An SPU Reference Model for Simulation, Random Test Generation and Verification

Yukio Watanabe

Toshiba Corporation
Semiconductor Company
580-1 Horikawa-Cho, Saiwai-Ku,
Kawasaki 212-8520, Japan
yukio.watanabe@toshiba.co.jp

Balazs Sallay, Brad Michael,
Daniel Brokenshire, Gavin Meil,
Hazim Shafi

IBM
11501 Burnet Rd,
Austin, TX 78758, U.S.A.
{balazs, bradmich, brokensh, meil, hshafi}
@us.ibm.com

Daisuke Hiraoka

Sony Computer Entertainment Inc.
2-6-1 Minami-Aoyama, Minato-ku,
Tokyo 107-0062, Japan
hiraoka@rd.scei.sony.co.jp

Abstract – An instruction set level reference model was developed for the development of synergistic processing unit (SPU) , which is one of the key components of the cell processor [1][2]. This reference model was used for the simulators to define the instruction set architecture (ISA), for the random test case generator, for the reference in the verification environment and for the software development. Using the same reference model for multiple purposes made it easier to keep up with the architecture changes at the early stage of the microprocessor development. Also including the reference model in the simulation environment increased the robustness for the random test executions and made it possible to find bugs that are usually difficult to catch.

I Introduction

The Synergistic Processing Unit (SPU) is the first implementation of a new processor architecture designed to accelerate media and streaming workloads. The SPU instruction set architecture (ISA) was defined considering the physical implementation such as area, timing and power efficiency as well as the efficiency to run media and streaming applications. To achieve the target performance, ISA definition was frequently changed as the logic and physical design advanced.

To define the ISA, it was very important to write workload codes using the candidate ISA and evaluate its performance. The first reference model was implemented for this purpose as a simulator along with the assembler program. As the project advanced, the ISA definition changes were implemented in the reference model first, and then the updated reference model was used by various applications.

Since ISA changes continued even after verification started, various teams such as design team, performance analysis team and verification team were affected by these changes. However, since the SPU reference model could be included and used in those applications, only the reference model had to be modified and it was considered as the 'golden' ISA definition. Using the same reference model in various applications could reduce the mistakes when the applications needed to be updated because of the change, and it also became easier to keep up with the ISA changes.

In this paper, the basic structure of the reference model is described and followed by the explanation of applications that use the reference model. Among those applications, the verification environment is described in detail in a separate chapter because the reference model played a very important role to build up an effective verification environment.

II Instruction Set Reference Model

The SPU reference model is a set of C programs that perform SPU instruction execution. The reference model is comprised of following components.

- All the architected memory/register resources such as a register file, a local storage and status registers.
- Instruction decoding.
- Instruction execution and updating architected memory/register resources as the result of the instruction execution.

All the architected memory/register resources are defined in a C struct as 'struct APU_t' and this struct is usually memory allocated in each application program and used as the argument to the reference model. However, the random test case generator uses its own memory/register resource structure for some reasons, which will be described later.

The major functions defined in the reference model are following two functions.

```
void INTERPRETER_init(void);
void INTERPRETER_exec(APU_t *apu, unsigned int inst);
```

The first function is used for initialization of the reference model. The second one is used for instruction execution. The first argument of the second function is a pointer to the struct of architected memory/register resources APU_t, which represents the entity of the SPU. The second argument is the instruction encoded in a 32bit integer. When this function is invoked, the instruction of the second argument is decoded and a proper function that corresponds to the instruction is called in the reference model. The following C program is an example of the function that is called when ‘a’ instruction is decoded.

```
/* add word: a rt, ra, rb */
void INTERPRETER_inst_a(APU_t *apu,
                        unsigned int inst)
{
  RTW(0) = RAW(0) + RBW(0);
  RTW(1) = RAW(1) + RBW(1);
  RTW(2) = RAW(2) + RBW(2);
  RTW(3) = RAW(3) + RBW(3);
}
```

RTW(n), RAW(n) and RBW(n) are macros which specify the register files whose numbers are determined by the portions of the instruction ‘unsigned int inst’. These macros specify the memory elements in the SPU memory/register struct ‘APU_t *apu’.

