# A Non-Copyable Disk (NCdisk)

## Concept, Architecture, Security Protocol, and Business Analysis

**Michael S. Wang**
**Advisor: Professor Ruby B. Lee**
**May 13, 2008**

# Honor Code

This paper represents my own work in accordance with

University regulations

Michael S. Wang

Submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Engineering
Department of Electrical Engineering
Princeton University

# Acknowledgements

First of all, I would like to thank Professor Ruby Lee for all of the guidance, opportunity, and help that she has given me throughout the years. Looking back at my Princeton career, I have to say that the most valuable experience that I got out of it was from the research that I have done with her. She has always challenged me and given me the opportunity to get involved with some of her most exciting work. Although the work was quite challenging at times, she was always there to provide guidance.

I would also like to thank Jeff Dwoskin, Reouven Elbaz, David Champagne, Yedidya Hilewitz, Yu-Yuan Chen, James Donald, Eric Keller, and Neil Vachharajani. These graduate students have not only mentored me throughout the years, but have now become some of my closest friends. Through them, I have learned what it takes to be a great mentor and I hope to behave the same way as I go off to graduate school.

Further, I would like to thank all of my friends. During the past four years, I have certainly lived through the entire spectrum of emotions from joy and excitement to anxiety and frustration. They were the stabilizing force in my life. They helped me to stay afloat during rough times and also cheered along with me when I was happy.

Last but certainly not least, I like to thank the two most important people in my life. I like to thank my dad, Jerry, who is my most important role model. When people ask me who I look up to the most or who I would most like to become, my answer has always been “my dad”. I would also like to thank my mom, Susan, who has sacrificed so much in order to provide me with the opportunities that I have today.

# Abstract

Piracy of copyrighted digital contents, such as movies and music is rampant in cyberspace. A piece of digital material may be repeatedly copied and proliferated throughout the Internet with ease. We examined both software and hardware vulnerabilities in existing digital copy-protection methods. As a result, we propose a non-copyable disk (NCdisk) that makes it significantly harder for digital contents to be copied. Any digital content written onto the NCdisk can only be read through a predefined set of outputs of the NCdisk, and the original plaintext digital form may never be read out of the NCdisk. We add a minimal set of components based on the Secret-Protection (SP) architecture to the existing disk's SoC chipset to attribute the disk with the non-copyable property. We enhance the original SP architecture with a new instruction and a defined set of trusted software APIs for the NCdisk application. We further present the security protocol to be used along with the NCdisk to provide a copy-protected digital movie download scenario. Finally, we analyze the prospects of marketing the NCdisk for the online movie download application by devising two different business models and examining their competitiveness and financial projections.

In the Appendices, we also describe an alternate, more complex ASIC-based architecture for the NCdisk that we first proposed, and the implementation of the PAX processor that performs fast cryptographic processing and could be used as the embedded processor in an NCdisk.

# Table of Contents

# Chapter 1
# Introduction

### *1.1 Problem and Motivation*

Today, an immense amount of information exists in digital form. A large percentage of it is copyrighted contents that should only be available to authorized users. In such cases, the user is usually permitted to read (or play) the contents but should not be allowed to copy and distribute the contents. Nevertheless, unauthorized copying and distribution of digital contents occur frequently and is a major problem for many content providers.

This content-piracy problem is currently a serious concern for the movie and music industry. Due to the increasing Internet bandwidth and the emergence of more powerful portable player devices, the demand for directly downloading media contents from the Internet to an end-user's player device is on the rise. A typical copy-protection method [1] used to prevent the illegal copying of these media contents is as follows: a content provider installs his own software onto the user's player device, such as a PC, an iPod, etc. Then, the provider sends encrypted contents to the user's device. In order to obtain the keys used to decrypt and read the encrypted contents, the user must authenticate with the content provider or with a third party licensing clearinghouse. Next, the keys are sent to and hidden on the user's device. Only the content provider's installed software on that device can find and use the keys to decrypt the encrypted contents. Hence, this copy-protection method restricts the copying of contents by sending only encrypted contents over public networks, hiding keys on the user's devices, and allowing only the provider's special software to find and use these keys.

A major weakness with the existing copy-protection method described above is that the encrypted contents are sent to various kinds of player devices that do not have secure processing architectures to hide the decryption keys. In the underlying processor architectures, machine instructions, registers, memories and buses are open resources that can be controlled or accessed by the operating system (OS), application software and also by malicious software. Furthermore, since both the application software and the OS can have bugs and software vulnerabilities, hackers can use these software weaknesses to find the hidden decryption keys.

We propose a non-copyable disk (NCdisk), which is a storage device that automatically encrypts all data written into it and does not allow the plaintext form of the data to leave it except through controlled display outputs. We propose a minimal set of changes to an existing disk controller System-on-Chip (SoC) to attribute the disk with the non-copyable property. Our proposal is based on the Secret-Protection (SP) secure processor architecture [2][3][4], which provides a secure environment to store critical secrets and allows only a trusted software module to access these critical secrets. Further, our proposal enhances the original SP architecture with a new instruction to simplify secure embedded storage. We also define a set of trusted software APIs for the NCdisk application.

### *1.2 Threat Model and Assumptions*

We assume that the content provider can write a trusted software module that will be allowed to use and access critical secrets like envryption keys, but does not leak these secrets out. Further, we assume that any other software is un-trusted and should not be allowed to access critical secrets. This includes the Operating System and other applications of the personal computer or handheld computing or entertainment device. The attacker is able to mount software

attacks. He can monitor all network transactions. He can also mount some hardware attacks, such as probing external memories and buses. We assume that physically probing inside a chip, such as a System-on-Chip (SOC) is more difficult without destroying functionality, and hence this is not in our threat model. We also do not consider side-channel attacks on a SoC, or denial of service attacks.

# Chapter 2
# NCdisk Concept

Figure 2-1 shows a flowchart of the NCdisk concept. The NCdisk is a data storage device, in which any digital content written into the device is automatically encrypted using a key that is generated by the NCisk that never leaves the NCdisk. All data stored on the NCdisk are in such an encrypted form, and the stored data can only be read through a set of predefined outputs, such

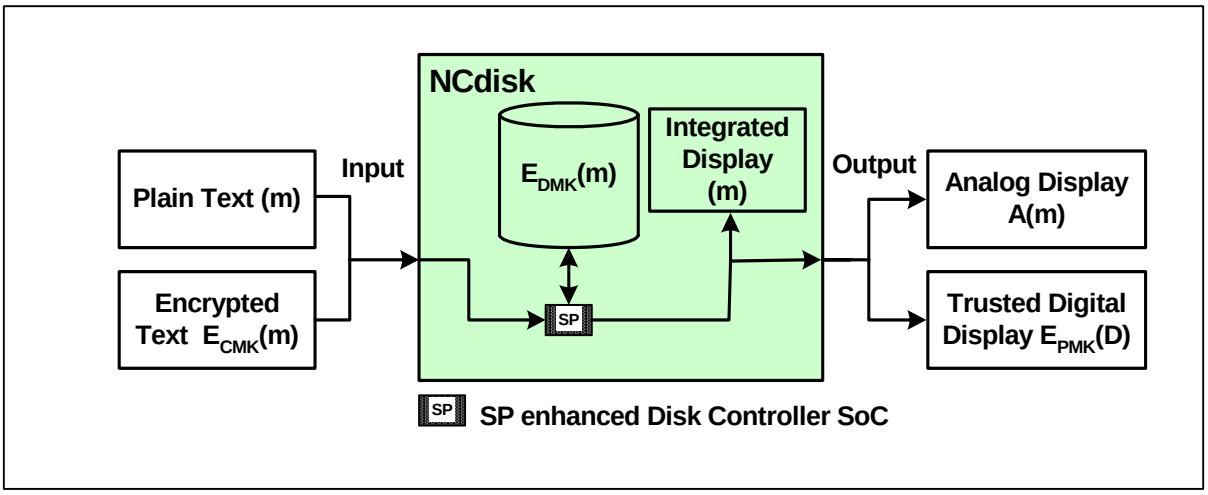



Fig 2-1. NCdisk Concept

that the digital plaintext form of the data never leaves the NCdisk.

