

* Communication Refinement
- press "CPU1_Bus"tab
- select start address 0x40000000 for c_link_CPU1__HW1
- select start address 0x50000000 for c_link_IO_Unit1__HW1
- press "CPU2_Bus"tab
- select start address 0x50000000 for c_link_CPU2__HW2
- select start address 0x40000000 for c_link_HW1__CPU2
- press "CPU3_Bus"tab
- select start address 0x40000000 for c_link_HW2__CPU3
- select start address 0x50000000 for c_link_CPU3__IO_Unit2
- perform communication refinement (pin-accurate model)
- rename generated model as "PlatformComm"
- compile and simulate
- Simulation should be working fine. It does not get stuck different from one CPU case.

* Code generation for CPU1,CPU2 and CPU3
Now SCE does not support the multiple C Code generation on the fly.
So C Codes for both CPU1 and CPU2 needs to generated separately until the next version of
    SCE is released.
- close PlatformComm.sir
- sir_note PlatformComm ARM_7TDMI_Core_20000_0_CPU1 '_PE_HAL_MODEL="
    ARM_7TDMI_HAL_20000_0_CPU1"'
- sir_note PlatformComm ARM_7TDMI_Core_20000_0_CPU2 '_PE_HAL_MODEL="
    ARM_7TDMI_HAL_20000_0_CPU2"'
- re-open PlatformComm.sir
```

```
- sir_note PlatformComm ARM_7TDMI_Core_20000_0_CPU3 '_PE_HAL_MODEL="
    ARM_7TDMI_HAL_20000_0_CPU3"'
- perform code generation for CPU1 (for ARM_7TDMI_OS_20000_0_CPU1_NET)
  (store in files "CPU1/CPU1.c" and "CPU1/CPU1.h")
- rename generated model as "PlatformCommC1"
- compile and simulate
- perform code generation for CPU2 (for ARM_7TDMI_OS_20000_0_CPU2_NET)
  (store in files "CPU2/CPU2.c" and "CPU2/CPU2.h")
- rename generated model as "PlatformCommC2"
- compile and simulate
- perform code generation for CPU3 (for ARM_7TDMI_OS_20000_0_CPU3_NET)
  (store in files "CPU3/CPU3.c" and "CPU3/CPU3.h")
- rename generated model as "PlatformCommC3"
- compile and simulate
- Simulation should be working fine. It does not get stuck different from one CPU case.

* Cross-compilation for CPU1
Now SCE does not support the multiple ARM ISS. Only one ARM ISS simulation is allowed.

- we will cross-compile the generated C code in the CPU1 directory
  cd CPU1
- inspect the generated ANSI-C code in files "CPU1.c" and "CPU1.h"
It appears that the generated C code converts our back-annotated timing
estimates into TaskDelay statements that suspend a task for the given
period of time. Thus, the software tasks are actually put to sleep for
our back-annotated waitfor() statements!!

After the C code generation step, take a look at the file CPU1.h.
At the top of the file, you will find the following definition:
#define WAITFOR(X) TaskDelay((unsigned long)((X)/1000))
Please change this to the following:
#define WAITFOR(X) // nothing!
This will make the bogus waiting disappear.
- to compile the generated code together with the microC-OS-II
  (and some other files), we'll use a prepared Makefile
cp /home/doemer/EECS222C_F08/lecture8/Makefile .
- update Makefile for reference path for SWARM ISS and user source code like below.
    # reference to SWARM ISS
    #ISS_DIR = /opt/pkg/sw/swarm
     ISS_DIR = /home/lecs/chkout/project/multi_swarm/swarm

   # USER specific source file
      USR_SRC     := CPU1.c
      USR_HDR     := CPU1.h
- type "make" after modification of Makefile
- the generated ARM-executable is found in file "userCode"

* Cross-compilation for CPU2
Now SCE does not support the multiple ARM ISS. Only one ARM ISS simulation is allowed.
- we will cross-compile the generated C code in the CPU2 directory
  cd CPU2
- inspect the generated ANSI-C code in files "CPU1.c" and "CPU1.h"
  more CPU2.c
  more CPU2.h
- to compile the generated code together with the microC-OS-II
  (and some other files), we'll use a prepared Makefile
  cp /home/doemer/EECS222C_F08/lecture8/Makefile .
- update Makefile for reference path for SWARM ISS and user source code like below.
    # reference to SWARM ISS
    #ISS_DIR = /opt/pkg/sw/swarm
```

```
    ISS_DIR = /home/lecs/chkout/project/multi_swarm/swarm

  # USER specific source file
    USR_SRC     := CPU2.c
    USR_HDR     := CPU2.h
- type "mae" after modification of Makefile
- the generated ARM-executable is found in file "userCode"
* Cross-compilation for CPU3
Now SCE does not support the multiple ARM ISS. Only one ARM ISS simulation is allowed.

