国 立 成 功 大 學

電腦與通信工程研究所

碩 士 論 文

支援GDB之指令集架構模擬器與其全系統虛擬平台
An Instruction Set Simulator with GDB Support and its Full System Simulation Virtual Platform

研 究 生： 李信穎　　Student： Shin-Ying Lee

指導教授： 陳中和　　Advisor： Chung-Ho Chen

Institute of Computer and Communication Engineering

National Cheng Kung University

Thesis for Master of Science

July 2010

中 華 民 國 九 十 九 年 七 月

國 立 成 功 大 學
碩 士 論 文

支援GDB之指令集架構模擬器與其全系統虛擬平台

研究生：李信穎

本論文業經審查及口試合格特此證明

論文考試委員：

陳中和	[illegible]
黃穎聰	[illegible]

指導教授：陳中和

所　　長：詹寶珠

中 華 民 國 九 十 九 年 七 月

An Instruction Set Simulator with GDB Support and its Full System Simulation Virtual Platform

by Shin-Ying Lee

A Thesis Submitted to the Graduate Division in Partial Fulfillment of the Requirement for the Degree of Master of Science

at

National Cheng Kung University
Tainan, Taiwan, Republic of China
July 2010

Approved by:

[illegible] [illegible]

[illegible] [illegible]

Advisor(s):

[illegible]

Director: [illegible]

支援 GDB 之指令集架構模擬器
與其全系統虛擬平台

研究生：李信穎　　　　指導教授：陳中和

國立成功大學電腦與通信工程研究所

摘要

在晶片系統的開發過程中，如何在全部的硬體裝置開發完成前即進行系統軟體的開發以及軟硬體的協同模擬與協同驗證，是晶片系統開發人員一直以來所面臨的一大挑戰。

在本論文中，我們利用 SystemC 模組實現了一個基於 ARM 架構的指令集模擬器與其全系統虛擬平台。此 SystemC 虛擬平台提供了功能準確性以及時間準確性的全系統模擬環境。藉由此 SystemC 虛擬平台，系統開發工程師能夠很容易地對整體晶片系統（包含：硬體裝置、作業系統、驅動程式、以及應用程式…等部件）進行協同模擬、協同驗證、系統評測與演算法分析的工作。除此之外，此虛擬平台亦內建了 GDB 遠端除錯協定的通訊通道。透過此遠端除錯通道，SystemC 虛擬平台可直接與 GDB 除錯器進行連接，便於軟體工程師利用此虛擬平台和我們所修改擴充的 naked GDB 除錯器於系統開發先期即開始進行各種系統軟體與應用程式的開發及除錯工作，以達到有效地縮短整體晶片系統開發時程的目標。

關鍵字：虛擬平台、全系統模擬、協同模擬、系統除錯

An Instruction Set Simulator with GDB Support and its Full System Simulation Virtual Platform

Student：Shin-Ying Lee Advisor：Chung-Ho Chen

Institute of Computer and Communication Engineering
National Cheng Kung University
Tainan, Taiwan

<u>Abstract</u>

When developing a system-on-a-chip (SoC) embedded system, how to develop the system software as well as co-verify the hardware and software before all hardware modules are available is usually a big challenge for engineers.

In this thesis, we have implemented a virtual platform with an ARM-based instruction set simulator in SystemC. This virtual platform provides a functional and/or approximate-timed accurate full system simulation environment. By this SystemC virtual platform, SoC developers are able to co-simulate, co-verify, evaluate, and analyze the whole SoC system including hardware devices, OS kernel, device drivers, and application programs…etc., in a simple way. Also, we have provided a GDB RDP communication channel to connect the virtual platform and GDB debugger directly. Through this virtual platform and the naked GDB debugger which we modify from GDB, software engineers can easily develop and debug the system programs in the early development stage. Thus, the time-to-market of a new SoC design can be reduced significantly.

Keywords: virtual platform, full system simulation, co-simulation, system debugging

Contents

List of Figures

List of Tables

Chapter 1 - Introduction

1.1 Motivation

As the improvement of the process technology of very large scale integrated (VLSI) circuit, in recent years, a system-on-a-chip (SoC) design becomes more and more complicated. The task of algorithm validation, system evaluation, hardware verification, and software debugging also becomes a big effort for engineers to design a vast computing system. How to make this heavy job easier is a very important issue for SoC developers.

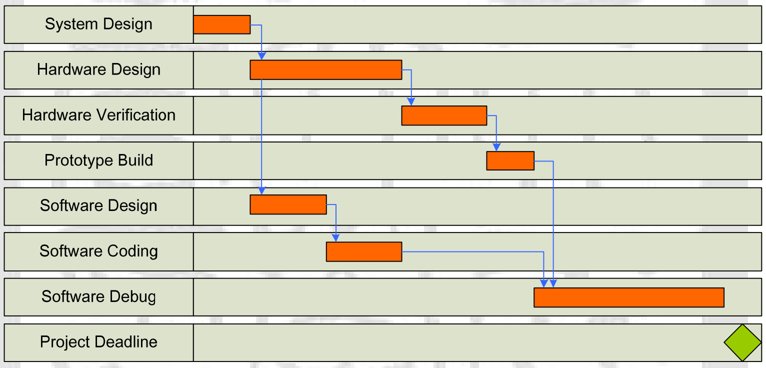


Fig. 1-1: Project schedule of traditional SoC design flow

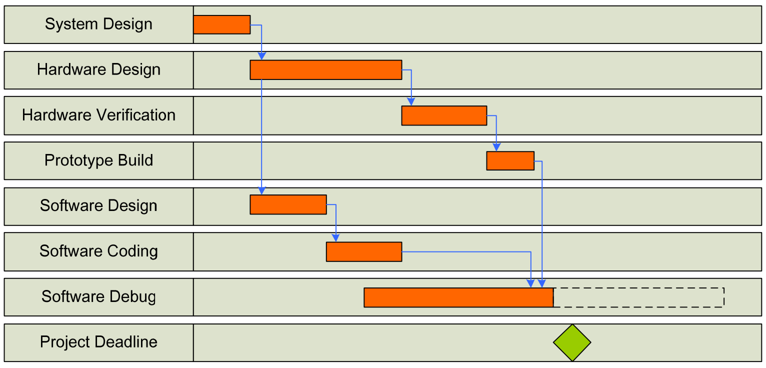


Fig. 1-2: Project schedule of ESL design flow

In traditional SoC design flow, to develop, verify, and debug the software programs before all hardware devices are available is a very difficult work when building a new SoC design. This is because the hardware modules don't have a robust and suitable testbench to validate until their corresponding software systems are ready to use. That is, the development of software and hardware are hard to advance and the time-to-market will be delayed as Fig. 1-1 shows [11].

Nowadays, as Fig. 1-2 shows [11], the electronic system level (ESL) design methodology becomes a popular way to reduce the complexity of system development and decrease the time-to-market. Following the ESL design flow, developers can co-verify both hardware and software easily. Nevertheless, it is still a difficult job to do full system simulation involving in the operating system (OS), hardware device and its corresponding device driver, as well as the application programs in the early development stage, i.e. before the synthesizable register-transfer level (RTL) code is ready.

In short, to achieve verifying and evaluating the entire SoC system in the early development stage, an efficient and robust full system simulation instruction set simulator (ISS) supporting of bare-level program (a program runs without any OS sustaining) debugging is intensely required for ESL design methodology.

1.2 Contribution

In this thesis, we have built a robust ARM-based ISS and SystemC virtual platform which has the following features:

- supporting full system simulation in approximate-timed/functional accuracy with unmodified OS kernels

- supporting debugging for bare-level programs, e.g., OS kernel and bootloader, with the GNU debugger (GDB)
- supporting system profiling, evaluating, analyzing, and validating of an SoC design with the proposed ESL mechanism

1.3 Scope and Organization

The rest of this thesis is organized as follows: Chapter 2 takes a brief introduction to the concepts and issues about the instruction set simulators and GDB debugger; Chapter 3 presents the system framework of our ISS and SystemC virtual platform design; Chapter 4 discusses how to validate this full system simulation virtual platform; Chapter 5 shows the experimental results. Finally, Chapter 6 gives the conclusions of this thesis.

Chapter 2 - Background and Related Works

2.1 Instruction Set Simulator

An instruction set simulator (ISS) is a tool that runs on a host machine to mimic the behavior of running a program on a target CPU, that is, it can execute the target binary code as a real CPU does [28][29][30]. Through the ISS, engineers can explore the CPU architecture and validate the compiler design easily. Moreover, the ISS is also the pivotal component for a full system simulation virtual platform. Thus, how to design and implement a powerful and robust ISS is a significant theme we have to explore in this thesis.

Basically, there are three types of ISS simulation method which are interpretive simulation, static compiled simulation, and dynamic compiled simulation. First we give an introduction of these ISS simulation frameworks.

2.1.1 Interpretive Simulation

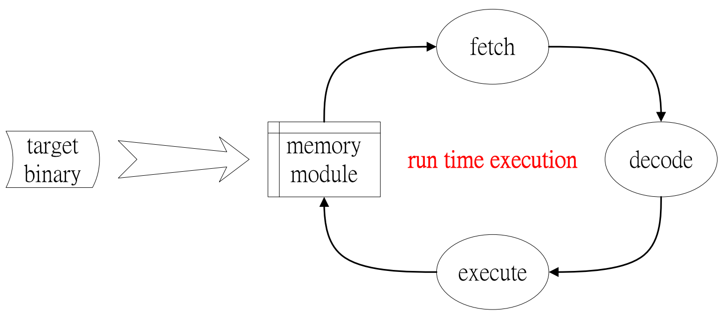


Fig. 2-1: Interpretive simulation framework

Fig. 2-1 shows the interpretive simulation framework of an ISS. Similarly as a real processor's datapath, an interpretive simulation ISS regularly works in sequence with three stages: fetching, decoding, and executing for every input target instructions as Fig. 2-1 illustrates. Because all input target instructions have to be decoded repeatedly in run time as routine, the throughput of an ISS designed in interpretive simulation is usually pretty poor. Regardless, an interpretive simulation ISS allows the developers to investigate the system design and can show up many details of program executing in run time, e.g., the timing information.

2.1.2 Static Compiled Simulation

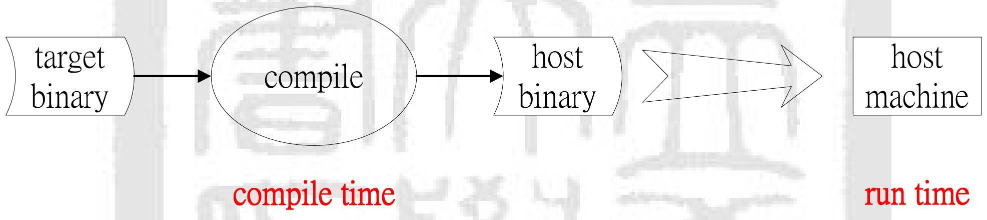


Fig. 2-2: Static compiled simulation framework

Static compiled simulation uses a translator or a specific compiler to interpret the entire target binary code as the host machine code by a one-to-one mapping technique before simulation. In this way, no matter how many times we want to do the simulating task, the ISS only has to translate the target binary once. Because the ISS doesn't need to decode the target instructions in run time, the simulation performance can be improved substantially. Fig. 2-2 shows the mechanism of a static compiled simulation ISS.

Because a static compiled simulation ISS translates the entire target binary before run time, the behavior of the simulated program must be predictable. That is, it cannot involve to a complex OS kernel if it will dynamically mount and manage lots of device drivers and application programs into memory at run time, e.g., Linux. A static compiled simulation ISS can be only used for application program simulations.

2.1.3 Dynamic Compiled Simulation

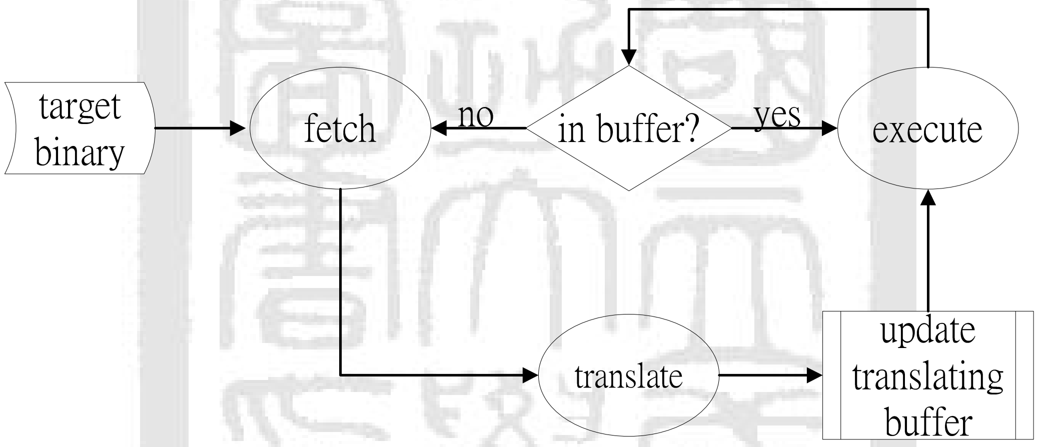


Fig. 2-3: Dynamic compiled simulation framework

Like the static compiled simulation framework, dynamic compiled simulation which is also known as dynamic binary translation interprets the target binary code as host machine binary to improve the performance of the ISS. The difference to static compiled simulation is that it translates the target binary by the unit of block (which is composed by a few of target instructions) in the run time dynamically and stores the translation result temporarily into a local translating buffer rather than in pre-run time.

Through its dynamical property, a dynamic compiled simulation ISS can be applied to simulate not only application programs but also OS kernels.

As Fig. 2-3 shows, for dynamic compiled simulation framework, the ISS doesn't fetch the target instruction after executing an instruction. In fact, it fetches, translates, and executes a block of instructions once a time instead of one by one. In this way, the executing overhead will be reduced when repeatedly executions occurring in the same translation block.