Both plaintext data and encrypted data may be written onto the NCdisk. Each encrypted data is encrypted using a secret key, called the Content-Provider Media Key (CMK), which is known only by the content provider and the NCdisk. The CMK is never actually stored anywhere but is

instead generated using a shared key between the content provider and the NCdisk. We examine the detailed key management protocol in Chapter 3. Both the content provider and the NCdisk must have a secure location to generate and use the CMK so that it is not revealed to anyone else. We assume that the content provider has such a secure location, and we show in Chapter 3 how the NCdisk achieves this. If the CMK is kept secret, then the plaintext form of the encrypted data will not be leaked out during the network communications phase, where the encrypted version of the movie is transferred from the content provider across the public networks and input into the NCdisk.

Either plaintext data or CMK-encrypted data can be input into the NCdisk. Plaintext data input is first encrypted using the Device Media Key (DMK). For CMK-encrypted data input, the NCdisk first decrypts the data using the CMK and then re-encrypts the data using the DMK. The DMK is generated within the NCdisk and it never leaves the disk. We discuss in Chapter 3 how to keep the DMK secret from everyone, including the user of the disk. Note also that each input data to the NCdisk is encrypted using a different DMK, as described in detail in Chapter 3. Encrypting all the data stored on the NCdisk using a DMK protects the storage phase of the data, by ensuring that the plaintext version of the digital data never resides on the disk.

Any data stored on the NCdisk can only be read out of the disk through a pre-defined set of output channels. An encrypted digital data can be decrypted and converted to an analog format, which can then be sent out of the NCdisk. Alternatively, an encrypted digital data can be decrypted using the DMK and re-encrypted using a Player Media Key (PMK), which is only known by a trusted digital display and the NCdisk. Both the trusted digital display and the NCdisk must have a secure location to generate and use the PMK so that it is not revealed to anyone else. The PMK-encrypted data is sent out of the NCdisk. Third, if the NCdisk has an

integrated display, such as a built-in LCD screen like in handheld entertainment devices (e.g., the iPod), then the NCdisk may decrypt the stored data and send the digital streaming data to the integrated display. It is assumed that it is hard for a casual attacker to siphon off information on the internal link connecting the NCdisk and its integrated display. Note that this integrated display is not foolproof against more dedicated attackers. Nevertheless, this integrated display raises the bar against possible attacks to siphon off information. In all three pre-defined output channels, the high-fidelity, digital plaintext version of the data never leaves the NCdisk in the output phase of the NCdisk. To summarize, the NCdisk ensures that no one, not even the legitimate user of the NCdisk, can obtain a copy of the digital plaintext version of the data stored on the disk.

The NCdisk addresses some of the weaknesses of existing copy-protection methods. Instead of sending copyrighted movie or music contents to insecure PCs or portable media players, a content provider can instead send these contents to a user's NCdisk. In a way, the NCdisk functions like a book in that only those people who have physical possession of the NCdisk can view the contents stored on it. Just as it would be very inconvenient for a person to copy a bound book, a user would have a very difficult time trying to copy the original digital plaintext data stored on the NCdisk. However, unlike a book, the NCdisk provides the convenience of directly downloading and viewing copyrighted digital contents without the need to physically travel to a store. Also, with the growing storage densities, an NCdisk can store many items of digital multimedia content.

# Chapter 3
# NCdisk Design Based on the SP Architecture

In this chapter, we show how the NCdisk design can be achieved using an adaptation of the Secret-Protection (SP) architecture targeted for general-purpose microprocessors [2,3]. We simplify the SP architecture for use in an embedded system like a commodity disk controller, but we also expand upon its reduced-mode version proposed for sensor-nodes in [4] by providing a more flexible secure scratchpad memory. We define new SP registers and a new SP instruction for this. The work reported in this chapter has been published in our conference paper [5]. This chapter takes material from, and expands upon, material from [5].

### *3.1 NCdisk SoC Architecture*

The NCdisk concept ultimately boils down to achieving two security goals. The first goal is to enable the NCdisk to be able to store secret keys and ensure that these keys never leak out of the NCdisk. The second goal is to fully predefine how data can be read out of the NCdisk such that the original digital plaintext data is never leaked out. We do not achieve these two goals by redesigning a completely new disk architecture from scratch. Instead, we only need to be concerned with the disk controller components (shown shaded in yellow in Figure 3-1), which control how data is written in or read out of a disk. We achieve these two goals by implementing a SoC consisting of existing disk controller components, plus a minimal set of additions. This

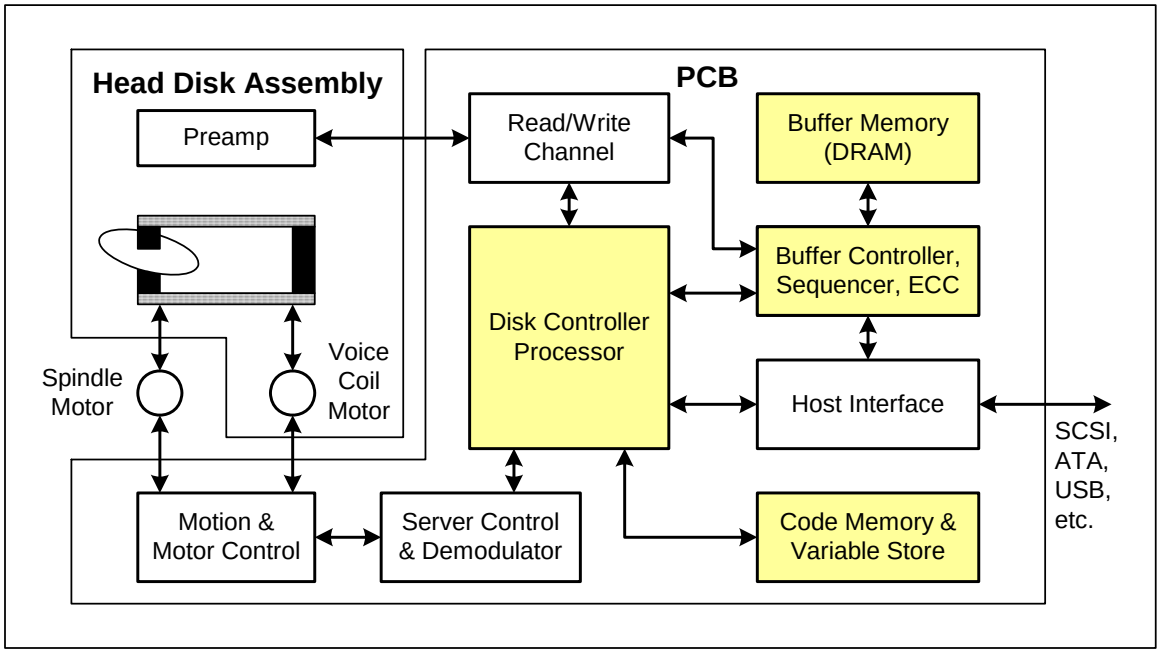



Fig 3-1. Existing Hard Disk Architecture

new SoC can then be connected to the rest of the existing disk components to turn an existing disk [6] into an NCdisk. Figure 3-2 shows the new SOC with enhancements described below.