As for the floating instruction implementations, an SPU specific floating-point model was written and used in the functions that represent the floating instructions. This is because the SPU introduced a new floating-point architecture that differed in some ways from the commonly used IEEE 754 floating-point standard, in order to enhance performance for media-oriented applications. One method for calculating floating-point results in a C-language reference model is to use C's "float" and "double" data types. However, operations using such data types will be calculated using the floating-point hardware native to the machine executing the reference model, which may not match the SPU's architected behavior in certain cases.

To avoid any dependency on the underlying machine's floating-point hardware, the floating-point model within the SPU reference model was written using only integer data types, using separate integer variables to store the sign, exponent, and significand parts that make up a floating-point value. A 64-bit integer type is used to store the significand, to support the precision required by single- and double-precision floating-point instructions. The floating-point reference model includes functions to normalize, align, add, multiply, multiply-and-add, and round, all using integer arithmetic; as well as functions to "pack" the final results into the 32- or 64-bit representation called for by the architecture. The reference model detects and handles all exception cases such as overflow, underflow, divide by zero and loss of precision.

Since there can be multiple SPUs in one system, the SPU memory struct might be allocated multiple times and each one is used to represent the resources of each SPU. As shown in Figure 1, when the reference model is used in an application, the application program allocates SPU architected memory/register resource struct as many as the number of the SPUs that the application program needs to handle. The application program can directly access the resources in each SPU, and can execute the instruction by passing one of the pointer of the SPU memory/register struct in INTERPRETER_exec() function.

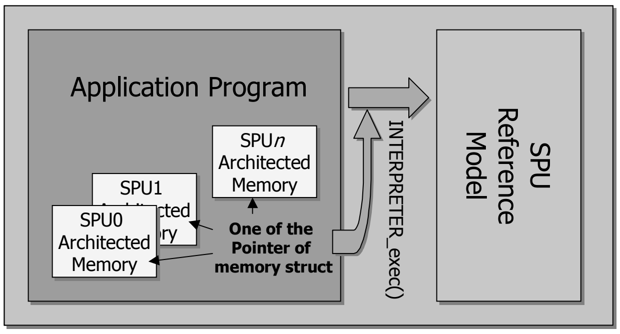


Fig. 1. Basic usage of the SPU reference model in an application

The definition of each instruction except for its function is written in a common definition file as a C macro. Following is an example of the definition of ‘a’ instruction.

```
APUOP(M_A,  RR, 0x0c0, "a", ASM_RR, 00112, FX2)
                              /* Add% RT<-RA+RB */
```

APUOP is the name of the macro that defines each instruction. The first argument of the macro is an identifier of the instruction. The second argument is the instruction format. This is followed by the op-code, the mnemonic, the assembler format, the register file usage and the kind of the pipeline used to execute the instruction.

The reference model itself does not use all the information in the macro. However, other applications such as the assembler, the pipeline simulator, the logic RTL and the verification environment refer the same common definition macro as well as the reference model, and each item in the macro is used by at least one application.

For example, the 6th argument in the macro indicates which register is the source register and which register is the target register. Each SPU instruction can take up to 4 registers as the arguments of the instruction depending on the instruction format. The four registers are represented as RA, RB, RC and RT. There are five digits in the 6th argument. From the most significant digit, the first one is always 0. The second digit corresponds to RC, and then the following digits correspond to RB, RA and RT, respectively. If the digit is 0, the register is not used by the instruction. If the digit is 1, that means the register is used as a source. If the digit is 2, the register is used as a target, and if the digit is 3, the register is used both as a source and a target. In case of ‘a’ instruction, two values of register RA and register RB are the source registers and their values are added and the result is put into the target register RT. So ‘00112’ is put into the 6th argument of the macro. The RTL for register dependency checking logic uses this information. A script converts the macro description into the RTL description.

III Usages of the Reference Model

The SPU reference model is used for various applications. In this section, some of them are introduced.

