Emulator environment based on an FPGA prototyping board

Kyung-soo Oh, Sang-yong Yoon, Soo-Ik Chae
System Design Group,
School of Electrical Engineering, Seoul National University 301-818,
San56-1, Shinlim-Dong, Kwanak-Gu, Seoul, 151-742, Korea
Email: {kyungsoo,syyoon,chae}@sdgroup.snu.ac.kr

Abstract

In this paper, we describe an emulator environment based on an FPGA prototyping board. This emulator environment is for functional verification of a multi-media processor we are currently developing and for software development and debugging of its application programs. For these purposes, the emulator environment includes a debugging network and provides virtual wires and some utilities, board control functions, and a virtual FPGA board. With this environment, we verifies the functionality of a multi-media processor and implements its cycle level simulator

1 Introduction

Now, the verification procedure of a processor becomes increasingly more important and time consuming. One of the efficient verification methods is to use an FPGA prototyping board [1]. With an FPGA prototyping board, we can verify the functionality of the processor at much higher speed compared to the conventional software simulator. Furthermore, we can test a processor at a system level, which includes external memory, other ICs and I/Os before the processor is fabricated into an IC.

Currently we are developing a multi-media processor for the videophone of which base applications are H.263 for video and g.723.1 for audio. Therefore, we developed an emulator environment based on an FPGA prototyping board as shown in Fig.1. We use the emulator for high-speed functional verification of the multi-media processor and for development of its application software.

This environment can be divided into two parts: hardware and software. The hardware part is comprised of an FPGA prototyping board, debugging networks and virtual wires. The software part is comprised of a virtual board, board control functions and some utilities. Because this environment is simple and modular, it is easily expandable.

The rest of this paper is organized as follows. In Sections 2 and 3, we describe the hardware and software environment of the emulator respectively. In Section 4, we introduce the target processor briefly and describe its porting procedure into the FPGA prototyping board. Finally, the conclusion is in Section 5.

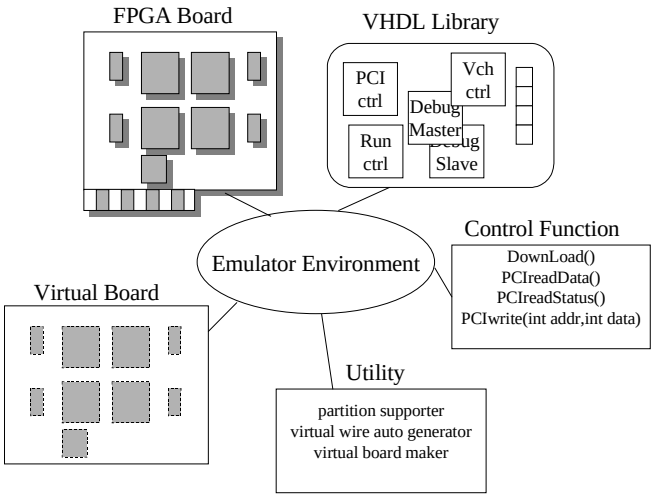


Figure 1: Emulator environment

2 Emulator environment: Hardware

2.1 Configuration of FPGA prototyping board

The overall structure of the FPGA prototyping board is shown in Fig.2. We made the detail configuration of this board for the requirement of verifying the multi-media processor and for the desire to control the FPGA prorotyping board more efficienly. Because of the flexible structure of this board, it can be easily adapted to any other processor or system.

As shown in Fig. 2, four virtex-1000 FPGAs, each of which capacity is one million system gates, are mesh-connected. We selected the mesh connection topology because of its simplicity. Each FPGA has 80 physical wires with its neighbors. We can expand the board capacity easily by increasing the number of FPGA in mesh from 4 to

0-7695-0668-2/00 $10.00 © 2000 IEEE

9. External memory components are three 1MB SRAMs and an 4MB SDRAM with a 32bit data bus. For communication with another board, several external 50pin ports are provided, each of which is connected with an FPGA. FPGA configuration is accomplished with a parallel port and the internal debugging operations are controlled through a PCI interface chip (PCI9050).