The existing disk controller components in the SoC include a disk controller processor, a read/write buffer control, and some RAM and ROM memory. The additions are divided into two types. The first type of additions comes from the Secret-Protection (SP) architecture[2][3][4], which provides a secure environment for a set of trusted software modules (TSM) to access critical secrets, while preventing these secrets from leaking out of the SoC. We enhance the original SP architecture with a new instruction. SP enhancements are shown in green in Figure 3.2. The SP additions include new SP registers and hardware support for new SP instructions. Also, portions of the RAM and ROM are dedicated for SP software. As we examine below, the processor used in the SoC will perform a significant amount of cryptographic functions.

Appendix B describes our work on a parallel project involving the PAX processor, which has special features for accelerating cryptographic processing. The PAX processor seems like a good fit for the NCdisk SoC.

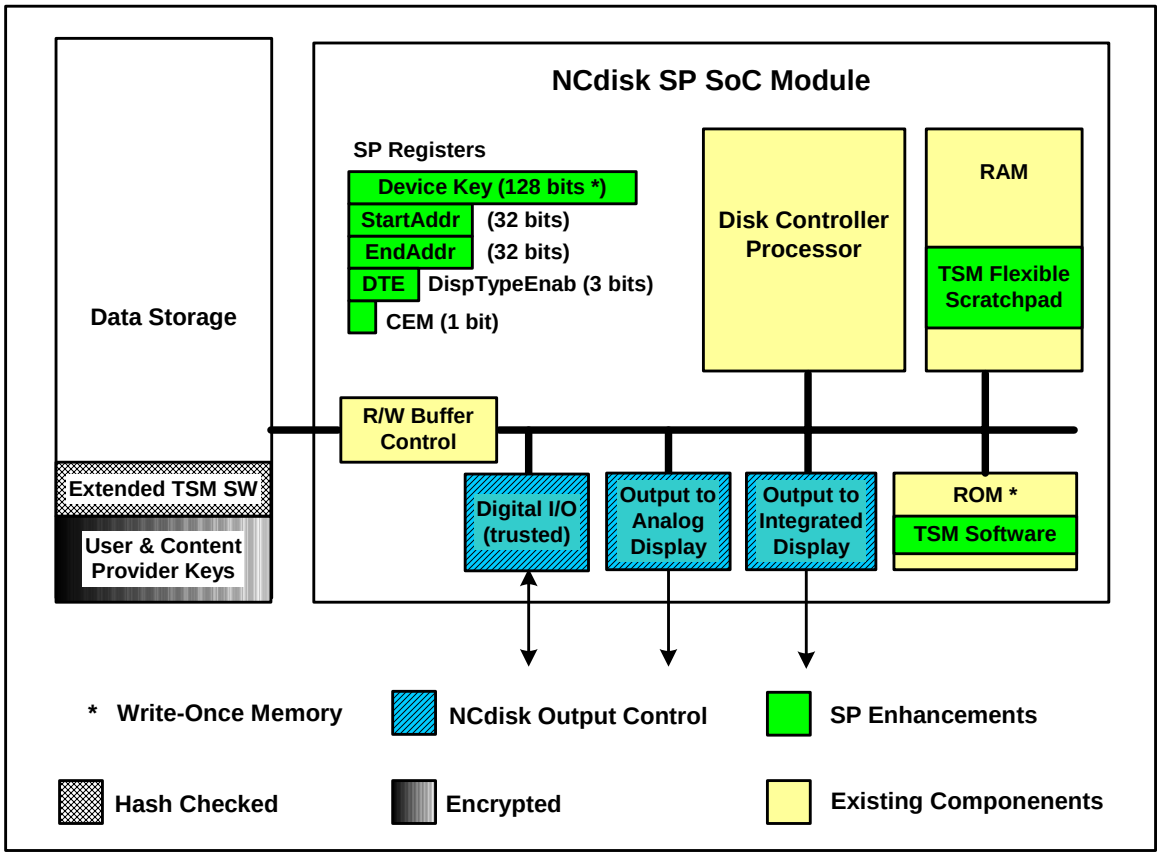



Fig 3-2. SP SoC for NCdisk (not drawn to scale)

The second type of additions is the output interface. There are three different output interfaces, which encompass the predefined set of outputs. These are shown in blue in Figure 3.2.

We also define a set of trusted software APIs for the NCdisk application that restricts the input and output functions with respect to the NCdisk. Other software can only invoke the NCdisk through this set of trusted software APIs.

Next, we examine how this SP-based SoC achieves the two goals of the NCdisk.

### *3.2 Use and Enhancement of SP Architecture for NCdisk*

*A. Storing and Protecting Keys*

The SP-based SoC stores keys in two places. First, the SoC stores a 128-bit key called the Device Key in a non-volatile on-chip register. The Device Key is not pre-set by the NCdisk manufacturer. Instead, each NCdisk generates its own Device Key upon initialization for deployment. This can be done by the owner of the NCdisk. The protocol for initialization will be described in a later section. This Device Key never leaves the NCdisk. (Note that this Device Key is called the Device Master Key, DMK, in the authority-mode SP architecture described in [3]. This is not to be confused with our DMK which is a different Device ***Media*** Key generated from the Device Key for each movie stored in the NCdisk.)

Second, user and content provider keys can all be stored off-chip, encrypted with the Device Key. Since the protection of the off-chip keys hinge on the protection of the Device Key, we focus on efforts to protect this on-chip Device Key.

The Trusted Software Module (TSM) of the SP architecture plays an important role in protecting the Device Key. No software, except the TSM, can access the Device Key register. Only the TSM software stored in the on-chip ROM can get a key that is derived from the device key. Table 3-1 contains SP instructions used for protecting the TSM and its execution. Note that the SecureMem_Set instruction is a newly defined instruction that I have added to the SP architecture, which provides more flexible secure scratchpad memory for TSM execution. This is described further below.

| SP Instruction | Description |
|---|---|
| Begin_TSM (on-chip ROM ) | Begins execution of the TSM (Enables access of TSM scratchpad memory) |
| End_TSM (TSM only) | Ends execution of the TSM (Disables access of TSM scratchpad memory) |
| SecureMem_Set (TSM only) | Sets the StartAddr and EndAddr registers to define the TSM scratchpad memory |
| DK_Derive_Key (TSM only) | Derive a new encryption key using the device key and an input key id |

Table 3-1. SP Instructions used for NCdisk

The SP instruction Begin_TSM turns on the Concealed Execution Mode (CEM) status bit register, while the End_TSM instruction turns off the CEM status bit register. The Begin_TSM instruction can only be invoked by programs stored on the on-chip TSM ROM. Also, the other SP instructions, such as End_TSM, SecureMem_Set, and DK_Derive_Key may only be invoked when the CEM status bit is turned on. This implies that the TSM software stored in ROM can execute the SP instructions only after the Begin_TSM instruction is invoked first. Further, this TSM ROM code can call more complicated TSM code that is stored off-chip. This extended TSM code must be integrity-checked before it is run. Note that since all TSM software must start off with the Begin_TSM instructions, which can only be invoked from the ROM, no other external software can call the DK_Derive_Key instruction to derive an encryption key using the device key. The DK_Derive_Key is the only instruction that can use the Device Key. Even this instruction cannot read out the value of the Device Key to another register or memory. Instead, it can only use the Device Key to derive different encryption keys (Device Media Keys, DMKs) for encrypting different files stored on the NCdisk.