A. Instruction Simulator

Since the SPU is a new architecture, it was very important to implement and use the instruction simulator to evaluate the new ISA. Actually, the initial reference model was developed for this purpose. Figure 2(a) shows the configuration of the instruction simulator. In this figure, assembler is also included. The assembler program uses the common definition file described in section II as well as the reference model, and only changing the common definition file is required for the instruction changes. In the instruction simulator, a COFF format file generated by the assembler is read and the information (memory contents and the program counter value) are written into the SPU architected memory/register struct. The simulator gets commands via command prompt or command files. Commands for the instruction simulator includes

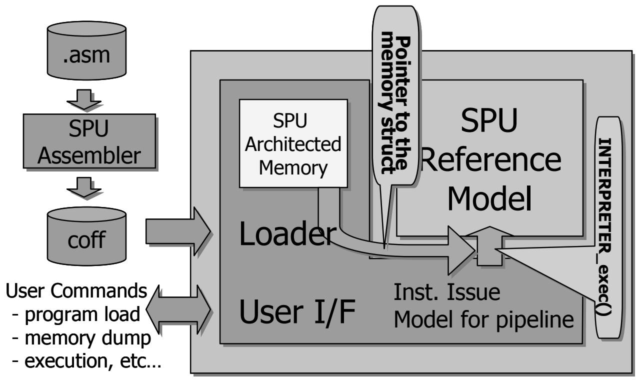
- Memory/Register dump
- Breakpoint setting
- Running program
- Step execution
- Show statistics such as number of instructions executed

The instruction simulator was mainly used to study the completeness of the instructions as a set and the correctness of the definition of each instruction. For example, at the beginning of the ISA study, the SPU ISA had byte/half word/word/double word-wise load/store instructions, but to realize a high frequency microprocessor, those instructions were divided into multiple instructions --- load quad word, generate control and shuffle byte instructions. Writing a code and running simulations with combining those instructions proved the completeness of the instruction set. Another example is that SPU has a series of right shift/rotate instructions that are represented as 'rotate mask' and 'rotate mask algebraic' instructions. The definitions are complicated and simulating those instructions found the bugs in the instruction definitions by running workload programs.

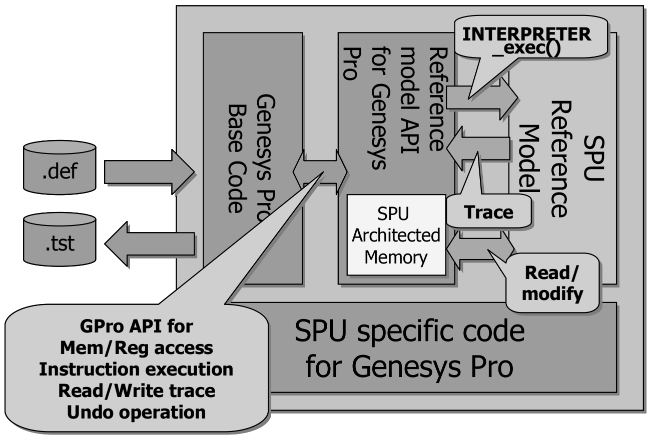
Extended Programming Features

In order to validate the SPU Instruction Set Architecture and implementation, real application workloads were developed. Since the targeted, media rich applications are visual in nature, verification of correct and efficient program execution required the development of several extended programming features. These features, which include check-pointing, file I/O, and streaming I/O, where added as special simulator extensions to the architected instruction set.

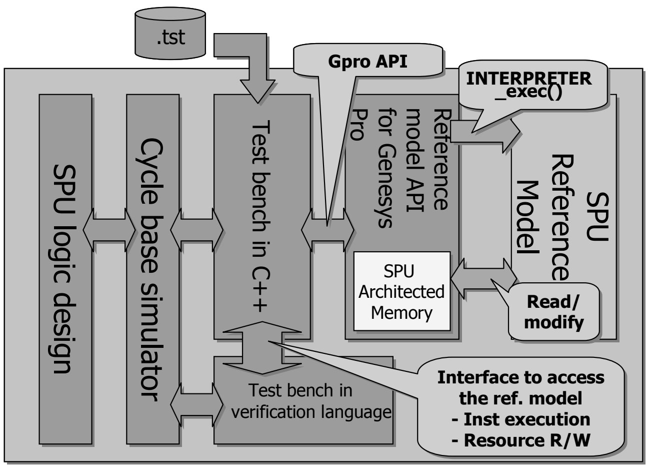
- Check-pointing



(a)


.def
.tst
Genesys Pro Base Code
Reference model API for Genesys Pro
INTERPRETER _exec()
SPU Reference Model
SPU Architected Memory
Trace
Read/ modify
GPro API for Mem/Reg access Instruction execution Read/Write trace Undo operation
SPU specific code for Genesys Pro

(b)



(c)

Fig. 2. Applications using the SPU reference model. (a)Instruction/Cycle-base simulator. (b) Random test case generator, (c) SPU verification environment.