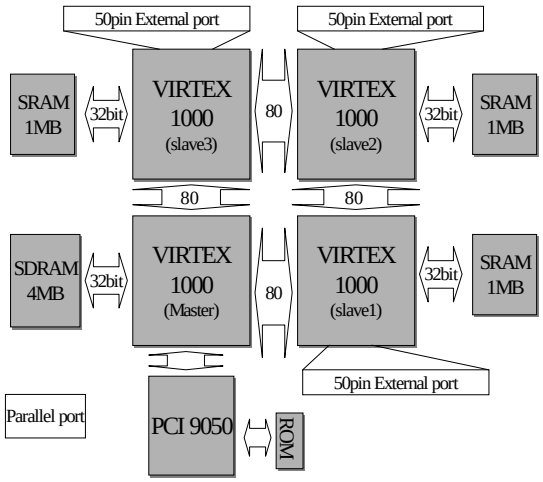


Figure 2: FPGA prototyping board

2.2 Debugging network

One of the important functions of this emulator is debugging capability. Therefore, we did not restrict the debugging function to the memory module. To trace all internal registers and signals, we tried to implement debugging networks that are hierarchical and modular. Fig.3 is its internal debugging network and Fig.4 is its external debugging network.

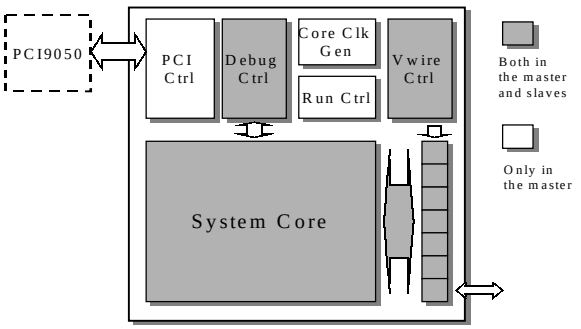


Figure 3: Debugging network (internal)

The front end of the debugging network is the PCI controller which sends or receives control, address and data with PCI9050. This PCI controller has internal address and data buffers.

In read operation, this controller only sends the value of the internal data buffer to PCI9050.

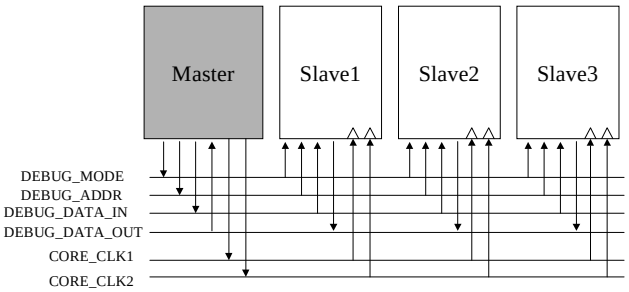


Figure 4: Debugging network (external)

In write operation, this controller store address and data into the internal buffer and then activates the debug controller. The received address is decoded as shown in Fig.5. Chip Id and Local Id determine which debug module will be activated.

31 30 29 28	27 26 25 24	23	22 0
Chip Id	Local Id	R/W	Address

Figure 5: Address field

At first, if the chip id is not “0000”, the debug controller in the master sends the received address and data to all slave FPGAs through the dedicated debug wires. Then the debug controller with the same chip id go to its debug state. In the debug state, the debug controller sends a request signal to the target debug module which has the same local id. After the debug operation ends, the activacted debug module returns an acknowledge signal and the result to its debug controller. Finally, this result is transferred to the data buffer of the PCI controller.

To run the target processor step by step, we can use several methods. One method is to control the core clock of the target processor. Another is to use special debug signals if the function is supplied by the processor. In the target multi-media processor, there is an debug_mode signal. If this signal is asserted, all pipeline registers are disabled and the processor stop immediatly.