To ensure that no non-TSM software may run during CEM, we disable interrupts during CEM mode, similar to [4]. This also allows us to simplify the SP architecture by eliminating the SP registers used to save the hash of the encrypted general registers, and the interrupt return address register needed to protect the saved TSM state upon interrupts [2,3].

Simply disallowing non-TSM software to access the Device Key register is not enough to prevent its contents from leaking out of the SoC. The run-time data generated by the TSM software in ROM must not be leaked out of the SoC because this data may include information that can reveal the device key. Similar to the sensor-mode SP architecture [4], we propose to dedicate a portion of RAM as TSM scratchpad memory. This scratchpad memory can only be accessed when the CEM status bit is on.

***Enhancement of SP architecture:***

***We propose a new SP instruction called SecureMem_Set, which can set the start and end addresses of the scratchpad memory. The start address and end address are stored in the new 32-bit StartAddr and EndAddr registers (see Figure 3.2). The SecureMem_Set instruction can change the values of these registers. When the SoC is in the CEM mode, the memory location between the StartAddr and EndAddr becomes accessible to the TSM software, which is the only software that can run during the CEM mode. However, when the SoC is not in the CEM mode, this scratchpad memory will not be accessible by any instructions. This enables a flexible-sized TSM scratchpad memory, which gives the TSM software programmer the ability to decide how to allocate the RAM between TSM trusted access and general access (untrusted) areas.***

Further, not allowing any software to directly access the Device Key register and preventing the run-time data of the TSM software from leaking out of the SoC still does not ensure that the Device Key will not leak out of the SoC. The TSM software must be carefully written to ensure that this trusted software does not send any information that can be used to detect bits of this key, outside of the SoC. Towards this goal, we define a fixed set of API functions (see Table 3-2) to be implemented by the TSM software for the NCdisk. Any other software or external control can only use the NCdisk by calling one of these predefined functions. None of these API functions will output the Device Key, or plaintext digital data, from the SoC.

| API Function | Description |
|---|---|
| TSM_Write | Write data into NCdisk |
| TSM_Read_Analog | Output to analog channel |
| TSM_Read_Trusted | Output to trusted display |
| TSM_Read_Integrated | Output to integrated display |

Table 3-2. TSM API

*B. Controlled Predefined Output*

The second goal of the NCdisk is to predefine how data can be read out of the NCdisk such that the original digital plaintext data is never leaked out. This part of the problem is not defined by the previous SP architecture papers in [2,3,4]. (In [2] and [3], trusted input/output mechanisms are assumed to exist in the trust model, but not explicitly defined.)

We achieve this goal of controlling the output of the NCdisk by carefully defining its TSM API functions. There are three API functions for reading data out of the NCdisk. Each API function reads data out through a different output interface on the SoC. These three API functions provide the only way for software or external control to read data out of the NCdisk and none of these API functions will leak out the original digital plaintext data.

The TSM_Read_Analog function decrypts the stored data, converts it into an analog format through the D/A converter, and sends it out of the NCdisk. This API function performs the analog conversion immediately after the decryption, and since interrupts are disabled, the plaintext *digital* data (e.g., the high fidelity movie) will not leak out.

The TSM_Read_Trusted function decrypts the stored data with the DMK, re-encrypts it with the PMK, and sends encrypted digital data out of the NCDisk. Each data has its own DMK, and each trusted display may have its own PMK. The DMKs, CMKs, and PMKs are never stored anywhere. Instead, they are deleted right after encryption and re-derived upon decryption by a shared secret key or the Device Key by the trusted software.

Finally, the TSM_Read_Integrated function decrypts the stored data and sends it to an integrated display through the integrated display interface. Since the display is integrated with the NCdisk, we assume that it is much harder for an attacker to siphon off the data on the internal link connecting the integrated display to the NCdisk.

While we have defined these three controlled output interfaces, it is left to future work to further strengthen their secure implementations. For example, the TSM_Read Integrated function should be implemented by some integrated mechanism that defeats siphoning of plaintext data between the NCdisk and the integrated display in a handheld multimedia device (e.g., an iPod device).

### *3.3 Security Protocol*

Before presenting the NCdisk security protocol, we first examine an online movie download scenario for using the NCdisk, as shown in Figure 3-3.

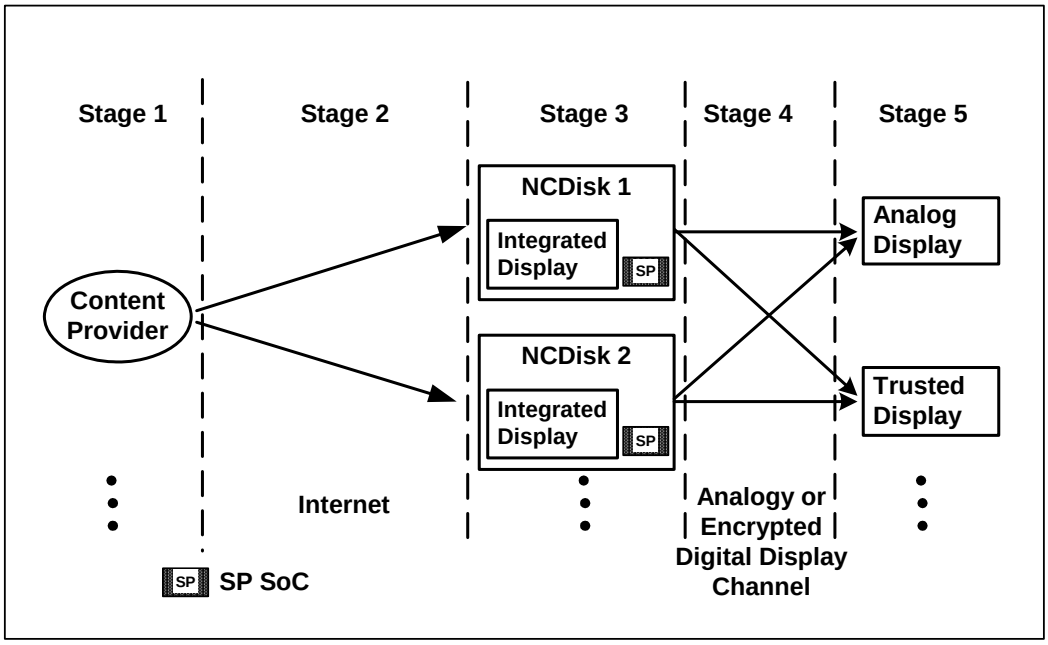



Fig 3-3. Online Movie Download Scenario using NCdisk

At stage 1, a content provider has a database of movies. The content provider sends the movies in encrypted form to its users through the public Internet (insecure) at stage 2. At stage 3, the users receive this encrypted movie. In existing DRM systems, users store these encrypted movies, along with the decryption keys on their PC or portable media players, which typically do not provide adequate protection for these keys. As a result, these movies may be easily copied and proliferated through the Internet. However, if the users store these movies onto their NCdisks, then those movies will not be easily copied in their original high-quality digital plaintext form. Instead, the movies can only be viewed through the NCdisk and one of three pre-defined type of outputs.

The NCdisk can provide both security for the content providers and convenience for the users. The NCdisk can be used to replace the current DVD-by-mail rental services such as Netflix™ and Blockbuster™. These service providers currently mail millions of DVDs to their

users each week. Instead, these service providers could mail to each user a single NCdisk. Using this NCdisk, the users can download their desired movies without having to wait for DVDs by mail, and the content providers will have the assurance that their movies cannot be mass copied in their original plaintext digital form.