In theory, the performance of a dynamic compiled simulation ISS is usually worse than a static compiled simulation one because of the translation overhead. Note that, the static compiled simulation framework doesn't have to translate the target instructions in the run time.

2.2 GNU Debugger

2.2.1 Introduction to GDB

A debugger is a software tool that helps software engineers to find out bugs resided in a program. A debugger might allow the programmers to examine the executing sequence of the debugged program.

GNU debugger (GDB) [17][33] is a well-known and widespread open source debugger for software debugging within GNU POSIX development environment. Since it's flexible to cross fit on many types of processor architecture, GDB is popularly used for embedded system developing.

Current GDB has already provided the functions of break point and step execution to control the program executing flow. Also, it has the watch point functionality, general registers' values probing, and calling stack inspecting (or called as backtrace) to monitor the memory system. Unfortunately, the memory space which GDB sees is virtual address space only and GDB has no mechanism to change the setting of the memory management unit (MMU) and/or co-processors, so that it is difficult to apply GDB to debug a bare-level program which might control and manage the system resources through the MMU. Indeed, the original GDB program is designed for debugging application programs only instead of bare-level programs.

2.2.2 Remote Debugging Protocol

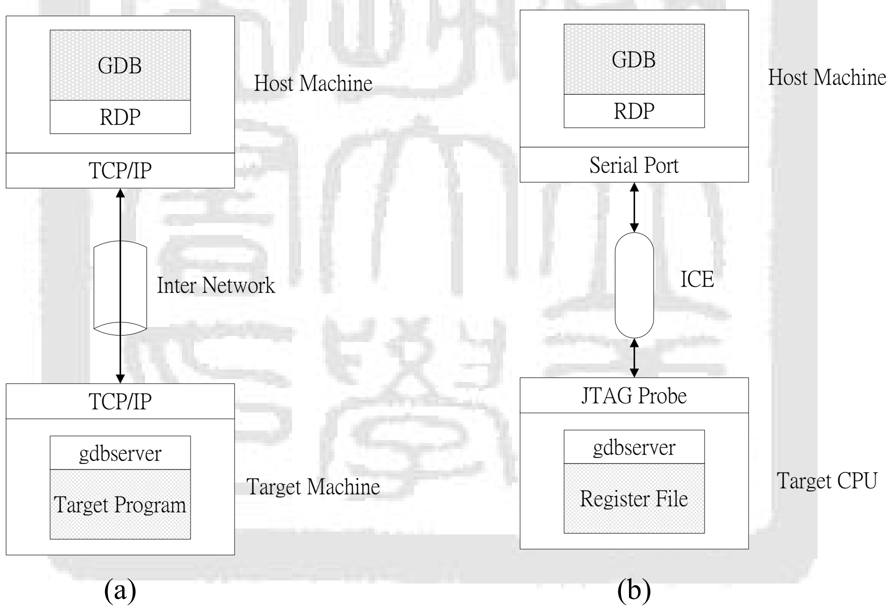


Fig. 2-4: RDP connection

(a)via Inter Network (b)via JTAG and ICE

For cross developing an embedded system, the debugged program is usually executed on a remote machine, e.g., a development board. The remote debugging protocol (RDP) defined by GDB is a protocol for communicating with target programs inside a remote machine. Through RDP, the host machine (where the GDB executed on) and the target machine (which the debugged program runs on) have the ability to connect each other via TCP/IP network as Fig. 2-4(a) shows. Furthermore, the debugged target program requires an extra module, called gdbserver or gdbstub, to parse and pack up the RDP data packet. In most cases, both the TCP/IP protocol stack and gdbstub are furnished by an OS, i.e., it's difficult to debug bare-level programs such as bootloader and OS kernel itself. Again, GDB is originally concerned to debug only application programs running on an OS, Linux, for example. This remote debugging scheme is not useful for embedded system development at the early stages.

```
→ $command/data #checksum

← +
```

Fig. 2-5: RDP packet format

In some cases, as Fig. 2-4(b) shows, GDB uses serial port to link with the target CPU directly by means of JTAG (joint test action group) probe and ICE (in-circuit emulator) for bare-level software debugging and hardware circuit testing. In this way, it is easy to scan and monitor the register file and memory system of the target CPU; nonetheless this scenario has a disadvantage, that is, the target system has to carry out the gdbstub by some additional hardware circuit. However, in this way, the cost of the target system is increased significantly.

Fig. 2-5 describes the RDP data packet format. Each data packet of RDP starts with a '*$*' sign and finishes by a '#' symbol following an 8-bit checksum value. No matter the GDB or gdbserver, after receiving a RDP packet with correct checksum value, they ought to immediately response a '+' symbol for acknowledgement. Again, all RDP packets are transferred by TCP/IP or the serial port.

2.3 Related Works

2.3.1 Simplescalar

Simplescalar [13] is an open source and very famous (be cited by nearly 2000 times) interpretive ISS that is extensively used in areas of computer architecture research and compiler designing. It provides the developers a CPU prototype to examine a new processor architecture design as well as to evaluate its performance.

Simplescalar executes target programs in cycle accurate model, but simulates at a very low speed. Besides, because Simplescalar doesn't have any I/O peripheral interface, it can only simulate with specific applications but an OS kernel which is necessary to run with peripheral device, e.g., timers, interrupt controllers, and keyboard interfaces…etc.

2.3.2 FaCSim

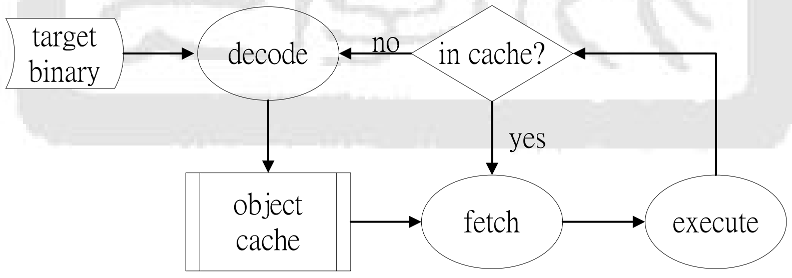


Fig. 2-6: Object cache methodology

In [22], it improved and accelerated the decoder module of interpretive ISS by an object cache technique. Similarly like the dynamic binary translation framework, it records some of the instructions that have been decoded and executed. Hence, it can reduce a great amount of instruction decoding overheads during simulation time. The difference to dynamic binary translation in this framework is that it interprets the target binary instruction one by one rather than block by block. Fig. 2-6 shows the concept of the object cache methodology.

FaCSim is a full system simulation virtual platform established in pure C/C++ language and optimized for multi-core host machines with the object cache ISS mechanism. The ISS of FaCSim is time accurate but the I/O peripherals are modeled with only functional accuracy. Consequently, because it is implemented by native C/C++, the developers are difficult to attach new hardware devices with the virtual platform.

2.3.3 Dynamic Binary Translation

In [31], the work proposed the concept of dynamic binary translation. Nowadays, there are many ISSs and virtual machines by means of the dynamic binary translation methodology in order to obtain a better simulation throughput.

QEMU [14][26] is a famous virtual machine which is implemented by dynamic binary translation in pure C/C++. In QEMU, it applies the core of gcc compiler to re-generate and optimize the target binary code.

Recently, QEMU is popularly applied to develop and emulate OS kernels of embedded systems from its high performance. Unfortunately, like most of binary translation frameworks, QEMU is functional accurate but timed accurate. The system developers cannot examine the interaction between the hardware and software with the QEMU virtual machine. Moreover, it is implemented in pure C/C++ language; the developers are hard to bind a new hardware device with QEMU as well. For the above reasons, we don’t attempt to apply QEMU to do full system simulation in ESL design methodology.

2.3.4 SimIt-ARM

In [20], the work proposed a virtual machine called SimIt-ARM implemented by both interpretive model and dynamic binary translation. SimIt-ARM has already supported an instruction count metric. Even though it has an instruction count metric, developers still cannot probe the timing information like a common dynamic binary translation framework. Again, it is not powerful enough to be used as a virtual platform in an ESL design flow.

2.3.5 Hybrid Compiled Simulation

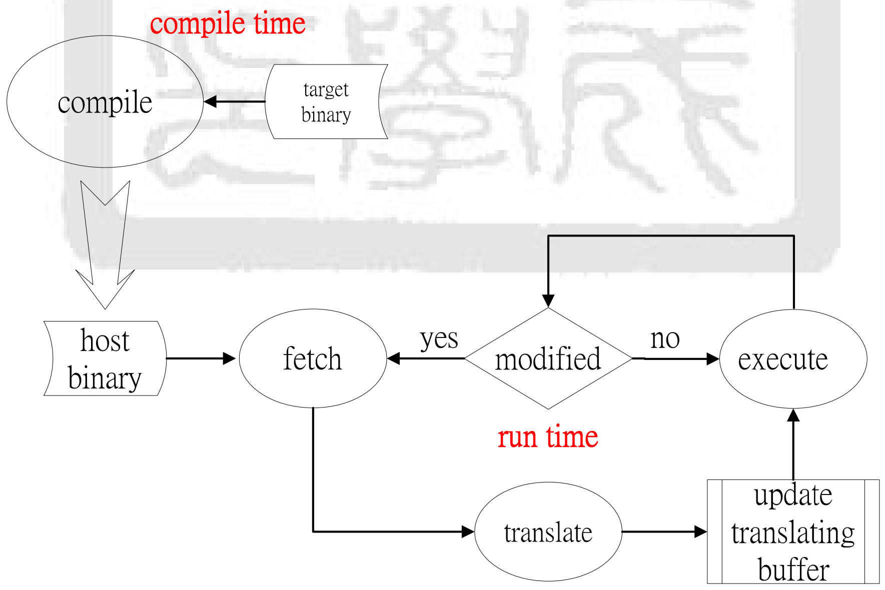


Fig. 2-7: Hybrid compiled simulation framework

In [28][29][30], they have proposed a hybrid compiled simulation framework to associate the advantages of both static compiled simulation and dynamic compiled simulation. Fig. 2-7 shows the framework of hybrid compiled simulation.

The hybrid compiled simulation framework translates the target binary before run time the same as a static compiled simulation framework. Yet, during run time, it will monitor whether the code segment in the memory is modified or not. If modified, the ISS will re-generate and update the host machine code. This framework is more flexible than a static compiled simulation ISS and more efficient than dynamic compiled simulation one, but it still cannot show up the timing information for system engineers.

In practice, to save the cost of storage devices, a lot of embedded systems, especially for portable devices, store the OS kernel image in a compressed form and decompress the image at boot up time. To simulate system like this case, the hybrid compiled simulation mechanism will not gain much benefit because it probably cannot translate the compressed image. It is not good enough for doing full system simulation with ESL design methodology.

2.3.6 Simics

Simics [10][23] is a commercial ESL tool for doing full system simulation. Recently, it goes popular in system design domain, but it is not open source and free for using. Therefore, the developers cannot easily investigate and modify the whole system design. Namely, it might not be flexible enough to explore the entire system for system developers.

Chapter 3 - System Framework

3.1 Emulation Methodology

In this thesis, the main goal we aim to and look forward is to design a full system simulation environment which can co-simulate and co-verify hardware devices as well as software programs including OS kernel (Linux for our example) for ESL design methodology. To design a full system simulation virtual platform and its corresponding ISS which can emulate the OS kernel with acceptable time duration, there are two major issues we have to trade off first, that is, the simulation accuracy and the simulation performance.

3.1.1 The Accurate Model

As Fig. 3-1 shows, the relationship between the accurate model and the simulation performance is that higher simulation accuracy usually makes lower simulation speed and vice versa.

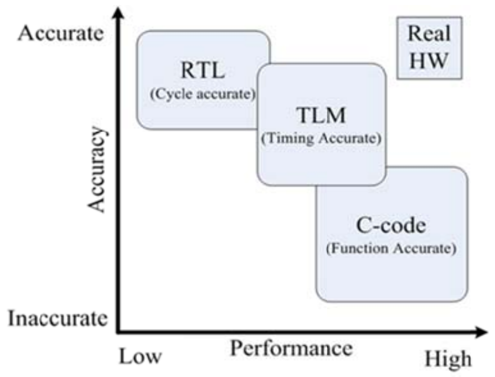


Fig. 3-1: Trading off between accuracy and performance

TABLE— 3-1: Abstract level of simulation accuracy

Abstract level		Features	Throughput
functional (untimed)		only mimic the functional behavior	very high
timed	**approximate-timed**	simulate the action during a specific period	high
	cycle	simulate the actions for each clock cycle	slow
pin		simulate the signal transferring on the wires for each clock cycle	very slow

TABLE— 3-1 shows the characteristics of different abstract models of simulation accuracy [18].

For the functional accurate model, e.g., binary translation scenario, it normally has the best simulation performance comparing with the other abstract models, but it is difficult to verify and investigate the interactions between software and hardware by means of a functional accurate virtual platform. On the other hand, a pin accurate virtual platform, such as RTL models, always simulates at a very low speed which is not suitable for running with an OS kernel.

To develop the proposed full system simulation virtual platform and its ISS, the policy of approximate-timed accurate model in SystemC [8][11][18] is chosen to achieve for the following reasons:

(1) First of all, timed accurate hardware models can be applied to profile the entire system including both hardware and software easily.

(2) Secondly, an approximate-timed accurate virtual platform can be simulated at a much higher speed than cycle accurate and pin accurate models, so that the amount of time to emulate booting up an OS is acceptable.

(3) The system developers can try, adjust, and determine the system parameters simply within SystemC simulation framework.

(4) Finally, SystemC is an IEEE standard used for building hardware modules; as a result, the hardware developers are able to modify and build up this virtual platform with additional hardware devices painlessly. Moreover it also allows developers to advance the hardware modules to become cycle accurate if needed.

3.1.2 SystemC Simulation Methodologies

TABLE— 3-2: Comparison of SystemC simulation scheme

	SC_CTHREAD	***SC_THREAD***	***SC_METHOD***
executing trigger	clock edge	signal events	signal events
executing suspend	yes	yes	no
infinite loop	yes	yes	no

resume from suspending	*wait()*
	wait_until()
		wait()	N/A

According to the IEEE 1666 standard [8], there are three categories of simulation process which are *SC_CTHREAD*, *SC_THREAD*, and *SC_METHOD* in SystemC simulation kernel. TABLE— 3-2 lists the comparison of these three kinds of simulation scheme.