For the purposes of computing cycle times for user specified code sections, software runtime check-pointing was provided by adding extended simulator operations for AND instructions in which all register fields are equivalent. For example:

- ♦ `and r0, r0, r0` – clear the current instruction/cycle counts.
- ♦ `and r30, r30, r30` – start counting instructions and cycles.
- ♦ `and r31, r31, r31` – stop counting instructions and cycles.
- ♦ `and r#, r#, r#` (where # is a number 1-29) – output check-point # including the current instruction and cycles counts.

These instructions can easily be added to C language programs using inline assembly.

- File I/O

To support rudimentary debugging of SPE programs, the simulator was enhanced to support a full-function, file print (printf) subroutine. Since the SPU has architected Special Purpose Registers (SPRs) but its implementation contains no SPRs, the mtspr instruction simulation was extended such that the unused RB opcode field is used as an indicator of the extended function. The simulator's printf function is serviced by fetching parameters in accordance with the SPU's ABI (Application Binary Interface) standard.

- Streaming I/O

To support the display of graphical workloads, the simulator was also extended to enable external socket communications. The mtspr instruction was again used to make Unix sockets requests so that graphical data can be streamed to an external display program. The streams (connection sockets) are identified by a filename and its stream direction (outgoing or incoming). The supported socket requests include opening, closing and selecting a socket connection, as well and sending and receiving data on the current (outgoing and incoming, respectively) socket connection.

B. *Pipeline Simulator*

The pipeline simulator can count the actual cycle number when a SPU program is executed. This is used to evaluate the performance of the SPU micro architecture and to optimize the SPU application programs. The basic structure of the pipeline simulator is almost same as the instruction simulator case in Figure 2(a) except that it has an instruction fetch and issue model. This model takes into account the instruction fetch flow including local store access, instruction line buffers and other pipeline buffers, branch hint instructions and instruction issue controls considering the register dependencies and structural hazards and so on. There is a common pipeline definition file that describes the depth of each pipeline and the resources that each pipeline uses, which made it easier to evaluate various pipeline configurations.

The same user interface as the instruction simulator is used in the pipeline simulator as well. In the actual simulator program, both the instruction simulator and pipeline simulator are consolidated in one program and the simulation mode (instruction or pipeline) can be switched by a simulator command.

By modifying the common pipeline definition file, the impact of the pipeline depth change could be easily estimated for each workload program. This played an important role when the pipeline depth of the floating-point instruction was determined. Also the pipeline simulator was used to optimize workload programs. This includes the evaluation of the SPU C/C++ compiler optimization. By utilizing the pipeline simulator, one workload program that calculates the product of two large matrices was fully optimized and could achieve almost 0.5 cycles per instruction (CPI), which is the best case for the dual issue microprocessor.

Since the reference model covered the entire instruction execution portion, by using the reference model, the pipeline simulator programmer did not have to work for the instruction changes at all and could concentrate on the instruction fetch/issue logic changes.

C. *Random Test Case Generator*

In the SPU verification, random test cases are used to stimulate the logic function. To guarantee the quality of the verification, function coverage is used as the metric. It is important to generate high quality random test cases to hit all the function coverage items. To generate directed random test cases, Genesys Pro (GPro)[3], a tool developed by IBM, was applied to the SPU. GPro has a base core code that uses a specific application program interface (API) to communicate with various kinds of reference models. So Gpro becomes able to communicate with the SPU reference model by preparing the specific API functions that GPro uses for the SPU reference model. The API functions include

- Accessing the architected memory resources.
- Executing an instruction at the program counter and get the information what memory/register resources were read to execute the instruction and what memory/register resources were updated by executing the instruction.
- Setting undo points and undoing to handle mispredicted branches and recovery from test case generation failure.