We present a security protocol to use along with the NCdisk for an online movie download application, as shown in Table 3-3. When the NCdisk is manufactured, it is completely empty. It has no movies stored on it, and its SoC registers and memory are void of keys and software. The manufacturer ships these blank NCdisks to the movie content distributor, who is the "trusted third party (TTP)" in our model. The 5 paragraphs below refer to parts (a) through (e) in Table 3-3. This is also further illustrated by Figures 3-4 through 3-7.

In Table 3.3(a), the content provider first installs initialization software on the NCdisk that will self-generate a random Device Key and store it in the Device Key register, load up the trusted TSM software, write in a unique serial number (SNj) identifying the NCdisk (Dj), and load up the shared key that the content provider shares with the NCdisk. Afterwards, the content provider can remove the initialization software. Note that since the Device Key and TSM software are in ROM (write-once Flash) memory, no one can re-program the NCdisk in the future. This prevents an attacker from tampering with the NCdisk. After the NCdisk is initialized, it can be deployed to users.

**NCdisl Security Protocol**

**C**: content provider.

**M**: movie content; **i**: $i^{th}$ movie.

**D**: NCdisk; **Dj** : $j^{th}$ NCdisk; **SNj**: serial number for jth NCdisk.

**Manufacturer provides a blank Dj**

Manufacturer builds a blank NCdisk Dj that does not have any software or keys stored inside.

**(a) Manufacturer sends Dj to C for initialization**

1.C loads secure installation software into NCdisk RAM.

2. This initialization software does the following:

a. Generates a random Device Key, DKj. and writes DKj into the Device Key register, which is a non-volatile register (e.g., write-once Flash memory).

b. Loads TSM software into the write-once memory area.

c. Generates a keyed hash of the extended TSM software and stores this extended TSM software and its keyed hash in the off-chip data storage area.

d. Stores a unique SNj into the write-once memory area.

e. Generates a random key CDKj, which it shares with Dj. It encrypts CDKj using the device key and stores it in the off-chip storage area.

3.Finally, C removes the initialization software, disables the writing of the on-chip Flash memory, and the NCdisk is fully initialized.

**(b) C distributes Dj to user j**

1. User j buys Dj from a store, or C sends Dj to user j.

2. User j connects Dj online to C's website

3. C reads SNj from Dj. C has a database that associates each SNj with a CDKj, which C shares with Dj. Using CDKj, C securely sends data to Dj.

**(c) C builds a movie database**

1. C generates a random movie encryption key $CMK^i$ for each movie $M^i$.
2. C encrypts movie $M^i$ with $CMK^i$.
3. C saves $E_{CMK^i}(M^i)$ and $CMK^i$ in movie database.
4. C periodically re-encrypts $M^i$ with a new $CMK^i$

**(d) C prepares $M^i$ to send to Dj**

1. For a given $M^i$, C prepares a different $M^i$ bundle for each Dj as follows:
   a. C searches up the CDKj that it shares with Dj
   b. C generates a $MID^i_j$ identifying $M^i$ and Dj
   c. C derives a key $K^i_j = MAC_{CDKj}(MID^i_j)$
   d. C encrypts $CMK^i$ with $K^i_j$
   e. C sends the bundle for $M^i$ to Dj: $\{E_{CMK^i}(M^i), E_{K^i_j}(CMK^i), MID^i_j\}$

**(e) Dj processes bundle before storing it**

1.Dj first decrypts the bundle to obtain plaintext $M^i$:
   a. Dj re-derives $K^i_j = MAC_{CDKj}(MID^i_j)$
   b. Dj decrypts $CMK^i = D_{K^i_j}(E_{K^i_j}(CMK^i))$
   c. Dj decrypts $M^i = D_{CMK^i}(E_{CMK^i}(M^i))$

2. Dj then re-encrypts $M^i$ for storage
   a. Dj generates a random $ID^i_j$ identifying $M^i$ and Dj
   b. Dj uses device key DKj and $ID^i_j$ to derive a new movie encryption key:
      $DMK^i_j = MAC_{DKj}(ID^i_j)$
   c. Dj encrypts $M^i$ with $DMK^i_j$
   d. Dj throws away $DMK^i_j$
   e. Dj stores $M^i$ bundle, which is now non-copyable $\{E_{DMK^i_j}(M^i), ID^i_j\}$

Table 3-3. NCdisk Security Protocol

As shown in Fig 3-4 and the protocol summary in Table 3-3(b), when a user connects the NCdisk Dj to the content provider through the Internet, the content provider can read off the SNj on Dj. The content provider has a database that associates each SNj with the corresponding key that it shares with that Dj. Using this shared key, the content provider can securely send movie contents to Dj. (The Shared Key j shown in Fig, 3-4 is the CDKj in Table 3-3(b).)

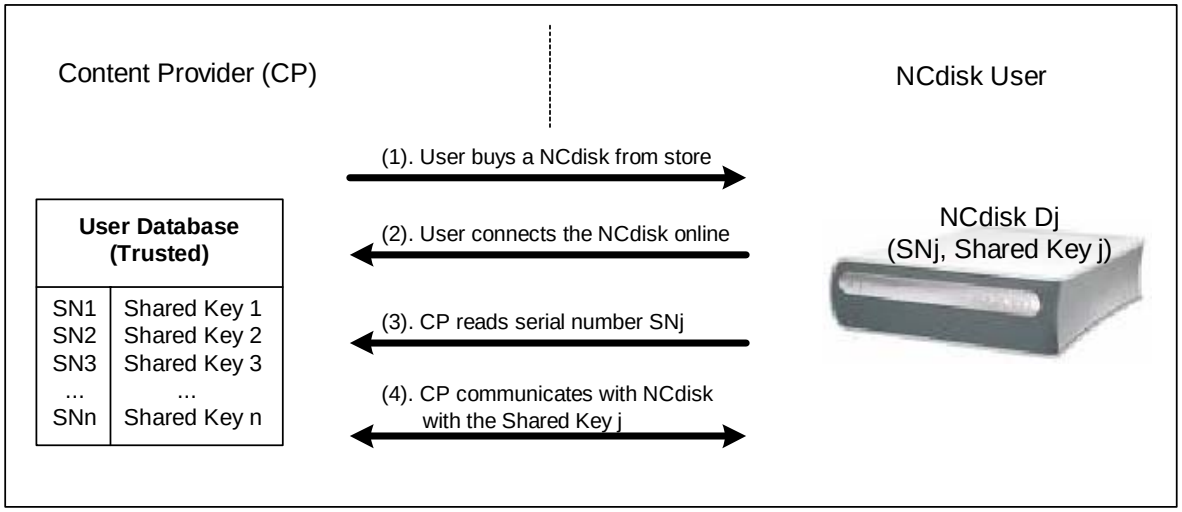



Fig 3-4. NCdisk Establishes a Secure Communication with Content Provider

Further, the content provider builds a movie database, where each movie $M^i$ is encrypted using a different movie encryption key $CMK^i$. These movies and their keys are assumed to be stored in a secure location on the content provider's server, as shown in Fig 3-5 and Table 3-3(c).