The main difference between *SC_CTHREAD*/*SC_THREAD* and *SC_METHOD* is *SC_CTHREAD*/*SC_THREAD* support executing suspending and resuming. In practice, this property makes the developers easy to implement a virtual hardware module especially for circuits which have the pipeline scheme. Moreover, the transaction-level modeling (TLM) standard is also defined based on *SC_CTHREAD* simulation process. Although applying *SC_CTHREAD* or *SC_THREAD* to model the behavior of hardware modules is more convenient and easier since they support executing suspending and resuming by the assigned SystemC events, their simulation performance is not as good as *SC_METHOD* [15].

SC_CTHREAD/SC_THREAD	***SC_METHOD***
counter::counter(sc_module name):	*counter::counter(sc_module name):*
sc_module(name)	*sc_module(name)*
{	*{*
SC_CTHREAD(run_thread, clk.pos())	*SC_METHOD(run_method);*
}	*sensitive << clk.pos();*
counter::thread(void)	*}*
{	*counter::method(void)*
do	*{*
{	*count++;*
count++;	*}*
wait();	
}while(1);	
}	

Fig. 3-2: SystemC simple counter module

Here we utilize a simple counter module in SystemC v2.2.0 to exam the upper bound of the simulation performance for each scheme on an Intel Core 2 Q9500 machine as Fig. 3-2 shows. From TABLE— 3-3, we see the upper bound of simulation performance is about 4.4 million clock cycles per second by *SC_METHOD*. In contrast, using the *SC_CTHREAD* scheme only reaches 3.0 million clock cycles per second. Here we observe that using *SC_METHOD* will gain a speedup of around 1.5 times faster than using *SC_CTHREAD*. This result implies *SC_METHOD* is a better option to implement the SystemC virtual platform.

TABLE— 3-3: Performance of SystemC v2.2.0 on Intel Q9500

Scheme	**Million cycles / sec**	**Speedup comparing with *SC_CTHREAD***
SC_CTHREAD	3.0	1.00x
SC_THREAD	3.3	1.10x
SC_METHOD	4.4	1.47x

3.2 ARM-Based Instruction Set Simulator

In this work, we model the interpretive ISS founded on ARMv5 architecture [5] in SystemC. This SystemC ISS design involves in datapath, MMU, and exception handlers. Now we are going to take a brief introduction to the SystemC ISS in the following Sections.

3.2.1 Datapath

As a general interpretive ISS, the SystemC ISS has three stages to perform when executing a target instruction like we described in Section 2.1.1.

For using the approximate-timed model to improve the throughput, we don't precisely model the pipeline architecture. The actions of fetching, decoding, and executing are accomplished at a time instead of step by step for each clock cycle. The ISS will calculate the total time spent and decide how many clock cycles to rest after executing one target instruction. In fact, the SystemC ISS has totally four states including an extra rest state as Fig. 3-3 shows. The rest state indicates that the ISS has to add delays for an appropriate period to model the execution time. Fig. 3-4 describes the pseudo code of this ISS mechanism.

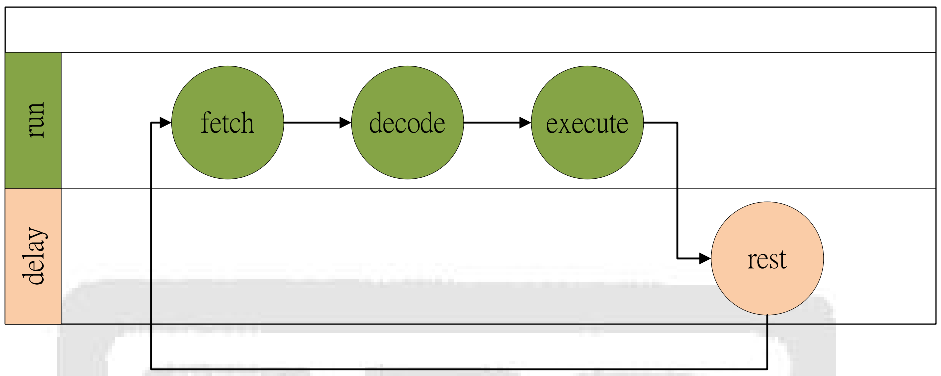


Fig. 3-3: State machine of the SystemC ISS

```
ISS::ISS(sc_module name): sc_module(name)
{
        SC_METHOD(datapath);
        sensitive << clk.pos()
}
ISS::datapath()
{
    ……
    if(!rest)
        fetch()
        decode()
        execute()
        rest = cycles in execution
    else
        pause()
        rest--
    ……
}
```

Fig. 3-4: Pseudo code of the datapath state machine

3.2.2 Memory System

The SystemC ISS has a complete MMU/co-processor system in compatible with the ARMv5 architecture. This MMU module involves in translating the virtual memory address into physical address and check the memory permission for every instruction fetching and data access.

The MMU module also has a separate I-cache and D-cache within round robin replacement policy. Like the datapath module, we don’t actually access instruction and data through the cache model to simplify the ISS design for better simulation throughput. These two “virtual cache” here are used only for system profiling and making the timing information more precisely.

For ARM processors, the I/O architecture is memory mapped I/O. All I/O peripherals are abstract to the ISS. On the virtual platform, the SystemC ISS treats these I/O devices as a universal memory module. This framework let the developers design the I/O peripheral devices without any need to concern with the ISS structure abstractly.

3.2.3 Exception Handlers

The SystemC ISS supports exceptions generated by internal and external sources consisting with all of the seven exception types of an ARM processor. The ISS will check the exception state at every clock cycle. As an exception occurs, the ISS switches to the corresponding executing mode and forces the program counter (register 15) pointing to the address of exception table automatically.

3.3 Naked GDB

One of our main goals is to debug both bare-level programs and application programs by the virtual platform and GDB in this thesis.

Because GDB is designed for debugging with only application programs originally as we have mentioned in Section 2.2, it has no available scheme to explore the MMU and/or co-processors of an ARM processor. Furthermore, the supervisor mode operations are not allowed to be performed by GDB. Consequently, it's apparently not fit for debugging a bare-level program, especially for an OS kernel which has to access the co-processors and perform supervisor mode operations frequently. For the above reason, the original GDB design requires fixing and enhancing to be adequate for the use in debugging bare-level programs.

3.3.1 The Virtual Platform with GDB

To use GDB, a gdbstub/gdbserver program embedded in the ISS is definitely necessary. The purpose of this embedded gdbstub is to work as an RDP parser and a JTAG scanner virtually. We use the embedded gdbstub to replace a real ICE/JTAG circuit, so that the debugged program has no need to run upon an OS. Namely, developers can debug bare-level programs on the virtual platform without an embedded scan chain circuit support.

Fig. 3-5 is the scenario of the gdbstub residing in the ISS. The gdbstub has two components, the RDP parser/packer (which is used to parse and pack up the RDP command packet) and the scanner (which is used to fetch the registers' values and data stored in the memory system). The connection between gdbstub and GDB is

TCP/IP socket as the figure shows, so that the GDB and the virtual platform can run on different host machines and make the debugging job more flexible.

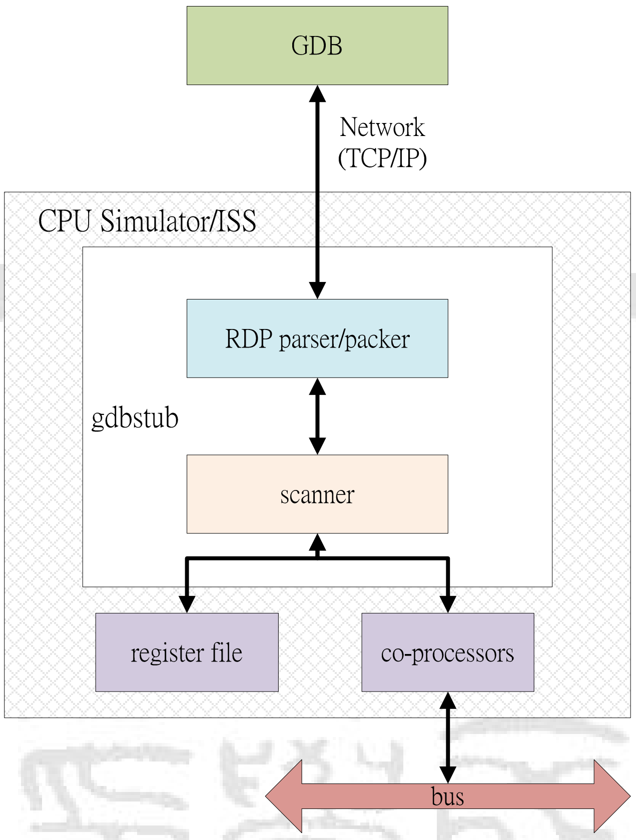


Fig. 3-5: gdbstub scenario with the ISS

3.3.2 Co-processor Probing

A bare-level program often manages system resources by MMU/co-processors. If there is any error existing in the MMU setting, it might cause the system going to crash quickly even if it's just a minute bug. Unfortunately, this kind of programming bugs is very difficult to find out and always makes software developers puzzled.

To solve this critical problem, we have to propose a method for developers reviewing the MMU setting. We add the co-processor specific registers of ARM processor into the architecture descriptor of the GDB program as TABLE— 3-4 lists. By the new modified GDB program named as the naked GDB and the embedded gdbstub in the ISS, developers are able to monitor the system status during simulation time. Thus, the software designers can investigate the MMU/co-processors settings and repair the system program when a system failure occurs.

TABLE— 3-4: List of co-processor registers of ARM

Register	**Description**
pid (context)	stores the process ID of current executed process
sys	the system control register of the co-processor 15 (MMU)
ttbr	the page table base address
domain	the memory access domain control register
dfsr	stores the data access abort status
ifsr	stores the instruction pre-fetch abort status
far	contains the data/instruction abort address

3.4 Power Estimation of the Memory System

Another issue we are interesting in when developing a new SoC design is the energy consumption, especially to design a portable device, e.g., a smart phone. According to [25], the energy consumption of the memory system is about 17% to 20% for an ARM-based embedded system. Therefore, the memory system is one of the major parts of energy consumption for an ARM SoC. In this section, we focus on discussing how to estimate and calculate the energy consumption of the memory system by the SystemC virtual machine.

Typically, the energy consumption of the memory system has three major parts which are:

(1) leakage power

the static operating power of the transistors and capacitors

(2) refreshing power

the power dissipation during the DRAM memory cell performs data refreshing

(3) switching energy

the energy consumption when the logic level switching

For a memory module, in general, almost of the logic level switching occurs at data storing and loading, that is, the total amount of switching energy dissipation extremely depends on the numbers of data access.

$$\sum E \approx tP_L + tP_R + a_{str}E_{str} + a_{ldr}E_{ldr} = \frac{c}{f}(P_L + P_R) + a_{str}E_{str} + a_{ldr}E_{ldr} \qquad \text{Equ. 3-1}$$

If we sum up the leakage power, refreshing power, and load/store switching energy, we can get to estimate the energy consumption of the memory module of an SoC design. Equ 3-1 is the approximate formula of total energy consumption for a memory module where

- P_L is the leakage power
- E_r represents the refreshing power
- E_{str} and E_{ldr} are the switching energy of data store and load
- a_{str} and a_{ldr} are the numbers of access of data store and load

- c is the total cycle count during executing
- f is the operating clock rate

Note that, the refreshing power (E_r) of a SRAM module, e.g., I-cache and D-cache, are nearly none that we can omit it.

Since P_L, P_R, E_{str}, and E_{ldr} are constant value depend on process technology, we can estimate the energy consumption of the memory system just by recording the total executing time/cycles and numbers of data accessing including I-cache, D-cache, and DRAM module through the SystemC virtual platform.

This power estimation methodology gives the system engineers an elementary way to predict the energy consumption of an algorithm in the early development stage. It can help the developers to decide and choose which algorithm or policy to be used for an SoC design appropriately.

3.5 The SystemC Virtual Platform

3.5.1 Platform Overview

Using SystemC *SC_METHOD*, we have established a full system simulation virtual platform in compliance with the ARM Versatile-PB [7]. TABLE— 3-5 lists the components on the virtual platform we have implemented. Fig. 3-6 shows the diagram of the full system simulation environment.

On the virtual platform, we have already provided some virtual I/O interfaces, e.g., keyboard, console, and LCD panel. Therefore, the program developers can emulate and verify the user interface on this virtual platform. In addition, since the virtual platform is implemented by IEEE SystemC standard, the hardware developers are able to attach new intellectual property (IP) modules and their corresponding device drivers for the virtual platform to co-simulation.