In our implementation, the run controller, one of the debug modules controls this debug_mode signal. This run controller has a special counter. By a debug-write operation, this counter is set to the debug data value and decreases its value by one whenever the core clock of the processor toggles. If the counter becomes to zero, the run controller asserts the debug_mode signal and processor stops its operation. In addition to the step operation, we can implement a

0-7695-0668-2/00 $10.00 © 2000 IEEE

break operation, with which the user can stop the processor at a specific program counter.

2.3 Virtual wiring

As explained before, each FPGA has only 80 physical wires to its neighbors. However, the number of signals between the partitioned modules is much more than this number. In partitioning the multi-media processor, we observed that the maximum number of signals between two FPGAs was about 1200. To solve this pin limitation, we decided to use the virtual wiring method [2][3], which slows the clock of the target processor to increase the number of signals transferred between the FPGAs.

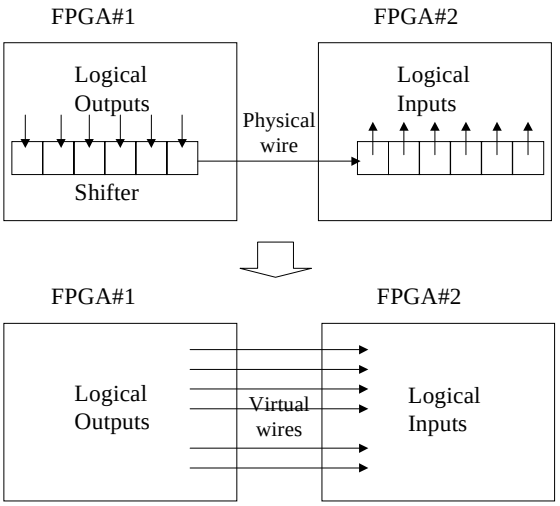
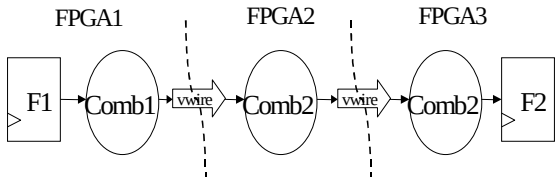


Figure 6: Virtual wire

For simplicity of the implementation, one virtual wire controller in each FPGA controls all the virtual wire operations. “A1” and “A0” are the states for synchronizing to the core clock of a target processor. “E#” is a state for the evaluation of each combinational logic. “LD” is for loading signals and “SHIFT” is for shifting signals in the virtual wire.

When the core clock toggles, F1 has a new value and the virtual wire controller starts its operation. After the evaluation time of comb1 elapses, the virtual wire transfers the result to comb2. Then comb2 starts its evaluation. In phase 2, the same operation repeats. When the core clock toggles again, F2 has the correct value

If the target processor is partitioned as shown in Fig.7 (a), a two-phase operation is sufficient for the correct execution. However, we can manage any multiple-partition structure by simply increasing the number of this phase.



(a) Partition structure

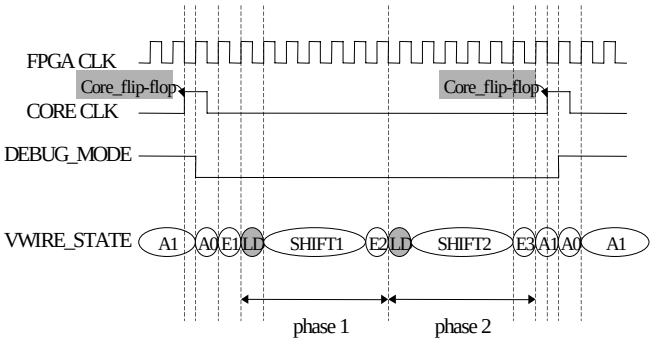


Figure 7: (b) Virtual wire operation

3 Emulator environment: Software

In this section, we explain the software environment. This software environment is comprised of a partition file generator, a virtual wire auto generator, a virtual board generator and board control functions that are basic functions of the emulator program.