To record the resource access traces and manage the undoing automatically using the reference model, all the SPU memory and register resources were defined in the random test case generator without using the struct APU_t which was described in section II. A C++ class was defined to access those memory/register resources and the resource access trace generation and undoing management. To utilize the reference model, the macros used in each instruction implementation such as RTW(n), RAW(n) and RBW(n) described in section II in the 'a' instruction example were redefined to use the class to access the appropriate memory/register resources. By using the operator overloading of the class, read trace information is added automatically when the resource was evaluated in the right hand side of the '=' operator, and the write trace and undo information are added automatically when the resource was placed in the left hand side of the '=' operator and updated with the value of right hand side. Thus, by redefining the macros in the reference model to use the C++ class, it became possible to use the same reference model and add the resource access trace and undoing functions.

Figure 2(b) shows the structure of the random test case generator. A definition file (.def file) is given to the random test case generator and a test case file (.tst file) is randomly generated with constraints written in the definition file. To generate the test case, GPro base code communicates with the SPU reference model. In this case, the API portion has the entity of the SPU memory/register resources. When GPro accesses the memory resources using the API, the content of the resources in the API is directly accessed. When GPro calls a function to execute an instruction specified by the program counter, the API portion picks up the instruction at the program counter in the SPU memory resources and invokes the function INTERPRETER_exec() with passing the instruction. The memory/register resources in the API are updated properly and the resource access information is generated and passed to the GPro base code and the information is used for the test case generation.

D. Reference in the Verification Environment

Figure 2(c) shows the SPU verification environment structure. There are two separate test benches written in C++ language and other verification language. The reference model is used to obtain the expected results for the checkers in the test bench written in the verification language. The C++ test bench is used to provide the API to the other test bench to communicate with the reference model to get and put the values of memory/register resources and get the expected values by executing instructions in it. In this application, the same API used by GPro, the random test case generator is reused as the interface to the reference model.

A test case file (.tst file) generated by GPro is given to the C++ test bench and the initial values in the test case are written into both SPU reference model memory/register resources and the logic design under the RTL simulator. When an instruction is executed in the SPU logic, it is observed by the monitor in the test bench written in the verification language and the test bench executes one instruction in the reference model and gets the expected results through the API.

There are also drivers in the test bench which stimulate the SPU external interface buses. Monitors in the test bench observe the external interface buses. When there is a stimulus on the external interface buses which is considered to update the SPU memory/register resources, the resources in the reference model is updated with the value on the buses at a proper timing using the API. An example is a DMA write access. When a DMA write access happens, the content on the data bus is written into the SPU local storage with the address specified by the address bus. If a memory load instructions refers the address updated by the DMA transaction later, actual logic will use the updated value and the load instruction executed in the reference model also uses the same value. Thus, even if the driver issues random external bus stimulus, the reference model can generate accurate expected values.

E. Cell System Simulator

The SPU reference model was integrated with the IBM full system simulator known as Mambo [4] at an early stage of the development program. Mambo is an execution-driven full system simulator that allows multiple system configurations to be simulated. For example, Mambo was adopted as the software bring-up environment during the development of the CELL architecture. It has been used for operating system development, programming model investigations, compiler development, porting of important applications and libraries, and performance tuning. During the early stages of the Cell project, Mambo was intended to be functionally accurate and as such did not include cycle-accurate models of the full system. As the development project matured, so did the Mambo model, which was enhanced to provide more cycle-accurate feedback about the interactions between the SPUs, system memory, and the PowerPC core. The availability of such a model was crucial for studying the performance of the architecture at an early stage and to allow the performance tuning of applications as the project progressed.

IV. Reference model usage in the verification environment

As briefly described in section III-D, the SPU simulation environment incorporates the reference model, which made it possible to generate the correct expected values on the fly even with the asynchronous random external stimuli. This feature is very important for the SPU verification environment because lots of corner cases can be covered by the combination of random instruction sequences and random external transactions. The important thing here is that the instruction sequences and external transactions can be generated completely independent. The reference model can keep the same state as the actual logic with the help of the test bench and can continue the verification generating the correct expect values.