TABLE— 3-5: Components of the virtual platform

Module	Description
ARM926 Processor [3] ● ALU ● register file ● MMU/co-processors ● gdbstub	an ARM v5 compatible ISS which includes arithmetic and logic unit (ALU), register file, MMU/co-processors, and the embedded gdbstub
DRAM module	the main memory module
PL011 UART [6] ● keyboard interface ● virtual console	the UART controller with a console emulator and a keyboard interface
PL110 VGA controller [1] ● virtual LCD panel	the LCD device with an LCD panel emulator
PL190 VIC [4]	the vector controller module with totally 32 interrupt channels
SP804 dual timer [2]	a 32-bits wide system counter/timer

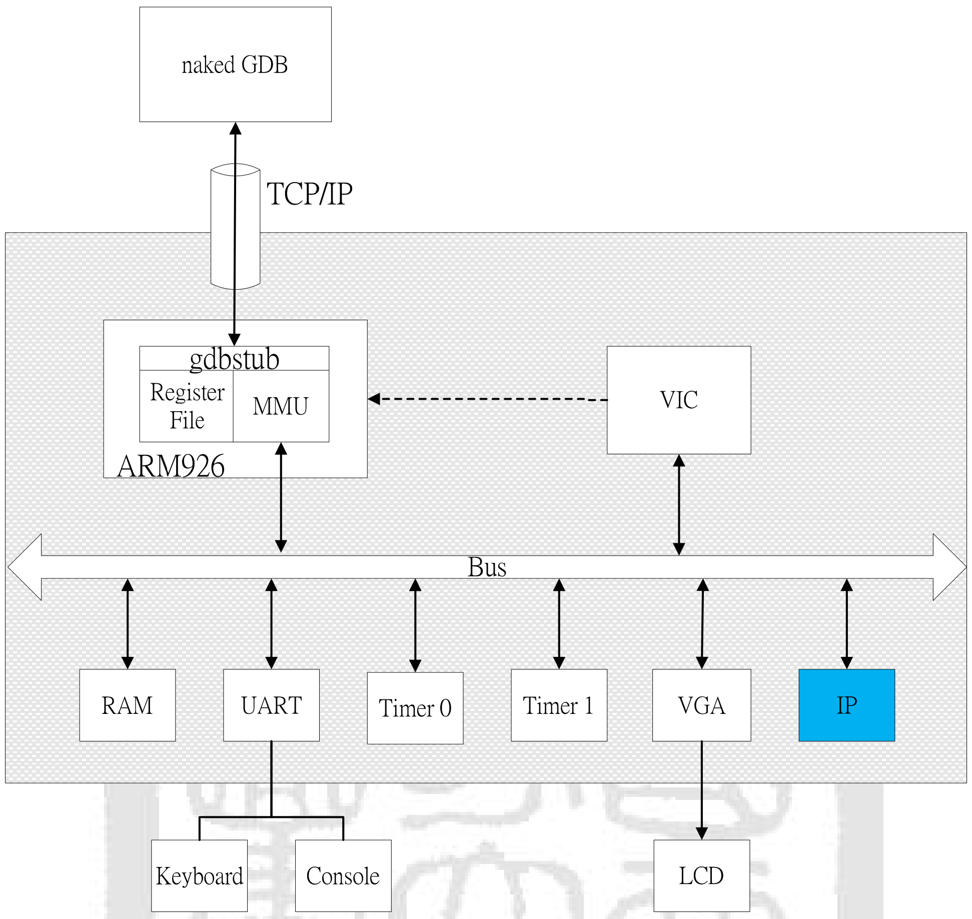


Fig. 3-6: Overview of the SystemC virtual platform

3.5.2 Full System Simulation

The main purpose of hardware and software co-verification is to verify if the software program executes correctly and efficiently on the hardware design. The job of full system simulation co-simulates the hardware and software at the same time to provide system developers a useful co-verification methodology. For a new SoC design, there are two primary benefits of applying full system simulation to co-verification [9]:

(1) First, it allows the system software to be tested and debugged before the

hardware devices are available to use. Therefore, this significantly shortens the development time of the intended design.

(2) Secondly, the system software itself which co-simulates with the hardware designs is an excellent and useful testbench.

For the above two reasons, to co-verify a complex SoC design, performing full system simulation is always an important job for system developers in ESL design methodology.

In this work, we use the Linux OS as an example, which is emulated on our SystemC virtual platform for full system simulation. Fig. 3-7 exhibits the structure of the full system simulation framework. This framework commonly has four echelons:

(1) *hardware devices*

The first echelon is the hardware devices consisting of some customized hardware designs in SystemC, that is, the hardware side.

(2) *OS kernel*

The second echelon is the place where the Linux OS kernel resides. This echelon involves the interrupt service routine (ISR), task scheduler, inter-process communication (IPC) module, and device drivers. This part is also the core component in our full system simulation platform.

(3) *system libraries*

The third echelon contains all of the dynamic linking shared libraries, system application programming interfaces (API), and some specific middleware including the C/C++ standard library.

(4) *user mode apps*

The highest echelon is the user mode programs. All apps including the console terminal and shell program reside in here.

Through this full system simulation framework, all components in these four echelons involving both hardware devices and software programs can be simulated at the same time. Also, all of the system software and application programs can run upon the simulation platform without any modification, so that the programmers are able to design, test, and debug before the corresponding RTL code designs are synthesized.

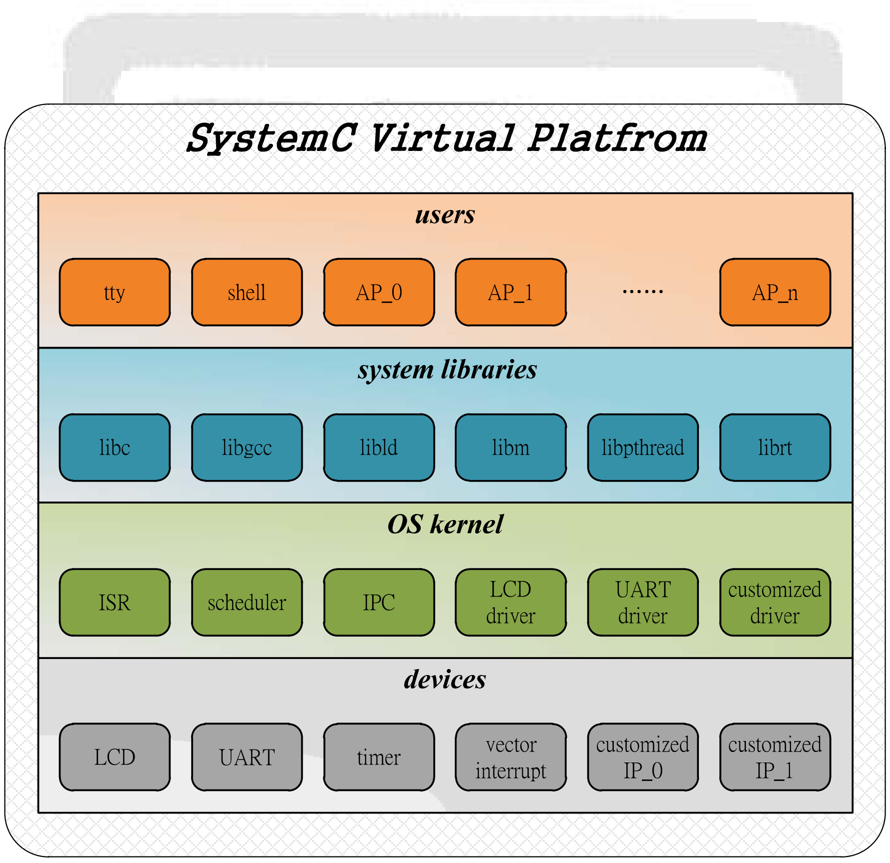


Fig. 3-7: Architecture of the full system simulation virtual platform

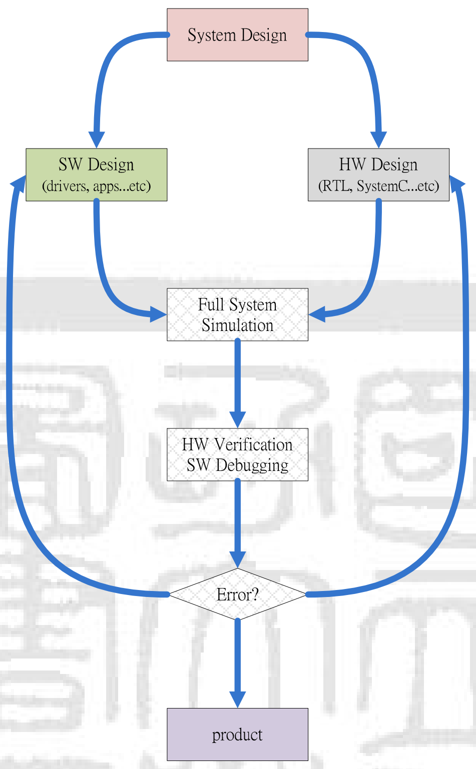


Fig. 3-8: System design flow

Fig. 3-8 is the system design flow of an SoC project. In the design flow, the first step is to sketch the system algorithm, and then implement the hardware and software design. Then, the SystemC virtual platform and the ISS can be applied to involve in the task of full system simulation to co-verify and validate the system algorithm, hardware devices, as well as software programs.

In traditional SoC design flow, the hardware engineers and software programmers are very difficult to start developing at the same time. Following this ESL system design methodology, developers can easily leap across the big hurdle. Because both the hardware and software can be developed and verified in parallel, the total time-to-market of a new SoC will be shortened.

3.5.3 Evaluation Methodology

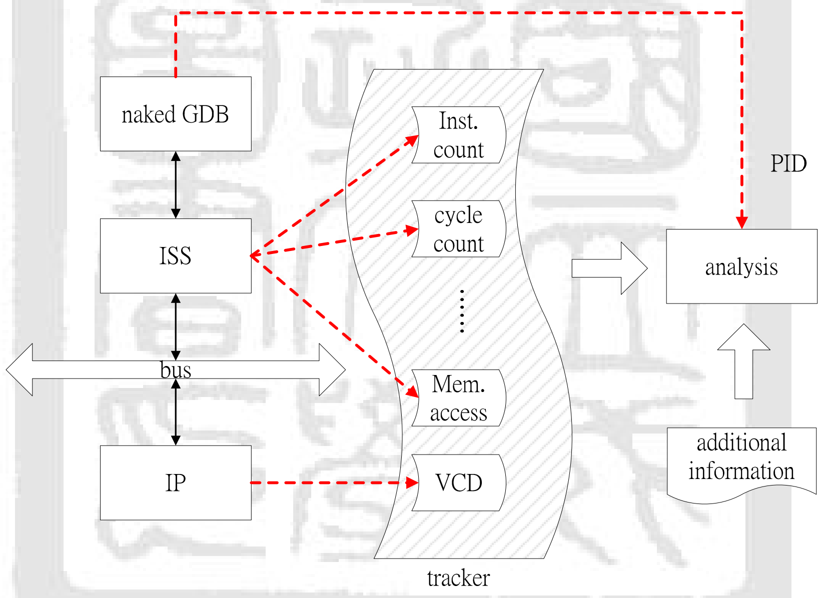


Fig. 3-9: System profiler scheme

The value change dump (VCD) file format is an IEEE standard to record the status of data change and signal waveform for hardware description languages (HDL) in simulation time. Current SystemC standard had already provided an *sc_trace()* API to collect and dump the information for hardware engineers. So that, for our full system simulation platform, we can simply add the *sc_trace()* function into the SystemC hardware model to capture and evaluate the signal waveform if necessary.

For software side, to get the profile we want, we log the information through the PID number of the process. As TABLE— 3-4 shows, the PID number will be registered into the co-processor in run time and our naked GDB scheme has the support to fetch this value.

Fig. 3-9 is the scenario of our profiler that is used to evaluate and log information we need. There are three kinds of traced information we can retrieve after system simulation, that is, the VCD file generated by hardware modules, PID number from the naked GDB, and the program profiles from the ISS. Besides, we might also have some additional information of the target program dumped by the cross toolchain in compiled time, e.g., the system map. These four records can be used to analyze the system algorithm, verify the hardware designs, and check the correctness of the software programs.

Chapter 4 - Platform Verification

4.1 Verification Methodology

After implementing the ARM-based ISS and SystemC full system simulation virtual platform, in this chapter, we are going to test and verify the behavior of the whole virtual platform and see whether it is correct or not.

To verify the virtual platform, running a bare-level program on it to check is necessary. A bare-level program has the ability to directly operate and access the system with

- user mode instructions
- privileged mode instructions
- exception handlers
- the MMU/co-processor of the CPU
- I/O peripherals

that the user mode application programs cannot freely do. In short, a bare-level program can go over the entire system smoothly without any privilege violations. Therefore, general application programs are not suitable for verifying the virtual platform because they cannot cover all the aspects of these system operations.

Since Linux is a very popular OS in embedded systems, the Linux kernel is a good selection to be the testbench for our full system simulation framework.

4.2 Linux Booting Sequence

Fig. 4-1 is the Linux booting sequence [12]. Generally speaking, the booting sequence of an embedded Linux can be separated into the following five steps:

(1) loads the kernel image into the main memory by the bootloader

(2) decompresses the kernel images and setups the MMU

(3) creates the *PID 0* process, setups scheduler, and setups the exception handlers

(4) the *PID 0* process forks the *PID 1* process, and then the *PID 1* process goes to setup all device drivers and the initial root file system

(5) starts to load and run user mode apps

In conclusion, during the boot up time, Linux will visit and access the MMU system, I/O peripheral devices, and exception handlers by both user mode and privileged mode instructions. Consequently, all of the five parts we want to test and verify will be visited during Linux booting.

In accordance with [21], it had proposed a methodology to verify an ARM-based embedded system by Linux kernel and proved that we can guarantee the action of all privileged mode instructions, exception handler, and MMU are working correct if an ARM Linux kernel can be booted up and performed on the virtual platform and the ISS successfully.

