Jurnal Teknologi, 36(D) Jun 2002: 99–110

HSP16: A HARDWARE SIMULATOR FOR PESONA 16

KHAIRULMIZAM SAMSUDIN[1], ABDUL RAHMAN RAMLI[2]
& ISMAIL MAT YUSOFF[3]

Abstract. Hardware Simulator for Pesona 16 (HSP16) is a simulated environment of the Pesona-16 microprocessor for execution of the host-code to enable parallel co-design and co-verification. The simulator core is a typical Instruction Set Simulation (ISS) model, also called Register Transfer Level (RTL) that incorporate an instruction simulation, debugging facility and devices interfacing. The simulator is developed with C and is capable of replacing the real hardware and are fully modularised. Additional microprocessor architecture and devices can be added without the need to rewrite the simulator.

Keywords: Pesona-16, Simulator, Microprocessor, RISC, Co-verification

Abstrak. Penyelaku Perkakasan Pesona 16 (HSP16) merupakan persekitaran pemproses Pesona 16 yang diselakukan untuk perlaksanaan kod perumah. Persekitaran ini membolehkan rekaan dan pengesahan sistem dilakukan secara selari. Teras penyelaku tersebut adalah model Set Arahan Penyelaku (Instruction Set Simulation) juga dikenali sebagai Pemindah Ingatan Daftar (Register Transfer Level). Penyelaku tersebut dibangunkan secara modular dengan C dan mampu menggantikan perkakasan sebenar. Penambahan senibina mikro pemproses serta peranti tambahan dapat dilakukan tanpa perlu melakukan pengkodan semula.

Kata Kunci: Pesona-16, penyelaku, mikroproses, RISC, pengecasan

1.0 INTRODUCTION

Growth in software and hardware complexity is straining existing design practices especially system verification. System verification now accounts for more than 40% of the overall design cycle, an unacceptable situation given the current market windows [1]. Verification is the crucial factor in maximizing the likelihood of first-time success. Until recently, the only really viable Hardware/Software (HW/SW) integration strategy was to bring the two components together after the hardware was built and prototyped.

[1&2] Department of Computer and Communication Systems, Faculty of Engineering, Universiti Putra Malaysia, 43400 Serdang, Selangor. kmbs@eng.upm.edu.my & arr@eng.upm.edu.my.
[3] Application Engineering, MIMOS Semiconductor, MIMOS Berhad, Technology Park Malaysia, 57000 Kuala Lumpur. ismail@mimos.my.

In the past few years, the underlying technology to support a true co-verification environment has emerged and matured. These approaches create a virtual test and integration environment, software and hardware can be developed together from the beginning. This eliminates time-consuming back-end integration and testing, that uncover problems earlier in the design process where they are less costly and easier to fix.

Co-verification becomes an extension of the available tools and technologies. Many different combinations and methods are available. These methods fall into four general categories based on the underlying technology used [2]: hardware only, logic simulation, combination of both and C simulation.

In the co-verification environment, hardware and logic simulation use existing logic simulation and debug tools. Using a co-verification technique, engineers can accurately verify hardware and software, even though they lack working silicon. Using this simulated representation of the hardware, the embedded system software can be run in two modes, host-code or target-code. Tools in the C simulation space typically target software engineers, but are more geared toward hardware engineers. Both host-code and target-code simulation can be done. Host-code mode uses the native workstation tools to compile and debug the software, while target-code mode uses the tools for the target embedded system [3].

The simulator interface provides a bridge between dissimilar domains. Interfacing techniques vary, depending on which domains are being linked. Some of the key features provided are:

- Debugging: Comprehensive debugging with an easy-to-use, full-featured graphical user interface (GUI).
- Mixed physical and virtual co-verification: Hardware and software solution that lets you add a mixed physical or virtual approach to system development and integration with live models which provide critical hardware-level accuracy.
- Model Libraries: Comes with an extensive set of virtual component libraries. For effective test-benches the user can choose from a rich set of blocks representing arithmetic, counter, conversion, data structure access, delay, execution control, traffic generator and vector access functions.
- Support system level code: System level code is implemented in operating system. Therefore, engineers can write their own device simulators, memory-management unit (MMU) and other modules to plug into the simulator.
- Profiling and Statistics: Memory system performance numbers such as CPU cycle, data cache, page table misses and instruction cache misses can be examined easily, thus summarising the usage of the microprocessor or the efficiency of the software.