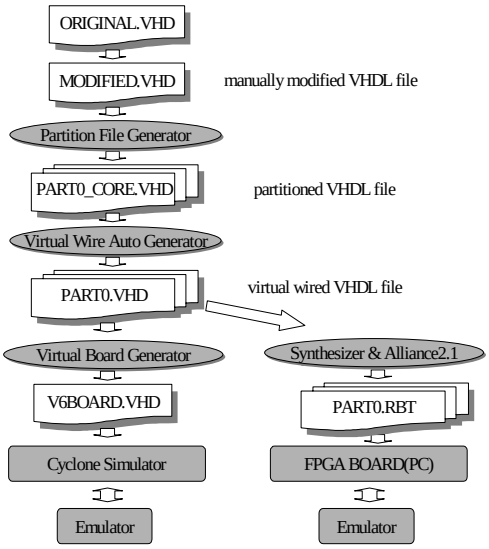


Figure 8: Software Environment

0-7695-0668-2/00 $10.00 © 2000 IEEE

We do not generate any special intermediate format in partitioning and virtual wiring. All input files and output files are VHDL based. Currently this environment requires some manual jobs. Before partitioning, we need to make several modifications.

A. If the size of a cache memory is larger than the size a FPGA can handle, it is necessary to make an appropriate cache model with an external SRAM.

B. If we will access a memory module or signal through the debugging network, it is necessary to insert a debug-interface to this module or signal as described in the section 2.2.

3.1 Partitioning

Presently, partitioning is mostly based on manual jobs. Usually, the top VHDL file consists of several components. In this step, we assume that any component in the top VHDL file must be fitted into an FPGA after logic synthesis. However, this procedure can be automated without difficulty.

Each component is referred to with its instance name. By using this instance name, we need to write a partition directive file and run the partitioning utility to generate the partitioned VHDL files. We can increase the target system speed if we partition the system between flip-flop modules, which may reduce the number of virtual wire operation phase as described in section 2.3.

3.2 Virtual wiring

In this step, physical wires must be converted into virtual ones and some debug controllers are inserted into the partitioned files. We need to write a virtual-wire specification file, which specifies the operation phase number, the number of required virtual wires and evaluation times and run the virtual wire generator to generate the full virtual wired VHDL files. If we insert some debug-interface modules, their appropriate debug controller and debug signals are inserted in this step.

3.3 Virtual board generation

There might be several bugs when a target processor is mapped into the FPGA board. These bugs might be inserted mainly due to the following reasons.

A. Imperfect software utility.

B. Incorrect debug-module.

C. Incorrect usage of the board control function.

D. Discord between the partition structure and the virtual wire operation

E. Timing problem such as clock skew and delay.

"A" occurs frequently in the software utility development stage, which is almost fixed. "B", "C" and "D" arise due to the manual works, which can be eliminated if we automated it. "E" is a critical problem in a real board, which might be eliminated by simply slowing the operation clock

We might spend much time in finding when, where and why these bugs are inserted on the real board. However, these bugs can be easily detected and fixed with the virtual board.

The virtual board is a VHDL file that includes all virtual wired components, external memories and a PCI9050 simplified model. Each virtual wired component can be mapped into an FPGA if the operation of the target processor is correct in the virtual board. Virtual external memories and the PCI9050 model have the same functionality with the real hardware. Especially, the virtual PCI9050 model includes C interface functions, called slang interface (Synopsys), which can communicate with the emulator program through socket programming. With the conventional VHDL simulation tools, we can trace the significant signals and detect the bugs more easily.