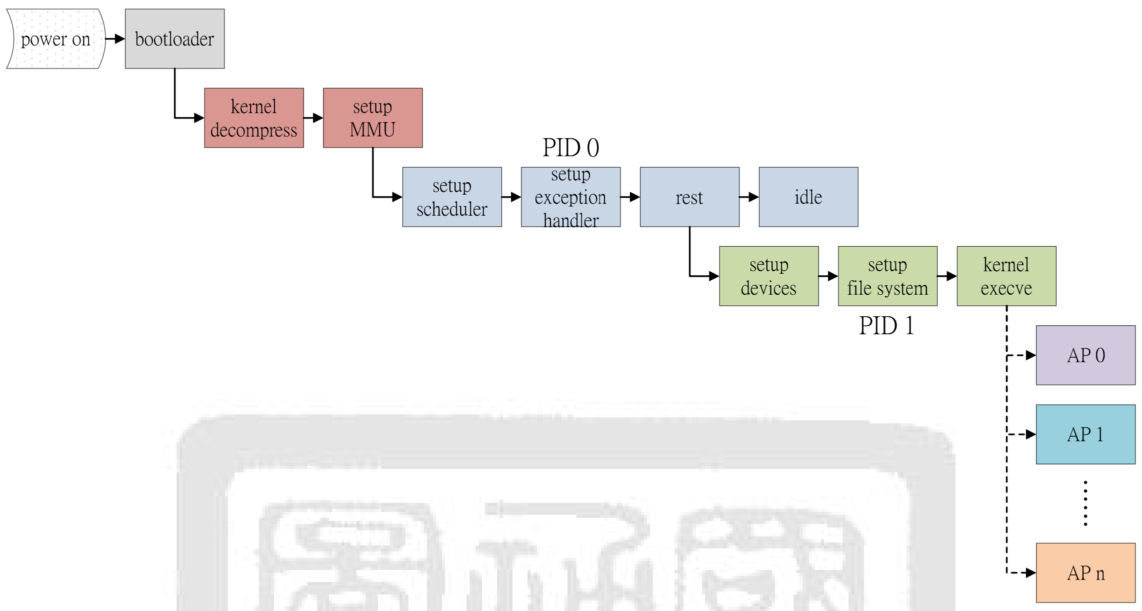
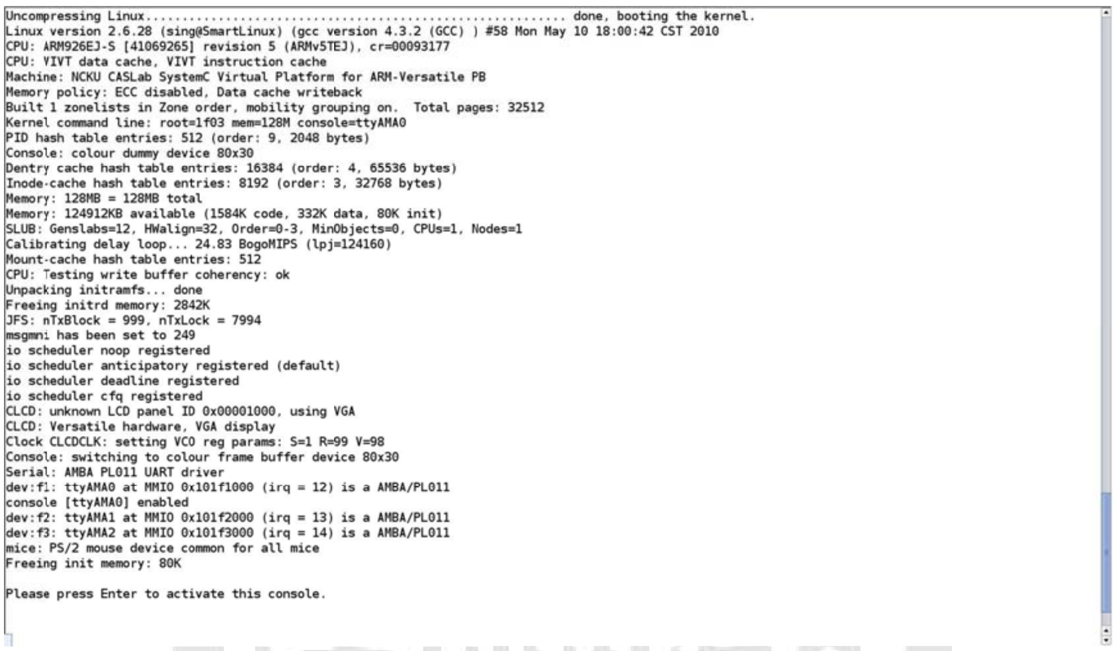


Fig. 4-1: Linux boot up sequence

4.3 Verification Result

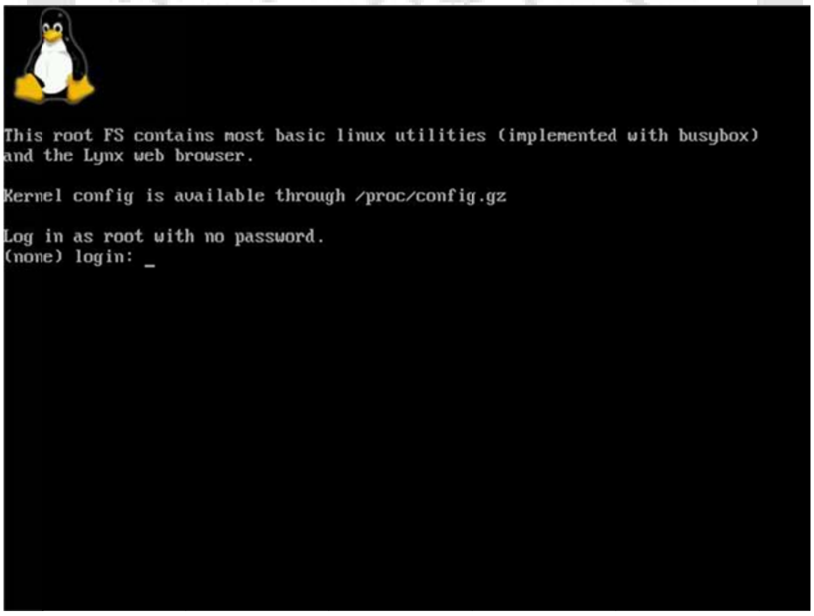
4.3.1 Verification by Linux Booting

Fig. 4-2 and Fig. 4-3 are the snapshots of the console terminal and the LCD panel respectively at Linux booting on our virtual platform. Fig. 4-2 shows the Linux kernel goes to get the system information of our CASLab SystemC virtual platform, and then to initialize the memory system, file system, and I/O peripheral devices. In Fig. 4-3, it presents the Linux kernel drawing the color penguin logo on the LCD panel and shows up the information of the initial root file system.



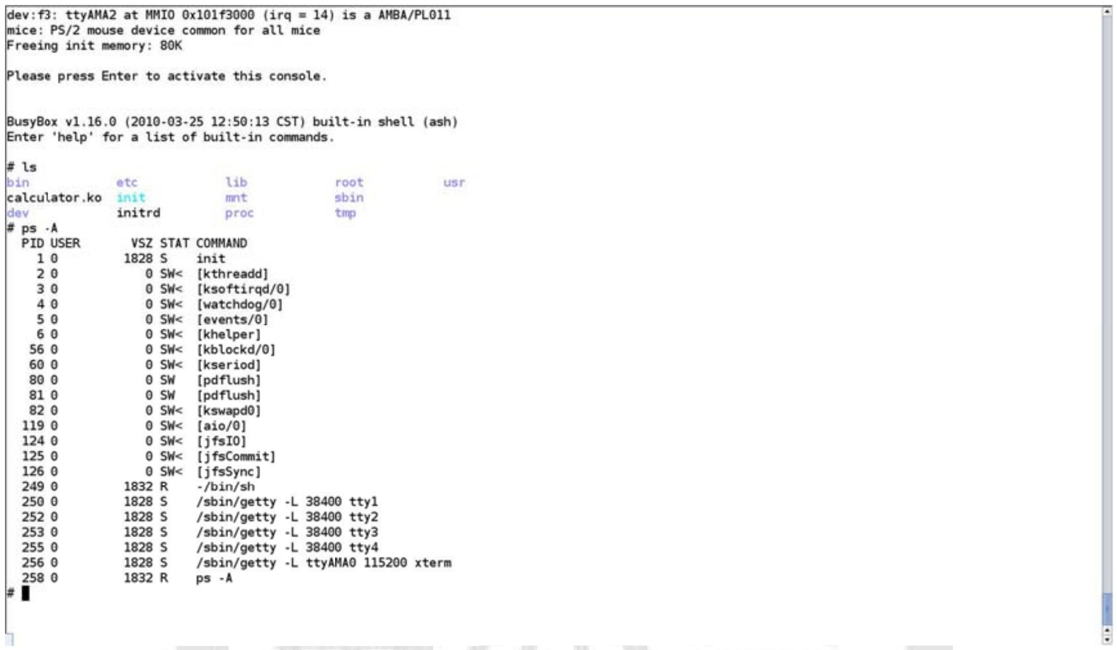
```
Uncompressing Linux.......................................................... done, booting the kernel.
Linux version 2.6.28 (sing@SmartLinux) (gcc version 4.3.2 (GCC) ) #58 Mon May 10 18:00:42 CST 2010
CPU: ARM926EJ-S [41069265] revision 5 (ARMv5TEJ), cr=00093177
CPU: VIVT data cache, VIVT instruction cache
Machine: NCKU CASLab SystemC Virtual Platform for ARM-Versatile PB
Memory policy: ECC disabled, Data cache writeback
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 32512
Kernel command line: root=1f03 mem=128M console=ttyAMA0
PID hash table entries: 512 (order: 9, 2048 bytes)
Console: colour dummy device 80x30
Dentry cache hash table entries: 16384 (order: 4, 65536 bytes)
Inode-cache hash table entries: 8192 (order: 3, 32768 bytes)
Memory: 128MB = 128MB total
Memory: 124912KB available (1584K code, 332K data, 80K init)
SLUB: Genslabs=12, HWalign=32, Order=0-3, MinObjects=0, CPUs=1, Nodes=1
Calibrating delay loop... 24.83 BogoMIPS (lpj=124160)
Mount-cache hash table entries: 512
CPU: Testing write buffer coherency: ok
Unpacking initramfs... done
Freeing initrd memory: 2842K
JFS: nTxBlock = 999, nTxLock = 7994
msgmni has been set to 249
io scheduler noop registered
io scheduler anticipatory registered (default)
io scheduler deadline registered
io scheduler cfq registered
CLCD: unknown LCD panel ID 0x00001000, using VGA
CLCD: Versatile hardware, VGA display
Clock CLCDCLK: setting VCO reg params: S=1 R=99 V=98
Console: switching to colour frame buffer device 80x30
Serial: AMBA PL011 UART driver
dev:f1: ttyAMA0 at MMIO 0x101f1000 (irq = 12) is a AMBA/PL011
console [ttyAMA0] enabled
dev:f2: ttyAMA1 at MMIO 0x101f2000 (irq = 13) is a AMBA/PL011
dev:f3: ttyAMA2 at MMIO 0x101f3000 (irq = 14) is a AMBA/PL011
mice: PS/2 mouse device common for all mice
Freeing init memory: 80K

Please press Enter to activate this console.
```

Fig. 4-2: Snapshot of Linux booting



```
This root FS contains most basic linux utilities (implemented with busybox)
and the Lynx web browser.

Kernel config is available through /proc/config.gz

Log in as root with no password.
(none) login: _
```

Fig. 4-3: Snapshot of the virtual LCD panel



```
dev:f3: ttyAMA2 at MMIO 0x101f3000 (irq = 14) is a AMBA/PL011
mice: PS/2 mouse device common for all mice
Freeing init memory: 80K

Please press Enter to activate this console.


BusyBox v1.16.0 (2010-03-25 12:50:13 CST) built-in shell (ash)
Enter 'help' for a list of built-in commands.

# ls
bin            etc            lib            root           usr
calculator.ko  init           mnt            sbin
dev            initrd         proc           tmp
# ps -A
  PID USER       VSZ STAT COMMAND
    1 0         1828 S    init
    2 0            0 SW<  [kthreadd]
    3 0            0 SW<  [ksoftirqd/0]
    4 0            0 SW<  [watchdog/0]
    5 0            0 SW<  [events/0]
    6 0            0 SW<  [khelper]
   56 0            0 SW<  [kblockd/0]
   60 0            0 SW<  [kseriod]
   80 0            0 SW   [pdflush]
   81 0            0 SW   [pdflush]
   82 0            0 SW<  [kswapd0]
  119 0            0 SW<  [aio/0]
  124 0            0 SW<  [jfsIO]
  125 0            0 SW<  [jfsCommit]
  126 0            0 SW<  [jfsSync]
  249 0         1832 R    -/bin/sh
  250 0         1828 S    /sbin/getty -L 38400 tty1
  252 0         1828 S    /sbin/getty -L 38400 tty2
  253 0         1828 S    /sbin/getty -L 38400 tty3
  255 0         1828 S    /sbin/getty -L 38400 tty4
  256 0         1828 S    /sbin/getty -L ttyAMA0 115200 xterm
  258 0         1832 R    ps -A
# █
```

Fig. 4-4: Snapshot of the PS command

Fig. 4-4 shows the processes information after entering the Linux *ps* command. We can discover that there are already more than 20 kernel mode and user mode processes running at the same time upon Linux kernel. Namely, the multitasking system including the scheduler and the timer device are active successfully.

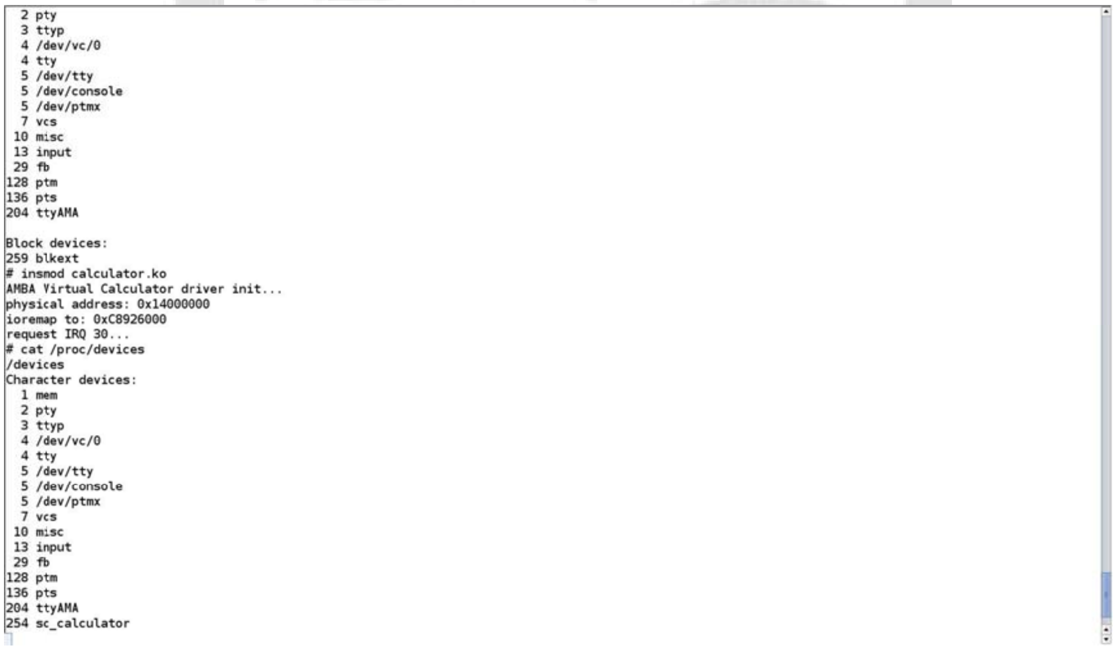
As we have said in Section 4.2, if the Linux kernel can boot up on the virtual platform as normally, we can almost believe it to be correctly implemented.