The rest of this paper contains information describing the design and development of HSP16. Section 2 provides a general description of Pesona-16. For a complete description of the microprocessor architecture, please refer [4]. Section 3 mentions previous work on other microprocessor simulators. The design and implementation of HSP16 is described in Section 4. Sections 5 and 6 describe future work and conclude the paper.

2.0 PESONA-16 BACKGROUND

The PESONA16 (P16) is a 16-bit RISC microprocessor design and fabricated by MIMOS Semiconductor (MySEM). The P16 is targeted towards embedded applications ranging from controls to communications system including SCADA, remote monitoring, industrial robotics, home security, appliances, switching equipment and digital set-top boxes.

The processes and components that make up an embedded system application are much like those of the general purpose computing world hardware, software, data and networking. The key difference lies in the speed with which the embedded application must work on the data it receives to produce a result, and the reliability of that result. It has to work very quickly and it has to work every time [5].

This is where the RISC microprocessor fits in. RISC (Reduced Instruction Set Computer), which evolved in university research labs, was built to reduce instructional and operational complexity, resulting in faster cycles and faster register to ALU to register operations.

Although RISC processor have been defined and designed in a variety of ways, the key elements shared by most design are:

- A limited and simple instruction set [6][7]
- A large number of general-purpose registers
- The use of compiler technology to optimise register usage [8]
- An emphasis on optimising the instruction pipeline

Currently there is only a macro assembler (ASMP16) used to support P16 assembly language and a customized C compiler for the P16.

3.0 RELEVANT PREVIOUS WORK

Simics [12] is a complete instruction level simulator for several multiprocessors. The simulator emulates the microprocessor at the level of individual instructions thus allowing it to run unmodified programs written for target machine. System level implies that it also emulates the hardware components of the computer system. Simics is capable of profiling the program execution including data cache, instruction cache misses and act as a development tool.

SimpleScalar [9] tool set is a suite of powerful computer simulation tools that provide both detailed and high performance simulation of modern microprocessor. SimpleScalar tools are highly flexible, portable, extensible and high performance. This tool set has dominated research in computer architecture [10] including memory behaviour and performance impact, compiler architecture, microprocessor register and pipelining technique.

BSVC [11] is a Motorola 68000 microprocessor simulation framework. The simulator framework was written with C++ and TCL/TK for the graphical user interface.

4.0 IMPLEMENTATION DETAILS

4.1 Design Ideas

HSP16 is a host-code simulator that is located in the user application layer of our PC compatible system. The system interfaces is depicted in *Figure 1.* The current platform for HSP16 include Linux (x86) and Windows 9x with command line interface (CLI).

HSP16 will specifically assist development of Pesona-16 embedded system by providing integrated development, debugging facility and CPU statistic [9][12]. A key feature of HSP16 is the direct memory and register manipulation while providing a vast possibility of hardware integration to the simulator system. In general, HSP16 will have the following features:

- Simulate the PESONA16^{TM} processors.
- Runs P16 software (S19 Motorola format).
- Development kit or monitor software (SDKPC), like commands.
- Breakpoints in registers and memory.
- Interface for the P16 disassembler.
- Easy for user to add module simulating IO peripherals.