A particular movie $M^i$ is sent to numerous NCdisks. The content provider prepares a different bundle for each NCdisk Dj (see Table 3-3(d).) The bundle consists of three components. The first component is the encrypted movie, which is encrypted (once) with the movie encryption key $CMK^i$. The second component is the encrypted $CMK^i$. For different Dj, the $CMK^i$ is encrypted using a different key $K^i_j$. This key is always derived upon use and immediately deleted afterwards. Only the content provider and the NCdisk can re-derive $K^i_j$. The

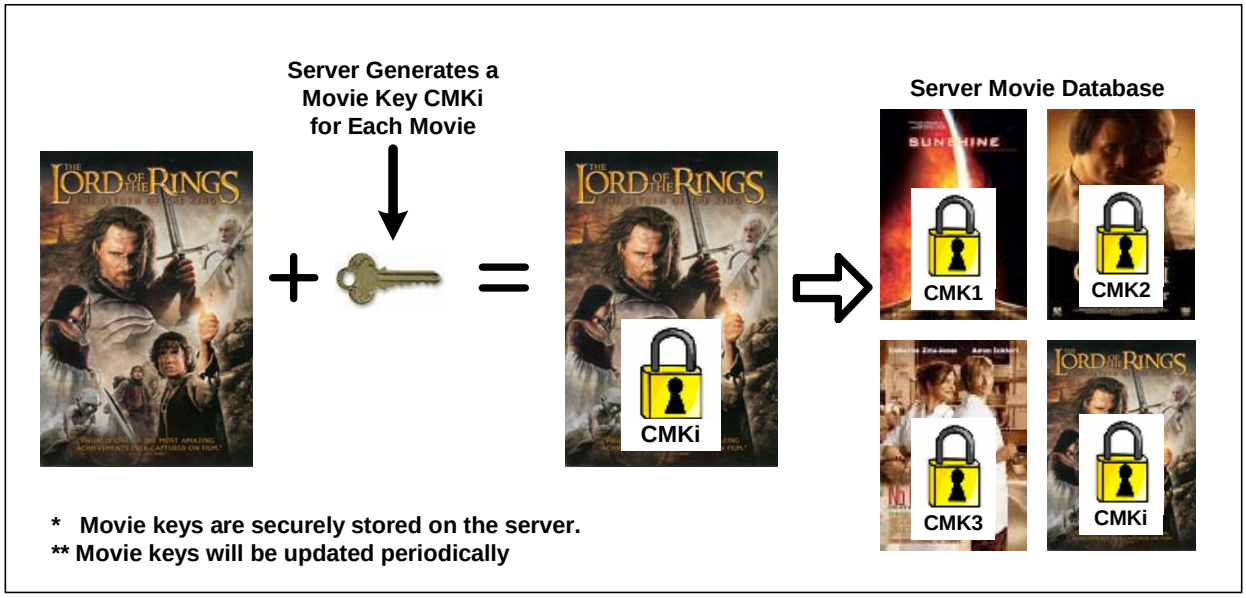



Fig 3-5. Content Provider Builds a Secure Movie Database

third component is a random value $MID^i_j$ that identifies that particular movie $M^i$ and that particular NCdisk Dj. This value plays a role in re-deriving $K^i_j$, but it is only useful to the content provider and the NCdisk. This process is described in Fig 3-6.

Table 3-3(e) describes how an NCdisk processes the encrypted movie before storing it in the NCdisk. When an NCdisk receives a movie bundle, it re-derives $K^i_j$ and uses $K^i_j$ to obtain the original movie encryption key $CMK^i$, which can be used to obtain the plaintext movie $M^i$. Next, the NCdisk re-encrypts $M^i$ using a new movie encryption key $DMK^i_j$, which is derived using the NCdisk's unique device key, Dj, as shown in Fig 3-7. At this point, no one can copy the original digital plaintext movie from the NCdisk. Using the predefined API functions, the NCdisk provides a set of controlled outputs to ensure that the original digital plaintext movie does not leak out during the output phase.