4.3.2 Verification by Device Driver under Linux

We are going to try to attach a customized hardware device—a specific calculator module onto the virtual machine and hang on the corresponding AMBA (advanced microcontroller bus architecture) device driver [16] into the Linux kernel.

Fig. 4-5 shows using Linux *insmod* command to load the device driver of a calculator module into Linux kernel. Comparing with before and after entering the Linux *insmod* command, we can see there is an extra kernel module called *sc_calculator* recorded in the file of */proc/device* after the command finished executing in this figure.

This work demonstrates that the virtual platform is able to co-work with customized hardware devices. Moreover, the programmers can examine and verify the device driver on this virtual platform by co-simulating with the hardware module.



```
  2 pty
  3 ttyp
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 29 fb
128 ptm
136 pts
204 ttyAMA

Block devices:
259 blkext
# insmod calculator.ko
AMBA Virtual Calculator driver init...
physical address: 0x14000000
ioremap to: 0xC8926000
request IRQ 30...
# cat /proc/devices
/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 29 fb
128 ptm
136 pts
204 ttyAMA
254 sc_calculator
```

Fig. 4-5: Snapshot of loading a device driver

4.3.3 Verification by User Mode Applications under Linux

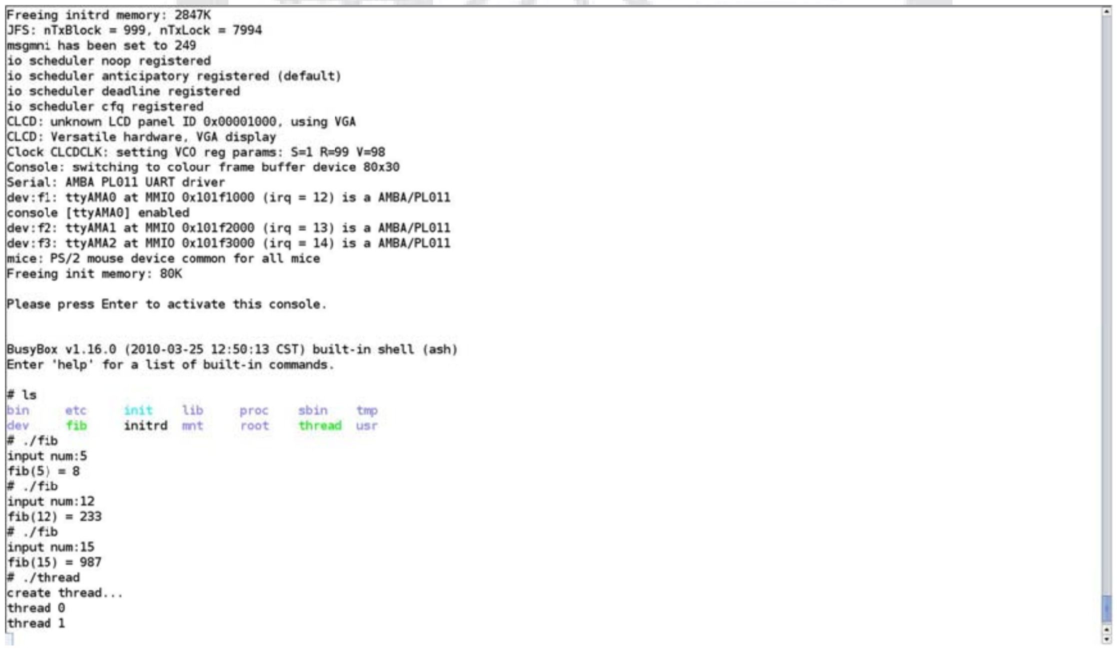
TABLE— 4-1: List of application programs we used

Category	**Program**	**Operations**
ours	loop	branches
	Hanoi	branches/recursion
	thread	multi-threading/multi-tasking
	bSort	branches/recursion/integer arithmetic
	hSort	branches recursion integer arithmetic
	factorial	
	gcd	
MiBench	SHA	
	dijkstra	
	qSort	
	string	
	susan	integer arithmetic floating arithmetic
	jpeg	
	math	

Undoubtedly, the full system simulation virtual platform can also be applied to emulate the user mode application programs the same way as the device drivers. Here we take several ARM Linux apps [32][34] to run on the virtual platform with Linux OS kernel.

TABLE— 4-1 lists the apps we used to verify the ISS and the virtual platform. There are two classes of our testbenches; one is our own basic program design and the other comes from MiBench [19] which is a famous benchmark suite in computer system exploration. This demonstration is used to confirm that the shared libraries and the user mode application programs run well on our full system simulation framework. Through these testbenches, we can check and verify operations of the ISS design as well. In TABLE— 4-1, it also shows the operations in coverage for these testbenches.

Fig. 4-6 presents an example that some ARM Linux apps run on the virtual platform. It shows an application program of calculating Fibonacci series and a multi-threading program within *pthread* library.



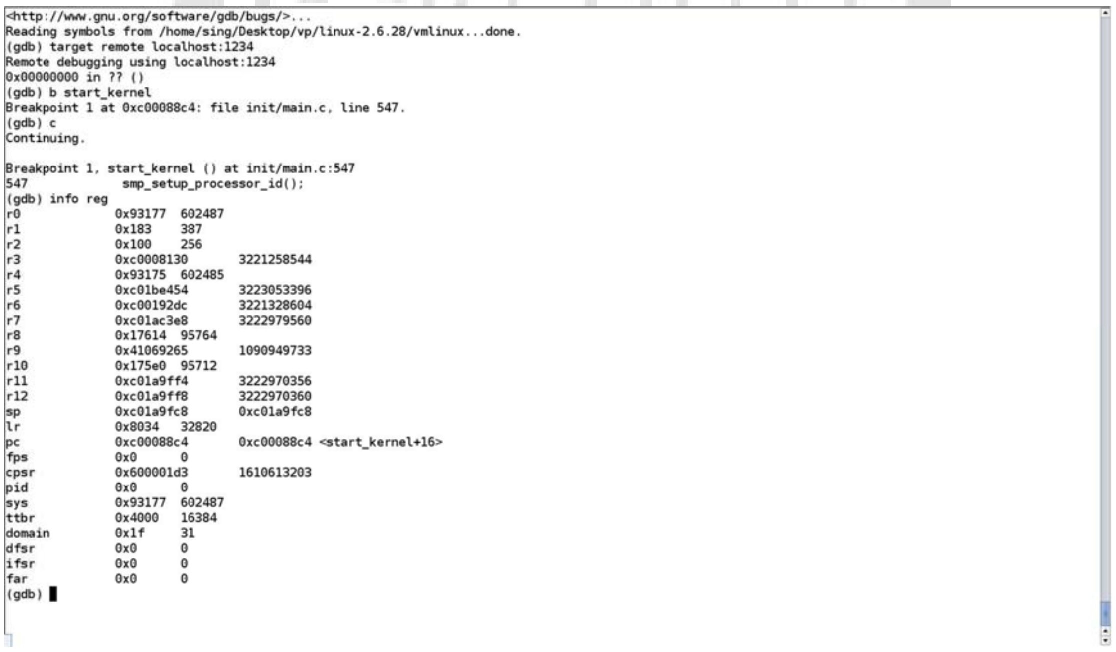
```
Freeing initrd memory: 2847K
JFS: nTxBlock = 999, nTxLock = 7994
msgmni has been set to 249
io scheduler noop registered
io scheduler anticipatory registered (default)
io scheduler deadline registered
io scheduler cfq registered
CLCD: unknown LCD panel ID 0x00001000, using VGA
CLCD: Versatile hardware, VGA display
Clock CLCDCLK: setting VCO reg params: S=1 R=99 V=98
Console: switching to colour frame buffer device 80x30
Serial: AMBA PL011 UART driver
dev:f1: ttyAMA0 at MMIO 0x101f1000 (irq = 12) is a AMBA/PL011
console [ttyAMA0] enabled
dev:f2: ttyAMA1 at MMIO 0x101f2000 (irq = 13) is a AMBA/PL011
dev:f3: ttyAMA2 at MMIO 0x101f3000 (irq = 14) is a AMBA/PL011
mice: PS/2 mouse device common for all mice
Freeing init memory: 80K

Please press Enter to activate this console.


BusyBox v1.16.0 (2010-03-25 12:50:13 CST) built-in shell (ash)
Enter 'help' for a list of built-in commands.

# ls
bin      etc      init     lib      proc     sbin     tmp
dev      fib      initrd   mnt      root     thread   usr
# ./fib
input num:5
fib(5) = 8
# ./fib
input num:12
fib(12) = 233
# ./fib
input num:15
fib(15) = 987
# ./thread
create thread...
thread 0
thread 1
```

Fig. 4-6: Snapshot of executing apps

4.3.4 Co-Work with the Naked GDB

Fig. 4-7 is the snapshot of the naked GDB. In this figure, we notice that the naked GDB goes to probe and fetch the general registers and the specific co-processor registers of ARM processor.

The naked GDB absolutely helps and allows the programmers to observe and monitor not only the general registers but the MMU/co-processor system in the program debugging stage. Through this naked scheme, the software developers can debug the bare-level programs and recognize these system programs like a common application program without any additional hardware circuit support.



```
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/sing/Desktop/vp/linux-2.6.28/vmlinux...done.
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x00000000 in ?? ()
(gdb) b start_kernel
Breakpoint 1 at 0xc00088c4: file init/main.c, line 547.
(gdb) c
Continuing.

Breakpoint 1, start_kernel () at init/main.c:547
547             smp_setup_processor_id();
(gdb) info reg
r0             0x93177  602487
r1             0x183    387
r2             0x100    256
r3             0xc0008130       3221258544
r4             0x93175  602485
r5             0xc01be454       3223053396
r6             0xc00192dc       3221328604
r7             0xc01ac3e8       3222979560
r8             0x17614  95764
r9             0x41069265       1090949733
r10            0x175e0  95712
r11            0xc01a9ff4       3222970356
r12            0xc01a9ff8       3222970360
sp             0xc01a9fc8       0xc01a9fc8
lr             0x8034   32820
pc             0xc00088c4       0xc00088c4 <start_kernel+16>
fps            0x0      0
cpsr           0x600001d3       1610613203
pid            0x0      0
sys            0x93177  602487
ttbr           0x4000   16384
domain         0x1f     31
dfsr           0x0      0
ifsr           0x0      0
far            0x0      0
(gdb) █
```

Fig. 4-7: Snapshot of the naked GDB

4.3.5 Summary of System Verification

In summary, as we have emphasized in Section 4.2, we believe and have confidence that the functions of our ISS and the SystemC virtual platform are implemented accurately and correctly since it can successfully boot up the ARM Linux kernel and execute user mode applications. That is, the system developers can trust and rely on the simulation result from this full system simulation virtual platform and its ARM-based ISS.

Chapter 5 - Evaluation and Results

5.1 Experimental Environment and Parameters

TABLE— 5-1 lists the experimental environment of our simulation framework. The experimental environment can be separated into two parts, the host machine side and the target machine side.

The host machine is an Intel x86 based computer with Linux OS where the virtual platform runs on. All programs on the host machine are compiled by gcc v4.2.4 and the SystemC library in used is v2.2.0.

The target machine is the SystemC full system simulation virtual platform which emulates and executes an ARM Linux kernel here. On the target virtual platform, the ARM Linux kernel is compiled by arm-elf-gcc v4.3.2 and all apps including the busybox tool set are compiled by arm-linux-gcc v4.4.3. Yet, the initial RAM disk and all dynamic liking libraries of Linux are made by busybox v1.16.0.

TABLE— 5-1: Experimental environment

Host machine	Target machine
● Intel Core 2 Q9500	● arm-elf-gcc v4.3.2
● 2GB DDRII SDRAM	● arm-linux-gcc v4.4.3
● kUbuntu v8.04 with 32-bits kernel v2.6.24	● Linux kernel v2.6.28 for AEM Versatile-PB
● gcc v4.2.4	● busybox v1.16.0
● arm-elf-gdb v7.1	
● SystemC v2.2.0	
● SDL v1.2	
● HP CACTI v6.5	

TABLE— 5-2: Parameters of the virtual platform

Name	**Settings**
CPU model	ARM926 (ARMv5)
D-cache	32kB, 4-ways, 256-sets, round-robin
I-cache	32kB, 4-ways, 256-sets, round-robin
Bus model	perfect (no latency)
DRAM model	perfect (no latency)
DRAM size	128MB, 2 banks
clock rate of CPU	200MHz
clock rate of peripherals	4MHz

TABLE— 5-3: Average executing cycle(s) of CPU emulator

	cycles
arithmetic & logic instructions	1
32-bits multiply instructions	2
64-bits multiply instructions	4
load/store instructions	N + 1 (N = number of words to transfer)
coprocessor access	2
branch penalty	3
swi instruction	3
D-cache miss penalty	16
I-cache miss penalty	8

TABLE— 5-2 lists the parameters of the ISS and the SystemC virtual platform in assumption. Both of the I-cache and D-cache in the ISS are size of 32 KB, 4 way set associative with 256 sets, and the replacement policy is round robin. In addition, here we assume the DRAM and bus model are perfect, i.e., no access latency, though other models can also be explored . Also, the clock rate of the CPU (ISS) is assumed to be 200 MHz as well as all of the I/O peripherals are running at 4 MHz.