On execution, some environment-specific initialisation code is required to define the software simulation environment, and starts the connection with the hardware simulator. The hardware interface and calls to the devices modules are made by the C

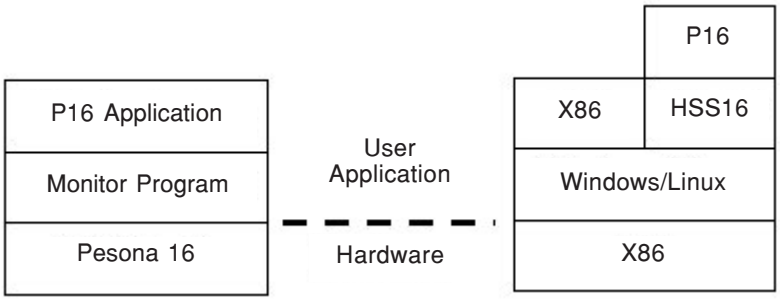

Figure 1 HSP16 System Interfaces

interfaces or application programming interfaces (API) and dereferencing the pointers [12]. Because these pointers exist in a co-verification mapped space, the access to the hardware is accomplished without the programmer having to specifically instrument the code for the co-verification environment. Interrupt handling in the host-code-mode driver was accomplished by using the interrupt API call to register an interrupt service routine (ISR) with the co-verification environment. This call specifies an ISR that will be called without further program interaction when an interrupt is detected in the hardware simulation model.

4.2 Simulator Internals

HSP16 is developed based on call-return model [13]. This is the familiar top-down subroutine module where control starts at the top of a subroutine hierarchy and through subroutine calls, passes to lower levels in the tree. From a functional viewpoint HSP16 is designed starting with a high level view and progressively refining this into a more detailed design. The simulator control structure is depicted in *Figure 2*. The data line represents data flow between each module while the control line represent control data flow to the control registers.

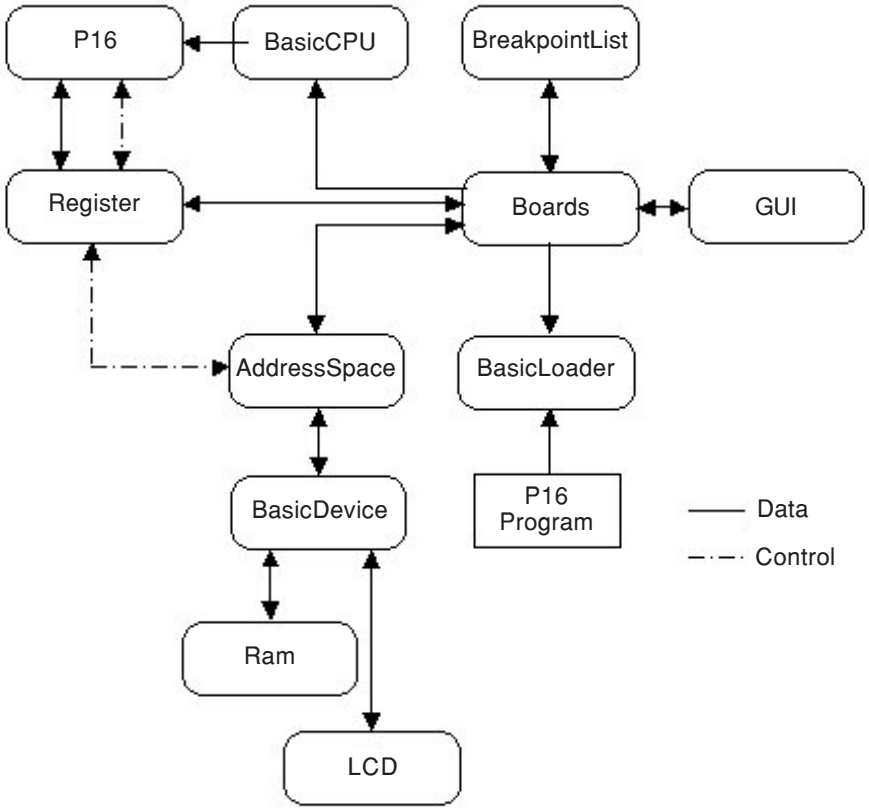

Figure 2 HSP16 Control Structure

An Instruction Set Simulator (ISS) replace the P16, ALU and CU at the core of the HSP16 as depicted in *Figure 3*. The ISS will decode and call the specific functions or macro to execute the instruction. The decoded instruction format will be implemented directly to enable decoding as much as possible with minimum work left for runtime[14]. This suggestion is based on the host architecture (x86) that is a 32-bit machine which is possible with the *mask* and *signature* variable. Each instruction can be identified as their signature is unique [15].