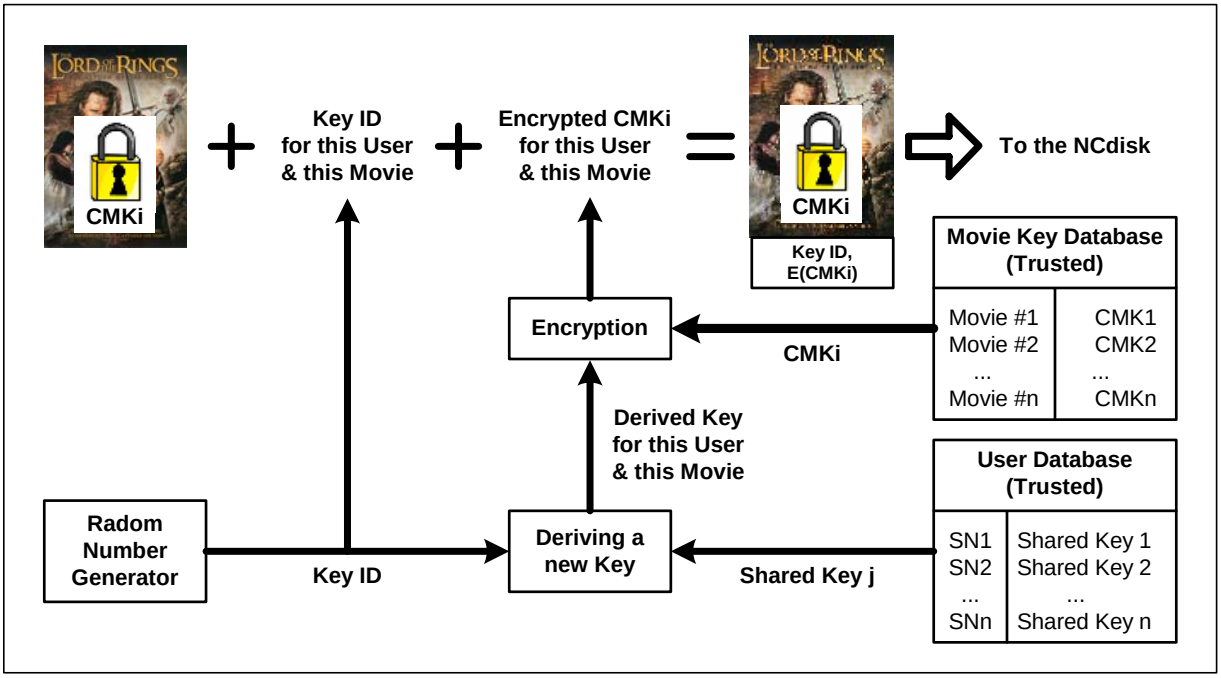



Fig 3-6. Content Provider Prepares a Movie for NCdisk

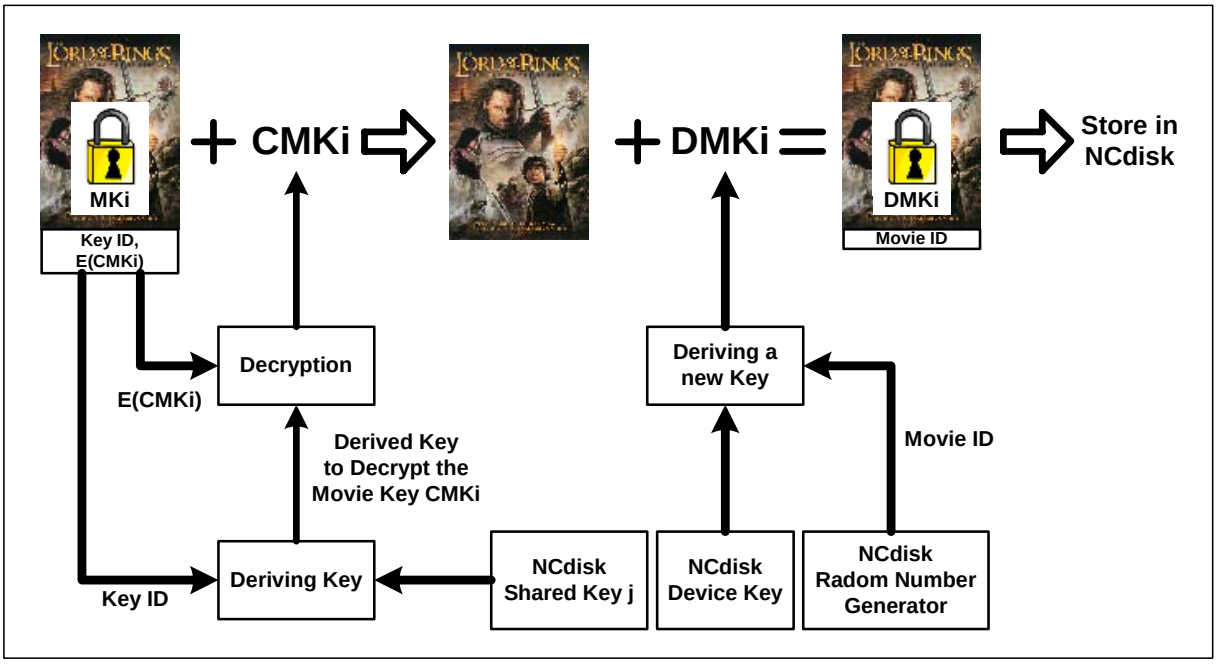



Fig 3-7. NCdisk Downloads a Movie and Stores it on NCdisk

# Chapter 4
# Business Models and Analysis

### *4.1 Application of the NCdisk*

In chapter 3, we examine that the NCdisk can be used in the online movie download application. It can be used to replace the current DVD-by-mail rental services such as Netflix™ and Blockbuster™. These service providers currently mail millions of DVDs to all their users each week. Instead, these service providers could mail to each user a single NCdisk. Using this NCdisk, the users can download their desired movies without having to wait for DVDs by mail, and the content providers will have the assurance that their movies cannot be mass copied in their original plaintext digital form. In this chapter, we analyze the prospects of turning the NCdisk technology into a business. In the sections below, we examine two different types of business models for using the NCdisk in the online movie download application and compare their competitiveness and financial projections.

### *4.2 The Business Models*

Just having the NCdisk technology is not enough for the online movie download business. We must also provide high quality digital movie contents that can be delivered to the NCdisk. There are two viable business models for achieving this.

***Business Model 1: Market NCdisk to content service providers***

(1) We form a company to develop the SP-based SoC (System on Chip). This development mainly involves programming in hardware description language (HDL) and simulating to test for its correct operation. We partner with a chip manufacturer to turn our HDL code into an actual chip. This way, as a "fables" company, we do not need to invest in manufacturing our chip.

(2) We further design the NCdisk system, which involves integrating the SP-based SoC with an existing disk architecture. We design the entire schematic and documentation. We also design and test a prototype, preferably on a FPGA. Then, we partner with a contract or OEM manufacturer, who will make the actual NCdisk product. In this scenario, the manufacturer will be responsible for the warrantee.

(3) We establish a server that runs the NCdisk security protocol (described in Chapter 3) used to securely send data content from the content provider to the customer's NCDisk. Our server does not hold any contents. We partner with a content service provider, such as Netflix, who will provide the actual movie selection, user account maintenance, billing, and customer support. Note in this case, our company does not directly get involved with the customer.

We analyze this model based on its people, context, opportunity, and deals. From a people perspective, this business model requires that our company focus on the NCdisk technology. We would mainly need people who are good in this technology area, but obviously, it would be beneficial to have people on our team who have experience in the media content industry.

From a context and opportunity perspective, there are two things that make this model appealing. First, the movie industry wants to get into the online movie download business

because they know that this is what consumers will want in the future. However, they are reluctant to put movies on the Internet because there is insufficient technology to protect the contents. The NCdisk, if it works, goes a long way in solving this problem. Further, two major DVD rental companies, Netflix and Blockbuster, have made it public that they wish to enter the online movie download industry. These two companies are at constant battle with each other for customers. The NCDisk seems like the appropriate device that will give one company a major advantage against the other company. This relates to the deal perspective. With the Netflix and Blockbuster tensions, we may be able to get a good deal if we talk to both sides and take the better offer. However, this type of business model involves taking money from Netflix or Blockbuster instead of directly from the end-users. The extra level of separation reduces our revenue, as shown in financial analysis later.

***Business Model 2: Market NCdisk directly to users as a new content service provider***

(1) Same as Model 1

(2) Same as Model 1

(3) We establish an online store to sell and rent digital contents, such as movies, TV shows, music, digital books, etc. We take a revenue sharing model with the content provider. In this case, we will provide the actual movie selection, user account maintenance, billing, and customer support. Note in this case, our company directly sells the service to the end-user. We no longer play a middle-man role.

We analyze this model based on its people, context, opportunity, and deals. From a people perspective, our company requires three types of skill sets: (1) technical (2) web merchandise system developer, and (3) marketing and business skills in media content area. Compared to

business model 1, this model requires more skill sets and manpower. From context and opportunity perspective, we will directly compete with existing content service providers, such as Netflix. We have a good technology, but as we examine below, we are not the first to market since there are similar types of devices already out there. From a deal perspective, we will be making deals directly with the content providers and we will hopefully try to enter a revenue sharing model. We will get a larger cut of the pie than with the business model 1.

### *4.3 The Competition*

For business model 1, the competition is mainly the content service providers themselves. For example, if we try to license our NCdisk to Netflix, we face the alternative that Netflix may wish to develop their own technology. Netflix currently has a 50-engineer team to develop their online downloading solution. Will they want to partner with us?

For business model 2, we have direct competition with content service providers, such as Netflix and Blockbuster, since we will also be a content service provider. Compared to DVD mailing, the online movie download strategy is more convenient for users and cheaper for us. The question is when Netflix or Blockbuster will come out with their online download service. The only way to take a large market share is to beat Netflix or Blockbuster to the market in developing a secure and convenient online download product and service, and to provide a more compelling on-going advantage.

### *4.4 The Financial Projections*

Now we analyze the income statement for both business models 1 and 2.

For model-1, we assume we partner with Netflix. The major revenue comes from selling the NCdisk at a price of $120. Either Netflix or the end-user customer will pay for this device. Another major source of revenue is in the 10% revenue share with Netflix. We assume that each customer on average pays a monthly fee of $15, and we hope to take $1.5 of that fee. The bottom line is that existing Netflix cost on US mailing fees is $100M/year and existing mailing facilities and employees is another $100M/year. Our revenue cannot exceed this $200M/year. Note that even this $200M is not a likely maximum because Netflix will not be completely stopping their DVD mailing business immediately.

For model 2, we first analyze several key components of revenue and expense sources in order to get the numbers for the income statement. The first number is how much we can charge customers in a monthly fee. By my renting experience from both Netflex and Blockbuster, a customer can mail about 4-6 times each month. Each mail has 3 DVDs if users sign on a $17.99 program. This is equivalent to $1-$1.5 for each movie. We may set $19.99 for 20 movies per month, and $1.49 for each movie after the 20 movie limitation.

The most important number is how much it costs to get movie content from the movie studios. Based on the marketing study of Wharton, DVD rental companies such as Netflix and Blockbuster have two ways to get movie contents. One way is to buy a DVD with a copyright of renting and reselling. The average cost is about $60 for each DVD disk (not each title). Another way is the revenue-sharing model. DVD rental companies pay about $8 per DVD upfront and then pay 30-45% of revenue earned from the DVD renting. The Wharton's study show that, the revenue-sharing model increases revenue by 75% during the market test period. The reason is that customers have more titles to select and so they rent more. For business model 2 of the NCdisk business, we assume a 50% revenue sharing without upfront pay with the movie studios.