However, since the reference model is not a pipeline model, there are lots of things to be solved to integrate the reference model and make it work properly in the simulation environment. In this section, some of the items that need consideration are described.

A. Timing to execute the instruction

To integrate the reference model in the simulation environment, what had to be considered first was the timing when to execute the instruction in the reference model. Since the SPU is an in-order issue in-order completion processor, an instruction can be executed in the reference model to generate expected values whenever after an instruction is committed in the actual logic unless there are external transactions. However, since there are interactions between the instruction execution and external transactions they have to be handled properly to generate the correct expected values.

There are two kinds of external transactions. One is a DMA transaction that reads or writes local store memory. The other is a channel access transaction which reads or writes channel registers. The verification test bench monitors the external transactions and checks if the transactions were properly executed. At the same time, the test bench updates the reference model memory/register resources at a proper timing so that the instruction execution can use the proper memory/register values. As shown in figure 3, the actual logic accesses the local store memory array at the 'p' stage of the pipeline which is 2 cycles after the load/store instruction committed at 'n' stage. In case of channel instructions, channel registers are accessed at 'q' stage, which is 3 cycles after the channel instruction committed. To cope with the pipeline stage differences to access the resources with the reference model which doesn't have the concept of pipeline, a decision was made to make all the updates of the memory/register resources including external transactions and instruction executions by the test bench happen at 'q' stage. This made it possible to execute instructions in the reference model with asynchronous external transaction occurrences. However, since the timing to update the local memory array is different between the actual logic and the reference model, the checkers have to take care of this. For example a checker which checks if a store instruction updated the memory array properly has to get the actual logic value at 'p' stage, then after the store instruction is executed in 'q' stage in the reference model, the expected value is obtained from the reference model and it is compared against the actual value obtained at 'p' stage.

B. Resource confliction case

Channel instructions and external channel transactions can access the same channel registers exactly at the same timing.

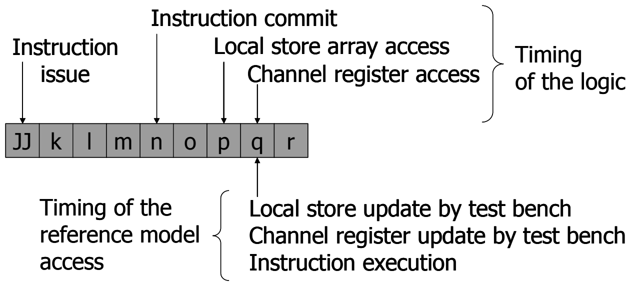


Fig. 3. Pipeline diagram of the actual logic pipeline and the timing to access the reference model.

Since what will be written into or what will be read out from a particular channel register in such a conflict case depends on the hardware implementation and there is no regularity, all the rules are described in the test bench and when a resource conflict happens, the test bench arbitrates this to generate the correct expect values. For example, there are cases that both of an instruction execution and an external transaction try to write into the same channel resource. The instruction is always executed in the reference model at 'q' stage and the channel register in the reference model is updated and then the expected value for the channel instruction result is obtained from the reference model. When the channel instruction has a higher write priority than the external transaction, the test bench does not update the channel register in the reference model with the value supposed to be written by the external transaction, but still obtain the expected value for the external transaction from the reference model. On the other hand, when the external transaction has a higher write priority, the test bench updates the channel register in the reference model at the same 'q' stage but after the instruction execution with the value that the external transaction is supposed to write. Then the test bench obtains the expected value for the external transaction from the reference model, and also the test bench overwrites the expected value for the channel instruction that was obtained right after the instruction execution in the reference model. Thus, with the help of the test bench, reference model can keep the same channel register values as the actual logic and continue to provide the correct expected results.