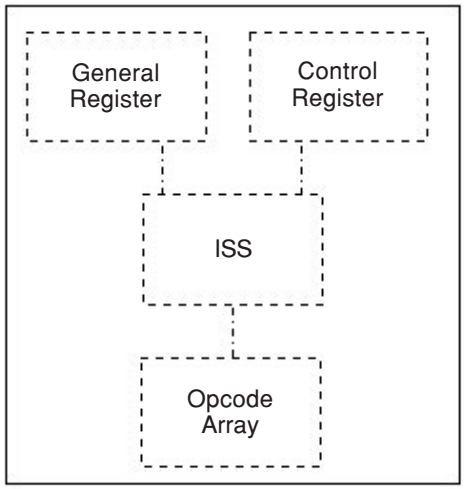

Figure 3 HSP16 Instruction Set Simulator

HSP16 CPU registers are implemented in a structure that consist of general registers array and control registers array. This ensures flexibility during code implementation. Each of the variables is implemented in a bit field structure. The general registers are divided to a high and low byte. This would simplify the operation on it. All the registers can be accessed directly by declaration in the header file. However to emulate R0 and R1 registers that is hardwired to zero and one respectively [4], the general registers must be accessed through a predetermined interface [15].

The only way to stop the execution of a program in the HSP16 is the insertion of the break Simulator Control Code (SCC). One SCC will represent each breakpoint. A new module is introduced in the exception-handling module to examine the presence of SCC. Implementation of this mechanism will enable the supports for the following breakpoint features:

- Single stepping.
- Instruction tracing – program flow can be traced after certain instruction cycle while gathering CPU or memory statistics.
- Invalid Instruction – halt the execution if a program attempts to venture beyond its bounds.
- Instruction Frequency – every instruction is counted.

The exception handler or interrupt function will handle the generated exception from HSP16 simulator accurately such as the real hardware. Further than that, simulator control codes from the HSP16 will also be handled.

Each device will be implemented with a standard device structure. Each device will also implement their own read, write and reset functions. This way, it is possible to implement just any type of simple devices such as Random Access Memory (RAM) and Liquid Crystal Display (LCD). Each device uses relative addressing that is handled internally, therefore address mapping for each device can be handled through the specified interface without the knowledge of the implementation. All devices on HSP16 will be memory mapped [16]. The clean interface between memory routines and HSP16 allow code to be added without the need to understand the internals of the simulator.

4.3 Simulator Code File Description

Each module from the design phase is allocated a different directory. Each module consists of smaller units that form a subsystem. The following list describes the functionality of the C code files in the hsp16/ directory.

- hss16.c: Performs all the initialization and launches the main simulator function.
- BasicCPU.[c,h]: Basic structure to implement a microprocessor and function to simulate a CPU cycle.
- BasicLoader.[c,h]: Load the target P16 program in S19 format to the memory.
- BreakpointList.[c,h]: Handles addition and deletion of memory and register breakpoint.
- Registry.[c,h]: Contain P16 registers structure and function to manipulate the registers.
- StatInfo.[c,h]: Build the statistic information for HSP16.
- defs.h: Contains important variable defination, initialization and function prototypes.
- p16.[c,h]: Defines function to simulate the execution of P16 instruction.
- AddressSpace.[c,h]: Act as an interface to BasicDevice. Handle creation of device, attaching, detaching and manipulation of device content. Also handle fault during fetch and store.
- BasicDevice.[c,h]: Handles creation of specific device and manipulation of device content.
- Ram.[c,h]: Defines the data structure and function to create, fetch, store and reset Ram devices.
- LCD.[c,h]: Defines the data structure and function to create, fetch, store and reset LCD devices.