TABLE— 5-3 is the timing setting of the ISS referring to [3][5][32]. The average executing time of all arithmetic and logic instructions (except the multiply instructions) are one clock cycle. The time to perform load/store instructions is N + 1 clock cycles where N is the number of words (4 bytes width here) to be transferred. The total time to access the co-processors is assumed as two clock cycles. The branch and exception penalty are three clock cycles. Finally, the penalty of D-cache miss and I-cache miss are set to be 16 and 8 clock cycles respectively.

5.2 Simulation Performance

5.2.1 The Throughput

In this section, we are going to evaluate the throughput of our ARM-based interpretive ISS and the SystemC virtual platform. TABLE— 5-4 shows the throughput of a few of ARM-based ISSs and their comparisons.

In Section 3.1.2, we have shown the simulation upper bound of SystemC kernel v2.2.0 is about 4.4 million clock cycles per second on our Intel Q9800 experimental machine. For our SystemC virtual platform, the simulation throughput is around 2.1 million instructions per second (MIPS) with 2.9 million clock cycles per second.

Comparing between the value 2.9 and 4.4, the simulation speed of our SystemC virtual platform has a 65.9% drop.

As TABLE— 5-4 shows, not unexpectedly, the QEMU and SimIt-ARM obtain much higher throughput because both of them are designed within dynamic binary translation technique which cannot explore the timing information in system design space. Meanwhile, the GDB ARMulator is a functional accurate ISS model and the FaCSim is implemented in pure C/C++ language that is hard to attach new hardware devices. Though they have gained much higher performance than ours, all of these ISS frameworks are not suitable for ESL design methodology. Finally, the throughput of Simplescalar and the synthesizable RTL ISS module are much poorer than our virtual platform; both cannot run an OS kernel in acceptable time duration.

TABLE— 5-4: Throughput of different ARM ISSs

Model	Scheme	MIPS	Features
real hardware	Versatile-PB	77.1	
dynamic binary translation	QEMU	> 100	
	SimIt-ARM	30.0	with instruction metric
functional accurate in pure C/C++	GDB ARMulator	8.2	
	FaCSim	4.3	optimized for MP
approximate-timed/cycle accurate	SystemC VP	2.1	approximate-timed accurate in SystemC
	Simplescalar	0.9	cycle accurate
	RTL	< 0.1	pin accurate, synthesizable

5.2.2 SystemC Speedup

TABLE— 5-5: Performance of different SystemC schemes

	SC_CTHREAD	***SC_THREAD***	***SC_METHOD***
Million cycles / sec	2.2	2.5	2.9
Speedup(*SC_CTHREAD*)	100.00%	113.64%	131.81%

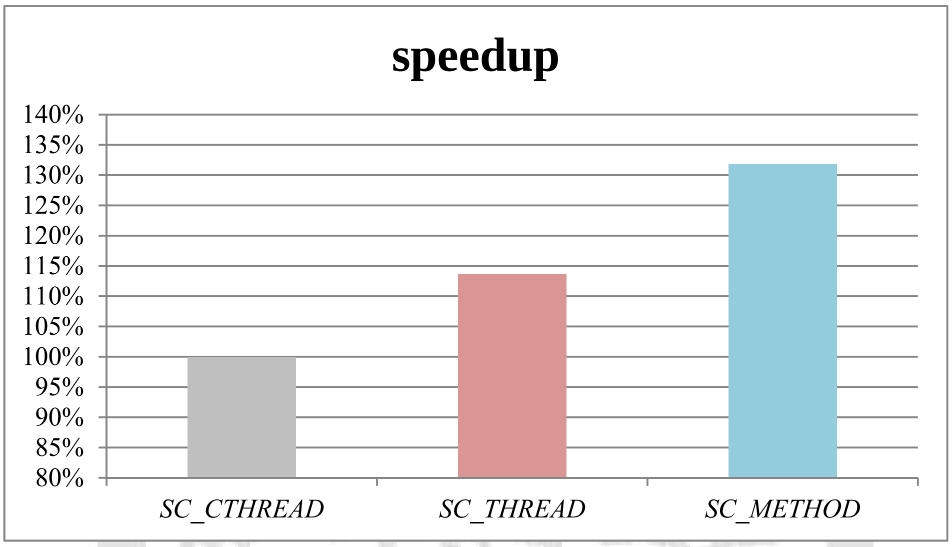


Fig. 5-1: Speedup of different SystemC scheme

We have implemented the SystemC virtual machine and the ISS in all of the three different SystemC simulation processes. Here we are going to measure and compare the performance of these SystemC simulation methodologies.

TABLE— 5-5 shows the simulation performance of our SystemC virtual platform in each implementation methodology of SystemC process. From the table, the simulation speed of our virtual platform is near to 2.9 million clock cycles per second in *SC_METHOD*. Instead, if it is implemented in *SC_CTHREAD*, the simulation speed is only 2.2 million clock cycles per second. Also, the virtual platform in *SC_THREAD* will run at 2.5 million cycles per second.

Fig. 5-1 is the bar chart of the speedup comparing with *SC_CTHREAD*. This figure illustrates that using *SC_METHOD* will have a speedup of more than 130% and the speedup of using *SC_THREAD* is about 114%. Using *SC_METHOD* to build the ISS and the SystemC virtual platform has obtained the best simulation performance as we have expected.

5.3 Cycles per Instruction

TABLE— 5-6: CPI of the ISS

	jpeg	susan	SHA	dijkstra	qSort	string	math	total
Cycles	18519389	28130600	13964233	76008446	96078007	25559581	536300593	794560849
Inst.	13614684	22361502	11426232	54258552	53904386	15721040	355114001	526400397
CPI	1.36	1.26	1.22	1.40	1.78	1.63	1.51	1.51

$$\text{average CPI} = \frac{\text{total cycle count}}{\text{total instruction count}} \qquad \text{Equ. 5-1}$$

From ARM Spec. [3] [5], the average clock per instruction (CPI) is 1.5 within MiBench [19] for ARMv5 architecture. Here we use MiBench and apply Equ. 5-1 to evaluate the average CPI of our ISS.

TABLE— 5-6 shows the results of the CPI evaluation for each test program in MiBench on our SystemC ISS. The CPI of our ISS is in the range from 1.22 to 1.78 for all test programs. Overall, the CPI of our ARM-based ISS is around 1.51 which tightly couples with the value of 1.5. Through this experiment, we believe that the timing setting in assumption is reasonable and trustworthy for our ARM-based ISS. Note that, these timing parameters are freely adjustable for SystemC modules.

5.4 Power Metric

TABLE— 5-7: Power model of ARM Versatile-PB

Model / Power	I-cache & D-cache (32kB, 4 ways associative)	DRAM (128MB, 2 banks)
leakage power (mW)	1.1601	34.9812
refreshing power (mW)	0	1.0107
read operation (nJ)	0.3394	3.6097
write operation (nJ)	0.1637	3.6241

To do the task of memory power estimating, at first, we use and look for HP CACTI [27] program (be cited over 2000 times) to figure out the power/energy model of I-cache, D-cache, and the DRAM module of ARM Versatile-PB platform. TABLE— 5-7 lists the leakage power, refreshing power, and switching energy of the ARM Versatile-PB memory system based on CACTI. Again, we ignore the refreshing power of I-cache and D-cache because they are SRAMs. Note that, to evaluate the power model, we assume both the cache and DRAM module are manufactured by the 90nm processing technology.

By recording the total executing time and number of memory access including read operations, write operations, and cache misses, we can easily figure out the total energy dissipation of a program by Equ. 3-1. TABLE— 5-8 is the results of power evaluation for MiBench on our SystemC virtual platform.

TABLE— 5-8: Power estimation of memory system

Program / Energy(mJ)			jpeg	susan	SHA	dijkstra	qSort	string	math
I-cache	read	No.	13614684	22361502	11426232	54258552	53904386	15721040	355114001
		energy	4621.232	7590.165	3878.406	18416.98	18296.766	5336.193	120536.345
	miss	No.	98821	88372	77454	89886	151992	83803	5937845
		energy	268339.334	239966.036	210319.211	244077.163	412720.293	227559.337	16123671.82
D-cache	read	No.	4506274	7973216	3301398	16290791	20523885	5806617	7262554
		energy	1529429.396	2706109.51	1120494.481	5529094.465	6965806.569	1970765.81	2464910.828
	write	No.	2518749	1517741	1996887	5556542	14664336	3871841	50798426
		energy	412243.649	248408.669	326830.495	909439.229	2400111.873	633704.216	8314178.383
	miss	No.	42647	27993	20685	156033	284938	21496	149274
		energy	171645.646	112666.226	83252.988	628001.618	1146818.462	86517.101	600797.995
DRAM	read	No.	1131744	930920	785112	1967352	3495440	842392	48696952
		energy	4085256.317	3360341.924	2834018.786	7101550.514	12617489.77	3040782.402	175781387.6
	write	No.	341176	223944	165480	1248264	2279504	171968	1194192
		energy	1236455.942	811595.45	599716.068	4523833.562	8261150.446	623229.229	4327871.227
leakage + refreshing		cycles	18519389	28130600	13964233	76008446	96078007	25559581	536300593
		energy	2.752	4.181	2.075	11.296	14.278	3.798	79.7
total			7707994.268	7486682.161	5178512.51	18954424.83	31822408.46	6587898.086	207733433.9

5.5 Profiling of Linux Booting Sequence

In this section, we try to profile and analyze the instruction count and cycle count of the Linux kernel booting procedure by our SystemC virtual platform. Note that, the evaluation results are deeply relying on the configuration of Linux kernel and the initial root file system. Here the kernel and the root file system in use are lightweight versions; the root file system is originally 5.8MB and 2.8MB after gzip compression.

To complete the ARM Linux booting procedure, the total instruction count is about 360 million and the total cycle count is about 480 million for our experimental environment. Referring to the ARM manual [3], for a real ARM926 CPU implemented in .18μm process technology, the operating frequency is about 200 to 250MHz. That is, the total boot up time is less than 3 seconds. Again, the kernel and root file system we used are very light, so it is not surprised with this rapid boot up performance.

Fig. 5-2 and Fig. 5-3 are the pie charts of total instruction count and cycle count of ARM Linux booting sequence respectively. From these two charts, to boot up Linux kernel, we discover that most of time is spent on mounting the initial root file system (including decompressing binary image of the file system) and kernel decompression. To setup the initial root file system, it costs more than 65% of the entire boot time; the process of kernel decompression occupies more than 20% of the boot time. The task of file system decompression is a quite large effort for a Linux based SoC design. In reality, the size of root file system in gzip format is usually much greater than tens or even hundreds of MB, so that the decompression process may spend more than 70% of the boot time.

For some specific embedded devices, the manufacture has customized the file system of Linux to slash the time of setting up the file system. In this case, the setup time of the initial root file system can be omitted. Fig. 5-4 and Fig. 5-5 are the total instruction count and cycle count of ARM Linux booting without the root file system. These two charts show the *start_kernel()* function (which is the PID 0 process of Linux) occupies about 20% of the boot time in this special case. Furthermore, the LCD (which is the function to draw and show up the Linux penguin logo on the color LCD panel) spends 6% of the entire boot time.