C. Self modifying code case

In the actual logic, instructions are fetched from the local store memory and travel through the pipeline stages. What will happen if a 'store' instruction writes into an address of the instructions following the store instruction? Since the instructions had been fetched a while ago, the next instructions would be the instructions before modification. On the other hand, if the same store instruction was executed in the reference model, the next instruction executed in the reference model would be a modified instruction because there is no concept of pipeline in the reference model. Therefore, if a self-modifying code is executed, the reference model cannot continue executing the same sequence as the actual logic. To handle this case, test bench keeps information which memory addresses were written by either store instruction or DMA write transactions. Before the instruction is executed in the reference model, the test bench compares the instruction to be executed with the instruction committed by the logic. If they are different, it is usually reported as a test failure, but if the instruction address was modified recently enough to make a difference, the content of the memory of the instruction address in the reference model is saved and replaced by the instruction value that was committed in the actual logic, and the instruction is executed in the reference model. Then unless the executed instruction is not a store instruction that modifies itself, the saved instruction is restored into the reference model. Thus, with the help of the test bench, the reference model can keep up with the actual logic state even with the self modifying code case which instruction set reference model cannot handle properly only by itself.

D. ECC error handling

The SPU has an ECC facility that can correct 1-bit error and detect 2-bit errors in 128-bit data in the local store memory. Since the reference model doesn't have the ECC information in it, the test bench has to support the reference model to handle the ECC. Basically, the ECC errors are injected at the beginning of the simulation or injected by the DMA transactions. The information which address has what kind of ECC errors is kept in the test bench. When a correctable error happens in the logic, the test bench doesn't have to do anything against the reference model because the correctable error should be corrected in the actual logic and the corrected data, which should be the same value in the reference model, will be used. When the instruction that has an uncorrectable ECC error is executed in the actual logic, the test bench can detect it because the test bench has the information of the addresses that have ECC errors. In this case, the test bench saves the original instruction and overwrites the address of the instruction in the reference model with the value modified by the uncorrectable ECC error. Then, as in the case of self-modifying code, the modified instruction is executed in the reference model and the original instruction is restored unless the modified instruction is the store instruction that modifies the instruction itself.

E. Others

There are several other cases such as asynchronous interrupt handling, error handling, undefined instruction handling and so on that cannot be handled only by the instruction level reference model. However, all those cases were implemented in the SPU verification environment with the help of the test bench. So the SPU verification environment has a good robustness that any kinds of combinations of instruction sequence and external transaction can be treated properly and the model generates the correct expected values. If the reference model were implemented as the complete pipeline model, the reference model itself could cover all the cases. But it is usually very difficult to implement the complete pipeline model, and it also slows down the simulation speed. Using the instruction level reference model realized the verification environment with a good quality and performance.

V. Summary and Conclusions

In this paper, the structure and usage of the SPU instruction level reference model was described. The reference model played important roles for the development of the SPU, a novel high performance processor. The reference model was used for the purpose of defining the ISA, analyzing performance, verification environment development and software development. The same reference model could be used for all these purposes, which reduced the burden to keep up with the ISA changes for each application developer and significantly reduced the likelihood of mistakes in the implementation. As for the verification environment, with the help of the test bench, the instruction level reference model could cover pipeline related items and it realized an ability to continue the simulation and generate correct expected values with asynchronous external events. As a result, though the SPU is a very novel processor, only one bug was found in first silicon, which was a mistake of the specification of the asynchronous interrupt. And a demo program that utilizes the SPU was successfully run on first silicon with the expected good performance.

Acknowledgements

The authors would like to thank Seiichiro Saito, Hiroko Fujii and Nobuhiro Kondo of Toshiba Corporation for the development of the prototype of the SPU reference model, and Kanna Shimizu and Peter Hofstee of IBM for useful suggestions.

References

[1] Pham D. et al, “The Design and Implementation of a First-eneration CELL Processor,” 2005 IEEE International Solid-State Circuits Conference Digest of Technical Papers, pp. 184-185, Feb. 2005.
[2] Flachs B. et al, “A streaming Processing Unit for a CELL Processor,” 2005 IEEE International Solid-State Circuits Conference Digest of Technical Papers, pp. 134-135, Feb. 2005
[3] Allon A. et al, “Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification,” IEEE Design & Test of Computers 21(2): 84-93 (2004)
[4] Patrick B. et al, "Mambo -- A Full System Simulator for the PowerPC Architecture,” ACM SIGMETRICS Performance Evaluation Review, 31(4): 8-12, Mar. 2004.