- tools.[c,h]: Contain numerous useful function to handle HSP16 operation.
- xerror.[c,h]: Holds code to handle error message and status.
- xmalloc.[c,h]: Holds usefull code to wrap the standard dynamic memory management function to localise memory handling.

4.4 Constraints and Limitation

HSP16 will not support system level code because portability issues will arise. System level code includes operating system calls to other PC peripheral device such as Network Interface Card (NIC) and serial communication port (COM). However these could be addressed by modularising system calls function. These will limit the capability of HSP16 to present a system view of an operating system or device driver. However, normal P16 program can be simulated accurately.

The accuracy of HSP16 is only applied to the memory and register contents in behavioural simulation. This means, the simulator will only provide accurate instruction set execution. Correct machine cycle and pipelining are not implemented as the main focus is providing an instruction set simulation. Cache contents, which imply cache hits and misses, cannot be identified, as these elements are not acquired.

During unassembling of an instruction, HSP16 will not replace any expression, value and variable with the symbol used by the P16 assembly code. The symbol can be obtained by parsing the symbol table generated by ASMP16 (Pesona 16 Assembler) that is located in the listing file [17] with other assembler output. HSP16 also could not identify data directive (DB) and may accidentally unassemble data directive.

Initially, HSP16 will only provide a basic device library with some basic functions. However each device modules can be coded from the data sheet or model library from the manufacturer. Thus the simulated software cannot be guaranteed to work perfectly when implemented to the real hardware. A solution for this problem is to interpret spice or netlist I/O module models directly which is possible by implementing circuit simulator engine such as Berkeley's Spice or NG Spice [18] which is available in source code, and most electrical simulators inherited its syntax.

4.5 Verification and Validation

HSP16 are built out of sub-systems, which are built out of modules, which are composed of procedures and functions. During the incremental development, feedback is vital and the most basic and critical feedback is that of 'Extreme Testing' [19]. Each feature will have their own test suite that will be divided to three stages of testing process:

- Module testing
- Sub-system testing
- System testing

A module is a collection of dependent components such as data structures, procedures and functions that encapsulates related components so it can be tested without other system modules. The module is tested incrementally as it is developed. Subsystem testing involves testing collections of modules, which have been integrated into sub-systems. HSP16 sub-systems are designed and implemented independently. The most common defects, which will arise, are sub-systems interface mismatches, therefore the sub-system test process will concentrate on the detection of interface errors by exercising these interfaces.

The sub-systems are integrated to make up the entire simulator system. The HSP16 system testing process is concerned with finding errors, which result from unanticipated interactions between sub-systems and system modules. System testing is also concerned with validating that the system meets its functional requirements and features. HSP16 will be tested with real P16 program rather than simulated test data. System testing may reveal errors and omissions in the system requirements definition because the real data exercises the system in different ways from the test data. System testing may also reveal requirements problems where the system's facilities do not really meet the users needs or the system performance is unacceptable.

These test suites are important during development of HSP16 and also during enhancement of the base simulator. There are eight automated test suites included with HSP16 source that will verify and validate the requirement, specification and implementation of HSP16.

5.0 CONCLUSION

Hardware/software co design is concerned with the joint design of hardware and software making up an embedded computer system. Hardware/software co design reduces market time and resources during development of an embedded system. Development of HSP16 as an instruction simulator are concerned with these concept by featuring some of the features that could be found on some commercial solutions and research work. However HSP16 is more focused towards a development tool for Pesona-16 microprocessor.

6.0 FUTURE WORK

A GUI will provide an easy-to-use, point-and-click interface for HSP16 which allows user environment customizations and device (I/O) modules configuration. TCL/TK allows one to quickly create custom graphical applications to interface the current CLI environment without the need to delve in arcane subjects like the internal structure of HSP16.

HSP16 only provide a functional instruction simulation and instruction are executed serially, simulating no instructions in parallel. However pipelining can be implemented with threads and traces of the out-of-order pipelining can be provided. Implementa-

tion of P16 microprocessor cache enables effect of cache performance on execution time to be examined. Time accounting for the simulator can also be provided with implementation of microprocessor pipelining and cache.