- we will cross-compile the generated C code in the CPU3 directory
  cd CPU3
- inspect the generated ANSI-C code in files "CPU3.c" and "CPU3.h"
  more CPU3.c
  more CPU3.h
- to compile the generated code together with the microC-OS-II
  (and some other files), we'll use a prepared Makefile
- cp /home/doemer/EECS222C_F08/lecture8/Makefile .
- update Makefile for reference path for SWARM ISS and user source code like below.
    # reference to SWARM ISS
    #ISS_DIR = /opt/pkg/sw/swarm
     ISS_DIR = /home/lecs/chkout/project/multi_swarm/swarm

  # USER specific source file
    USR_SRC     := CPU3.c
    USR_HDR     := CPU3.h
- type "make" after modification of Makefile
- the generated ARM-executable is found in file "userCode"

* Insertion of ISS model for CPU1, CPU2 and CPU3
- select Comm.sir (in SCE Project window)
- Edit->ImportDesign
  "/home/lecs/chkout/project/multi_swarm/arm7tdmiwrapper/arm7tdmiiss1.sir"
  (this will show up in the list of unused behaviors in the working window
  as behavior "ARM_7TDMI_ISS1")
- Edit->ImportDesign
  "/home/lecs/chkout/project/multi_swarm/arm7tdmiwrapper/arm7tdmiiss2.sir"
  (this will show up in the list of unused behaviors in the working window
  as behavior "ARM_7TDMI_ISS2")
- Edit->ImportDesign
  "/home/lecs/chkout/project/multi_swarm/arm7tdmiwrapper/arm7tdmiiss3.sir"
  (this will show up in the list of unused behaviors in the working window
  as behavior "ARM_7TDMI_ISS3")
- locate the instance "CPU1" of behavior "ARM_7TDMI_Core_20000_0_CPU1"
  in the hierarchy browser
- replace this abstract model with the ISS model
(right-click "ARM_7TDMI_Core_20000_0_CPU1", ChangeType to "ARM_7TDMI_ISS1")
- locate the instance "CPU2" of behavior "ARM_7TDMI_Core_20000_0_CPU2"
  in the hierarchy browser
- replace this abstract model with the ISS model
(right-click "ARM_7TDMI_Core_20000_0_CPU2", ChangeType to "ARM_7TDMI_ISS2")
- locate the instance "CPU3" of behavior "ARM_7TDMI_Core_20000_0_CPU3"
  in the hierarchy browser
- replace this abstract model with the ISS model
(right-click "ARM_7TDMI_Core_20000_0_CPU3", ChangeType to "ARM_7TDMI_ISS3")


- go to Project menu -> Settings
- change Import path to ".:/home/lecs/chkout/project/multi_swarm/arm7tdmiwrapper"
- change Library path to "/home/lecs/chkout/project/multi_swarm/swarm"
```

```
- set Verbosity level and Warning level to 2 .
- save this design model (as a new model in the project)
  File->SaveAs "PlatformISS.sir" (in your working directory!)
- compile and simulate
(Simulation should be ended like below.)
       0: Stimulus sends block 9.
       0: Stimulus sends block 10.
UARTCTRL: Serial device Slave pts [/dev/pts/2]
Note: Uploaded the Program-Binary: ./CPU3/userCode
UARTCTRL: Serial device Slave pts [/dev/pts/3]
Note: Uploaded the Program-Binary: ./CPU2/userCode
UARTCTRL: Serial device Slave pts [/dev/pts/4]
Note: Uploaded the Program-Binary: ./CPU1/userCode
       0: Stimulus sends block 11.
Register IRQ Register IRQ Register IRQ 000 pFunc  pFunc  pFunc
0x0x0x5e05e05e0 pArg  pArg  pArg 0x0x0x000
     402: Stimulus sends block 12.
     821: Stimulus sends block 13.
    1197: Stimulus sends block 14.
    1597: Stimulus sends block 15.
    1817: Monitor received block 0 (with 337 bytes).
    1817: Encoding for block 0 took    1817 micro seconds.
    .
    .
    .
    .
......
   90817: Encoding for block 174 took    8157 micro seconds.
   91350: Monitor received block 175 (with 15 bytes).
   91350: Encoding for block 175 took    8189 micro seconds.
   91879: Monitor received block 176 (with 9 bytes).
   91879: Encoding for block 176 took    8213 micro seconds.
   92391: Monitor received block 177 (with 6 bytes).
   92391: Encoding for block 177 took    8196 micro seconds.
   92894: Monitor received block 178 (with 6 bytes).
   92894: Encoding for block 178 took    8175 micro seconds.
   93467: Monitor received block 179 (with 305 byt
es).
   93467: Encoding for block 179 took    8236 micro seconds.
   93467: Monitor exits simulation.
```