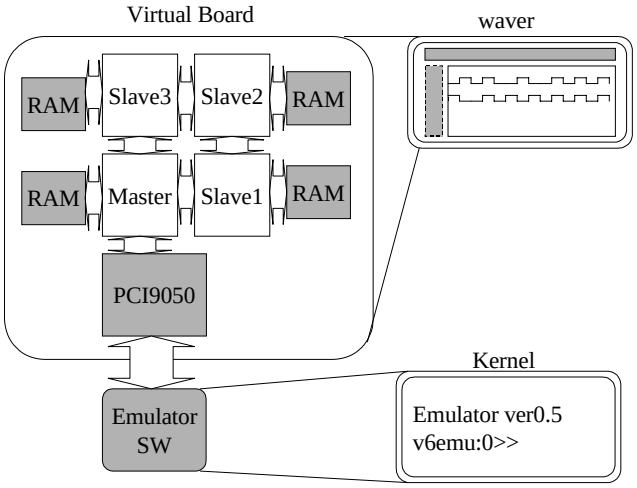


Figure 9: Virtual board

3.4 Board control function

We provide two basic board control functions, which are PCIreadData () and PCIwriteData (int addr, int data). They activate the operations explained in the section 2.2 for the debugging network. With these basic functions, we can construct the debug functions such as debugRead (int chipId, int localId, int addr) and debugWrite (int chipId, int localId, int addr, int data), which are more useful interface function.

If we insert debug modules in the target processor, we must assign chip id and local id to each debug module. We can activate a debug module by simply calling these func-

0-7695-0668-2/00 $10.00 © 2000 IEEE

tions without understanding the complicated debug procedure in detail.

For example, when the debug module of the data cache is located in slave3, whose chip id is 3, and the local id is "0", we can read the content of address x by calling debugRead (3,0,x). The control hierarchy is as Fig.10.

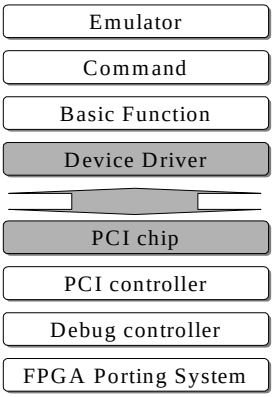


Figure 10: Board control hierarchy

4 Emulator Implementation

4.1 Feature of target processor

We designed the target processor as a core for video-telephone and implemented its emulator on this emulator environment. The basic applications are H.263 for video codec and g.723.1 for audio codec. In this emulator, we verify the kernel of each application.

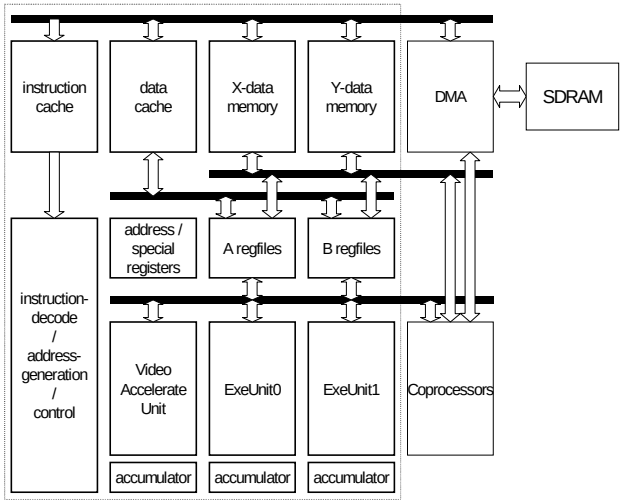


Figure 11: Target processor

Fig.11 is the block diagram of the processor. To accelerate multimedia-specific applications, the architecture of this processor has several features. It merges the features of RISC and DSP and its instruction set is extended to accelerate both video and audio applications. Furthermore, it includes X/Y memories to reduce both bandwidth and latency for multimedia applications with frequent memory accesses.

4.2 Porting procedure

First, we tried to find out the size of each sub-component and analyze the signals among them. From this information, we partitioned the multi-media processor as shown Fig.12. The value in the arrow-box is the ratio of used virtual wires to the available virtual wires. The value in the square-box is the ratio of mapped logic blocks to the total available logic blocks in a single FPGA.