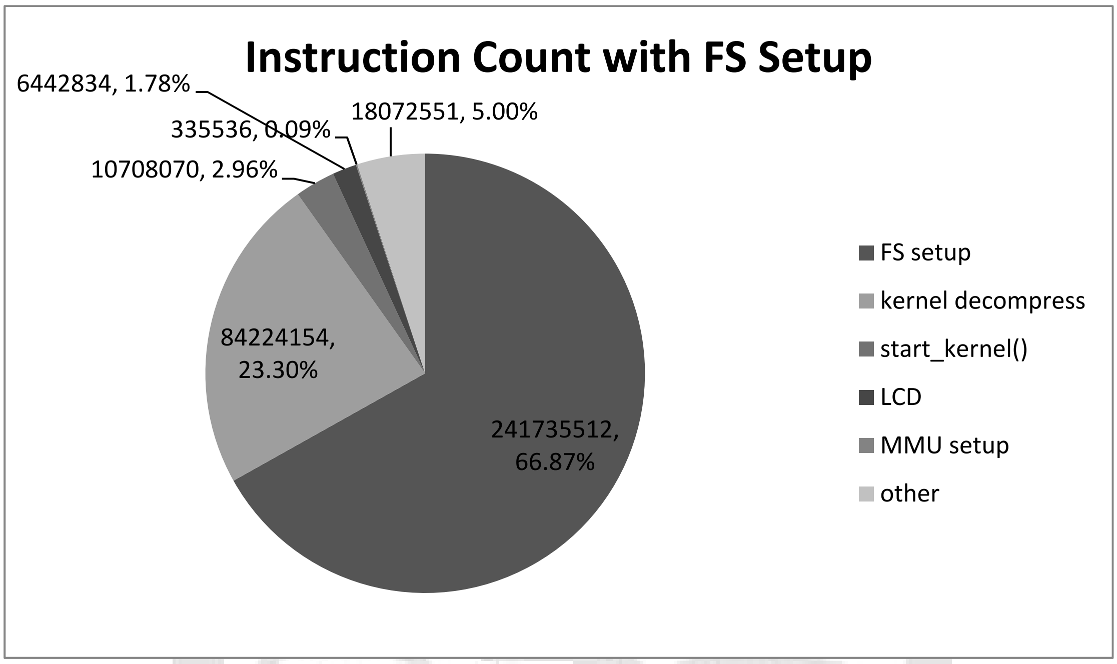


Fig. 5-2: Instruction count of Linux booting with FS setup

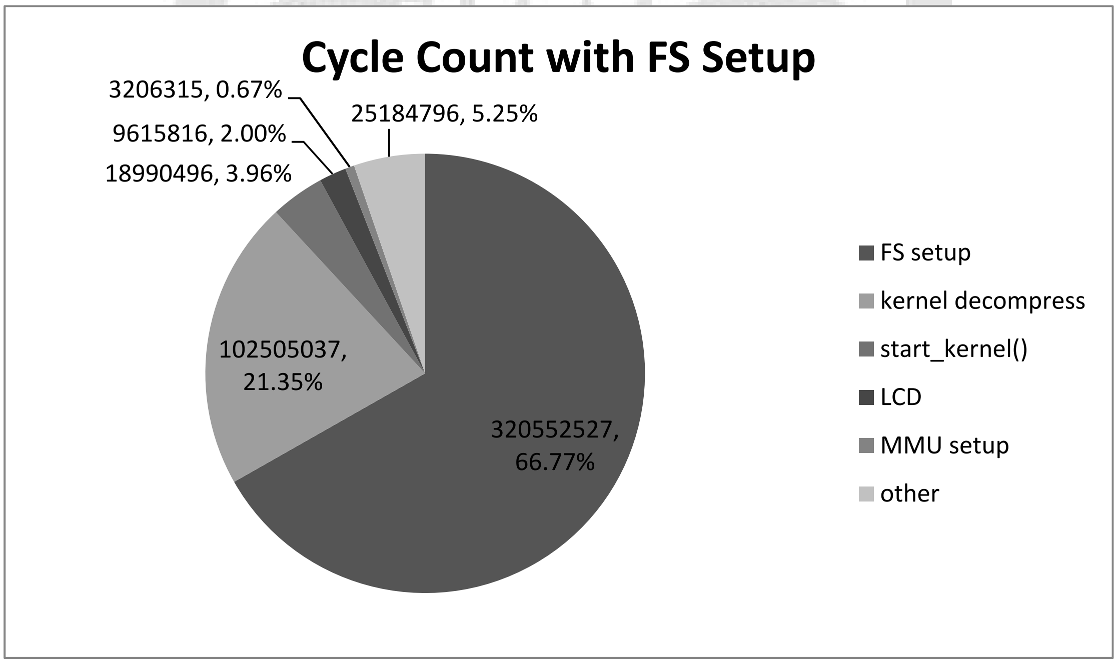


Fig. 5-3: Cycle count of Linux booting with FS setup

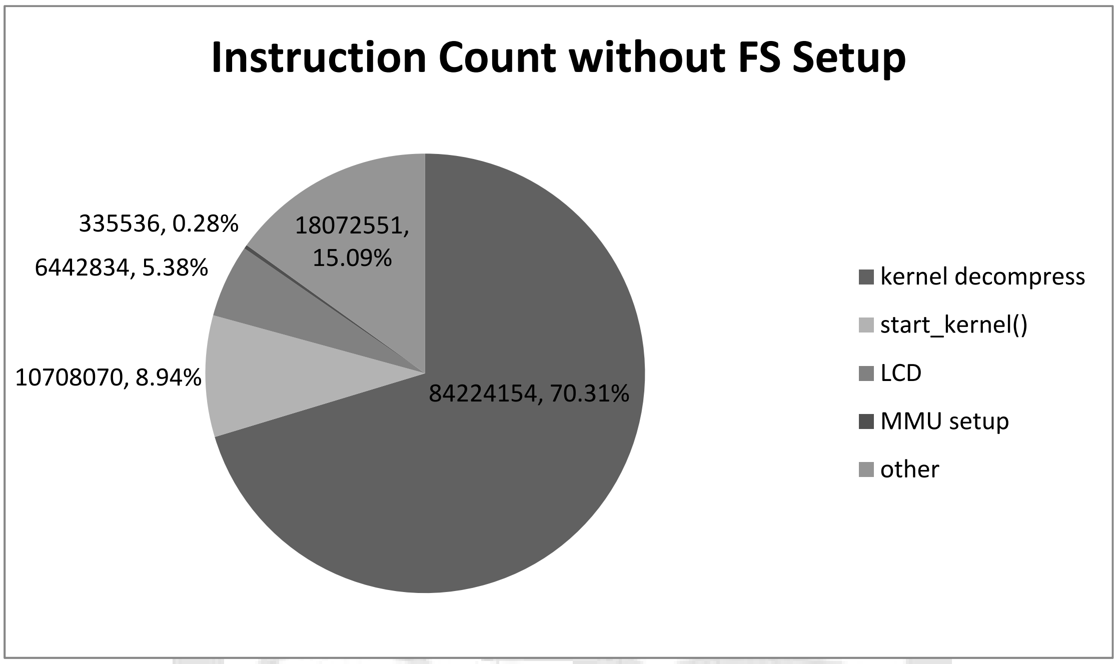


Fig. 5-4: Instruction count of Linux booting without FS setup

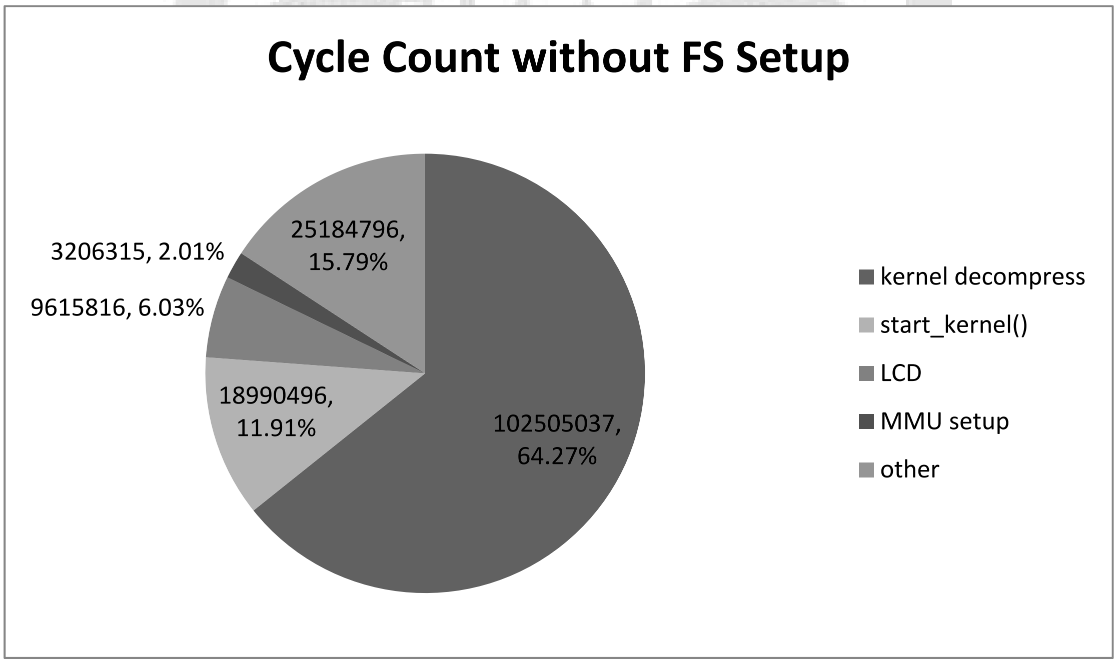


Fig. 5-5: Cycle count of Linux booting without FS setup

Chapter 6 - Conclusions

We have designed a functional and/or approximate-timed accurate virtual platform and its corresponding ARM-based interpretive ISS in SystemC models for ESL design methodology.

This virtual platform provides a full system simulation environment that can directly execute system programs including OS kernel, device drivers, and the corresponding application programs without any or only with tiny modification. The virtual platform also has built in a customized gdbserver called naked GDB which is suitable for debugging both bare-level and application programs, and thus software engineers are able to develop the system software in the early development stage without any real hardware devices support. In addition, system developers are apt at analyzing the system algorithm and reviewing the interaction between software and hardware by utilizing this full system simulation framework.

Finally, using this SystemC virtual platform and the ISS, the complexity of developing a new SoC design can be reduced and the time-to-market will be shortened altogether.

Chapter 7 - Future Works

To make the ISS and the SystemC virtual platform more powerful, in the future, we would like to keep on maintaining and improving it:

- to improve the SystemC simulation kernel with *pthread* library, so that the simulation performance can be accelerated by symmetric multiprocessing (SMP) host machine in parallel [24].
- using a local cache scheme or another methodology to improve the instruction decoder of the ISS to raise up the throughput.
- to fix the CPU emulator to be a multi-core processor and provide a proficient method to develop multi-core programs.
- to extend the naked GDB to support multi-core program debugging and verifications.

References

[1] "PrimeCell Color LCD (PL110) Technical Reference Manual DDI-0161E," ARM Co. Ltd., May 2003.

[2] "ARM Dual-Timer Module (SP804) Technical Reference Manual," ARM Co. Ltd., January 2004.

[3] "ARM926EJ-S Technical Reference Manual DDI-0198D," ARM Co. Ltd. January 2004.

[4] "PrimeCell Vectored Interrupt Controller (PL190) Technical Reference Manual DDI-0181E" ARM Co. Ltd., November 2004.

[5] "ARM Architecture Reference Manual DDI-0100I," ARM Co. Ltd., July 2005.

[6] "PrimeCell UART (PL011) Technical Reference Manual DDI-0183F," ARM Co. Ltd., November 2005.

[7] "Versatile Application Baseboard for ARM926EJ-S User Guide DUI-0225B," ARM Co. Ltd., July 2006.

[8] "IEEE Standard SystemC Language Reference Manual," Design Automation Standards Committee, IEEE Computer Society, March 2006.

[9] J. R. Andrews, "Co-Verification of Hardware and Software for ARM SoC Design," Elsevier Inc., August 2004.

[10] D. Beal, "The Magic of Virtualized Systems Development," Virtutech Co. Ltd., October 2009.

[11] D. C. Black, J. Donovan, B. Bunton, and A. Keist, "SystemC: From the Ground up 2nd Edition," Springer Media Inc., 2010.

[12] D. P. Bovet and M. Cesati, "Understanding the Linux Kernel 3rd Edition," O'Reilly Media Inc., November 2005.

[13] D. Burger and T. M. Austin, "The Simplescalar Tool Set Version 2.0," University of Wisconsin-Madison Computer Sciences Department Technical Report, June 1997.

[14] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," Proceedings of the 2005 USENIX Annual Technical Conference, Anaheim, CA, USA, April 2005.

[15] L. Charest, C. Pilking, and P. Paulin, "SystemC Performance Evaluation Using a Pipelined DLX Multiprocessor," Proceedings of the 2002 ACM/IEEE Design, Automation, & Test in Europe Conference (DATE'02), Paris, France, March 2002.

[16] J. Corbet, A. Rubini, and G. Kroah-Hartman, "Linux Device Driver 3rd Edition", O'Reilly Media Inc., January 2005.

[17] J. Gilmore and S. Shebs, "GDB Internals—A Guide to the Internals of the GNU Debugger," Cygnus Solutions, February 2004.

[18] T. Grötker, S. Liao, G. Martin, and S. Swan, "System Design with SystemC," Springer Media Inc., 2002.

[19] M. R. Guthaus, et al., "MiBench: a Free, Commercially Representative Embedded Benchmark Suite," Proceedings of the 2008 IEEE International Workshop on Workload Characterization (WWC'01), Austin, TX, USA, December 2001

[20] A. H. Han, Y.-S. Hwang, Y.-H. An, S.-J. Lee, and K.-S. Chung, "Virtual ARM Platform for Embedded System Developers," Proceedings of the 2008 IEEE International Conference on Audio, Languages, and Image Processing (ICALIP'08), pp. 586-592, Shanghai, China, November 2008.

[21] H.-W. Kao, "Embedded Processor Verification Using Particular Characteristics of Linux Operating System," 2006 master thesis of National Cheng Kung

University, Tainan, Taiwan, July 2006.

[22] J. Lee, et al., "FaCSim: A Fast and Cycle-Accurate Architecture Simulator for Embedded Systems," Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08), pp. 89-100, Tucson, AZ, USA, June 2008.

[23] P. S. Magnusson, et al., "Simics: A Full System Simulation Platform," IEEE Computer, Vol. 35, No. 2, pp. 50-58, February 2002.

[24] P. Ezudheen, et al., "Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines," Proceedings of the 23rd ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS'09), pp. 80-87, Lake Placid, NY, USA, June 2009.

[25] J. Montanaro, et al., "A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor," IEEE Journal of Solid-State Circuits, Vol. 31, No. 11, pp. 1703-1714, November 1996.

[26] M. Montón, A. Portero, M. Moreno, B. Martínez, and J. Carrabina, "Mixed SW/SystemC SoC Emulation Framework," Proceedings of the 2007 IEEE Symposium on Industrial Electronics (ISIE'07), pp. 2338-2341, Vigo, Spain, June 2007.

[27] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," HP Laboratories, April 2009.

[28] M. Reshadi, P. Mishara, and N. Dutt, "Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation," Proceedings of the 40th ACM/IEEE Design Automation Conference (DAC'03), Vol. 8, No. 3, pp. 758-763, Anaheim, CA, USA, June 2003.

[29] M. Reshadi and N. Dutt, "Reducing Compilation Time Overhead in Compiled

Simulators," Proceedings of the 21st IEEE Internaional Conference on Computer Design (ICCD'03), pp. 151-153, San Jose, CA, USA, October 2003.

[30] M. Reshadi, P. Mishara, and N. Dutt, "Hybrid-Compiled Simulation: An Efficient Technique for Instruction-Set Architecture Simulation," ACM Transactions on Embedded Computer Systems, Vol. 8, No. 3, pp. 20-27, April 2009.

[31] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary Translation," Communications of the ACM, Vol. 36, No. 2, pp. 68-81, February 1993.

[32] A. N. Sloss, D. Symes, and C. Wright, "ARM System Developer's Guide: Design and Optimizing System Software", Elsevier Inc., March 2004.

[33] R. Stallman, R. Pesch, S. Shebs, et al., "Debugging with GDB—The GNU Source-Level Debugger 9th Edition", Cygnus Solutions, February 2004.

[34] R. Stones and N. Matthew, "Beginning Linux Programming 3rd Edition," Wiley Publishing Inc., January 2004.