System level simulator can be implemented with a proxy handler that intercepts system calls made by the simulated binary, decodes the system call, copies the system call arguments, make the corresponding call to the host's operating system and then copies the results of the call into the simulated program's memory. System level simulator will allow device level emulation, where user application is capable of using the PC peripherals such as NIC, audio card and also CD drive. The simulator can also boot an operating system with minimal modification to the source code.

ACKNOWLEDGEMENT

The author would like to thank the Universiti Putra Malaysia and MIMOS Semiconductor for providing the facilities and support throughout this research. The author, Khairulmizam Samsudin is currently a tutor in the Department of Computer and Communication Systems, UPM.

REFERENCES

[1] Ed Hunnell and Michael Lyons. 1998. *Co verification Goes From Cutting Edge To Mainstream.* Electronic Design. Jun-1998.

[2] Karl Van Rompaey. 1999. *A checklist for SOC hardware/software co design.* [Online document]. April-1999. [cited 15-August-2001]. Available HTTP: http://www.computer-design.com/Editorial/1999/04/0499a_checklist_for_system.html

[3] Jason Andrews. 1998. *Verify Hardware In A Logic Simulation Environment*. Electronic Design. Jun-1998.

[4] *16-Bit RISC Microprocessor PESONA-16 R1620*. MIMOS Berhad. 1998.

[5] Dave Bursky. 1998. *RISC And CISC Processors Compete For Embedded Applications*. Electronic Magazine. Aug-1998.

[6] Lunde, A. 1977. *Empirical Evaluation of Some Features of Instruction Set Processor Architecture*. Communication of The ACM. Mar-1997.

[7] Huck, T. 1983. *Comparative Analysis of Computer Architectures*. Stanford University Technical Report. May-1983. pg 83 – 243.

[8] Patterson, D. 1982. *A VLSI RISC.* Computer. Sep-1982.

[9] *SimpleScalar Tools Homepage*. [Online document]. January 28,1998. [cited 2-Sept-2001]. Available HTTP: http://www.cs.wisc.edu/~mscalar/simplescalar.html

[10] *Citation Detail: SimpleScalar Tool Set*. [Online document]. August 14, 2001. [cited 2-Sept-2001]. Available HTTP: http://citeseer.nj.nec.com/context/42861/0

[11] Bradford W. Mott. 1998. *BSVC, a microprocessor simulation framework*. [Online document]. Nov-1998. [cited 31-Dec-2000]. Available HTTP: http://www2.ncsu.edu/eos/service/ece/project/bsvc/www/

[12] Peter S. Magnusson. 2002. *Simics: A Full System Simulation Platform.* IEEE Computer. Feb 2002. pg 50 – 58.

[13] Ian Sommerville. 1995. *Software Engineering*. 5th Edition. Addison-Wesley Singapore.

[14] R. Bedicheck. 1990. *Some Efficient Architecture Simulation Techniques*. USENIX. Winter 1990.

[15] Khairulmizam Samsudin. 2001. *Hardware Simulator Software for Pesona 16*. Undergraduate Thesis, Faculty of Engineering, Universiti Putra Malaysia. Sept-2001.

[16] Fujimoto R. M., and W. B. Campbell. 1988. *Efficient Instruction Level Simulation of Computers*. Transactions of the Society for Computer Simulation, Vol 5, No 2, Apr-1988.

[17] *Assembler Pesona-16 User Manual*. MIMOS Berhad. 1998.
[18] Paolo Nenzi(2000). *NG-Spice Web Site*. [Online document]. [cited 21-Dec-2000]. Available HTTP: http://ieee.ing.uniroma1.it/ngspice
[19] Ronald, E. Jeffries. 1999. *Extreme Testing: Why aggressive software development calls for radical testing efforts*. [Online document]. Mar 1,1999. [cited 15-August-2001]. Available HTTP: http://www.stickyminds.com/pop_print.asp?ObjectId=2748&ObjectType=ARTCO