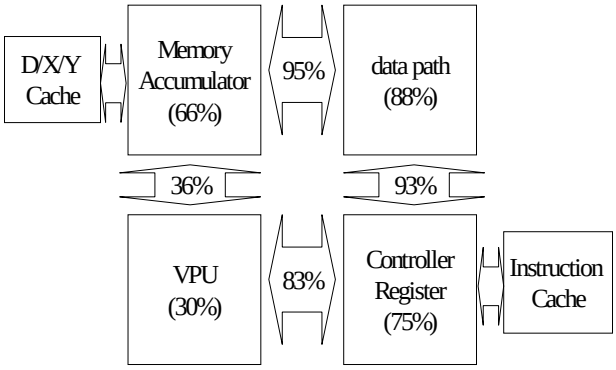


Figure 12: Partition

Second, we modified the internal memory such as instruction cache, data cache and X/Y memories to the cache-like model by using an external SRAM. These cache models use internally fast clock which is also used in debug controller, virtual wire operation and SRAM access, and are synchronized to the slow clock which is used by the target processor.

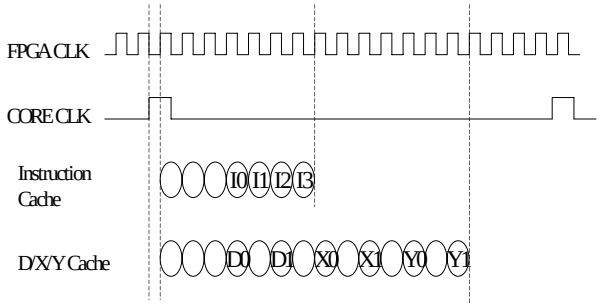


Figure 13: Cache model

Third, we ran partition generator, virtual wire generator and virtual board generator. The number of virtual wire operation phase was two and the core clock is 80 times slower than the fast clock.

0-7695-0668-2/00 $10.00 © 2000 IEEE

Fourth, we implemented a simple emulator program that included some basic commands such as program, step, view register, view memory and reset. With the virtual board and the emulator program, we tested the functionality of the processor.

Finally, by mapping the target processor into the real board and using the same emulator program, we can also test the functionality of the processor. The effective operation speed of the processor is currently 128 KHz, which can be increased with optimizing the partition.

4.3 Result

The full test of this emulation environment is now going on. We can verify the RTL of the target processor faster and reflect its changes more flexibly with this emulator. After this verification ends, we plan to upgrade the user interface that is similar to the software simulator already implemented in the early stage of the processor development.

5 Conclusion

We implemented a flexible emulator environment and verified our multi-media processor in this environment. After most verification is completed, we will release this emulator for our software developers. Currently, we are also developing an expanded FPGA prototyping board with 3x3 mesh, which includes nine FPGAs and a flash ROM, and more upgraded software and hardware environment. The target of this second version is to verify the functionality of the multi-media system with dual processors. This dual processor system is targeted to run the applications that execute a web browser and a voice recognizer concurrently.

References

[1] Sudheendra Hangal and Mike O'Connor, "Performance Analysis And Validation Of The PicoJava Processor", IEEE MICRO, May/June 1999.

[2] J. Babb. Virtual Wires, "Overcoming Pin Limitations in FPGA-based Logic Emulator", MS Degree Thesis, MIT 1994.

[3] M. Dahl, "An Implementation of the Virtual Wires Interconnect Scheme", MS Degree Thesis, MIT, 1994.

[4] Xilinx inc," Virtex 2.5V Field Programmable Gate Arrays", Technical Manual, May 1999.

[5] Duncan A. Buell, Jeffrey M. Arnold, Walter J. Kleinfelder,"Splash2: FPGAs in a Custom Computing Machine", IEEE Computer Society Press.

0-7695-0668-2/00 $10.00 © 2000 IEEE