It is reasonable to assume no upfront fee for our model because our sharing percentage is higher. Further, note that our sharing revenue is equal to 50% of our renting revenue minus the cost of the NCdisk.

Moreover, there is more expense for website development and customer support in model 2 than model 1. However, the resources needed are proportional to the customer size and revenue income. As shown in the income statement, we increase our support staff only as our customer base increases.

Note that the breakeven point for both models is less than 100K customers. Also, note that model 2 is three times more profitable than model 1. The income statements are shown below.

| **Model 1 - We provide NCDisk for Netflix** | **1st year** | **2nd year** | **3rd year** | **4th year** | **5th year** |
|---|---|---|---|---|---|
| (assume Netflix or user pays for NCDisk) | | | | | |
| | | | | | |
| Accumulated number of NCDisk users | 0 | 10 | 100 | 500 | 1,000 |
| New added NCDisk users | | 10 | 90 | 400 | 500 |
| Sales revenue items: | | | | | |
| NCDisk sales, $120 each | 0 | 1,200 | 10,800 | 48,000 | 60,000 |
| NCDisk protection, $3.99/mon, 50% enroll | 0 | 240 | 2,400 | 12,000 | 24,000 |
| Revenue share with Netflix, 10% of $15/mon | 0 | 180 | 1,800 | 9,000 | 18,000 |
| Total revenue | 0 | 1,620 | 15,000 | 69,000 | 102,000 |
| Less cost of sale, 50% of NCDisk sale price | 0 | 600 | 5,400 | 24,000 | 30,000 |
| Gross margin | 0 | 1,020 | 9,600 | 45,000 | 72,000 |
| Operation expense items: | | | | | |
| Employee salary, including benefits | 2,000 | 2,000 | 3,000 | 4,000 | 5,000 |
| Equipment | 500 | 500 | 1,000 | 1,000 | 1,000 |
| Office renting | 100 | 100 | 300 | 300 | 500 |
| NCDisk RMA, 80% of protection revenue | 0 | 192 | 1,920 | 9,600 | 19,200 |
| Web server cost | 100 | 200 | 500 | 500 | 500 |
| Other cost (travel etc.) | 500 | 500 | 1,000 | 1,000 | 1,000 |
| Total operation expense | 3,200 | 3,492 | 7,720 | 16,400 | 27,200 |
| Incoming before tax | -3,200 | -2,472 | 1,880 | 28,600 | 44,800 |
| Provision for income tax, 50% rate | 0 | 0 | 940 | 14,300 | 22,400 |
| Net incoming, in thousand USD | -3,200 | -2,472 | 940 | 14,300 | 22,400 |

| **Model 2 - We directly provide Movie renting** | **1st year** | **2nd year** | **3rd year** | **4th year** | **5th year** |
|---|---|---|---|---|---|
| **(assume NCDisk is free)** | | | | | |
| | | | | | |
| Accumulated number of NCDisk users | 0 | 10 | 100 | 500 | 1,000 |
| New added NCDisk users | | 10 | 90 | 400 | 500 |
| Sales revenue items: | | | | | |
| NCDisk protection, $3.99/mon, 50% enroll | 0 | 240 | 2,400 | 12,000 | 24,000 |
| Movie rental revenue, $19.99/mon | 0 | 2,400 | 24,000 | 120,000 | 240,000 |
| Non-movie rental revenue, 20% of rental revenue | 0 | 480 | 4,800 | 24,000 | 48,000 |
| Total revenue | 0 | 3,120 | 31,200 | 156,000 | 312,000 |
| Less cost of sale, $60 per NCDisk | 0 | 600 | 5,400 | 24,000 | 30,000 |
| CP rev share, 50%*(rentRev – NCDiskcost) | 0 | 900 | 9,300 | 48,000 | 105,000 |
| Gross margin | 0 | 1,620 | 16,500 | 84,000 | 177,000 |
| Operation expense items: | | | | | |
| Employee salary | 2,500 | 2,500 | 4,000 | 6,000 | 8,000 |
| Equipment | 500 | 500 | 1,000 | 1,000 | 1,000 |
| Office renting | 100 | 100 | 500 | 500 | 1,000 |
| NCDisk RMA, 80% of protection revenue | 0 | 192 | 1,920 | 9,600 | 19,200 |
| Web server | 200 | 300 | 500 | 1,000 | 2,000 |
| Customer support | 0 | 200 | 500 | 1,000 | 2,000 |
| Other cost (travel etc.) | 500 | 500 | 1,000 | 1,000 | 1,000 |
| Total operation expense | 3,800 | 4,292 | 9,420 | 20,100 | 34,200 |
| Incoming before tax | -3,800 | -2,672 | 7,080 | 63,900 | 142,800 |
| Provision for income tax, 50% rate | 0 | 0 | 3,540 | 31,950 | 71,400 |
| Net incoming, **in thousand USD** | -3,800 | -2,672 | 3,540 | 31,950 | 71,400 |

Table 4-1. Financial Analysis for the NCdisk Business Models (units in thousand USD)

In table 4-1, the major marketing data is referenced from the Netflix annual report and University of Pennsylvania Wharlton's marketing research paper [8-10].

### *4.5 Next Step for Business*

The rough financial analysis shown above suggests that Business Model 2 provides far greater revenue than Business Model 1. Also, Model 2 provides more control over the company for the founders than does Model 1. However, Model 2 has greater competition since we will be

competing directly with the more established content providers. The exercise of working through the financial projections above provides a good first step to analyzing the prospects of the NCdisk business. As our work in Chapters 2, 3, and 4 show, we are fairly confident of the technology, but in order to better understand the real need of the online movie download market, the next step should be to talk to real customers, content providers, and content service providers.

# Chapter 5
# Conclusions and Future Work

We propose an NCdisk concept to prevent copying of protected digital content. The NCdisk concept boils down to implementing two design goals: protecting secrets and providing output control. We achieve these two goals by implementing a (Secret-Protection) SP-based SoC architecture that can be added to existing disk architecture to turn that disk into an NCdisk. Further, we design a security protocol that can be used along with the NCdisk to provide security and convenience for the online movie download application.

Future work includes extending the NCdisk architecture and security protocol to support multiple content service providers. This means that the NCdisk has to be shared by mutually-distrusting content service providers. Future work also includes investigating how the NCdisk can be extended to support other applications besides online movie download. We would like to research using the NCdisk to provide more copy-protection and privacy-protection for computer data.

# References


[1] “Architecture of Windows Media Rights Manager”, Microsoft Corporation, May 2004. http://www.microsoft.com/windows/win dowsmedia/howto/articles/drmarchitecture.aspx

[2] Ruby B. Lee, Peter C. S. Kwan, John Patrick McGregor, Jeffrey Dwoskin, and Zhenghong Wang, “Architecture for Protecting Critical Secrets in Microprocessors,” Proceedings of the 32nd International Symposium on Computer Architecture (ISCA 2005), pp. 2-13, June 2005.

[3] Jeffrey S Dwoskin, Ruby B. Lee, "Hardware-rooted Trust for Secure Key Management and Transient Trust", *ACM Conference on Computer and Communications Security*, pp. 389-400, October 2007.

[4] Jeffrey Dwoskin, Dahai Xu, Jianwei Huang, Mung Chiang, Ruby Lee, "Secure Key Management Architecture Against Sensor-node Fabrication Attacks", *IEEE GlobeCom 2007*, November 2007.

[5] Michael Wang and Ruby Lee, “Architecture of Non-Copyable Disk (NCdisk) Using Secret-Protection (SP) SoC Solution”, Forty-First Asilomar Conference on Signals, Systems and Computers, November 4-7, 2007.

[6] James Jeppesen et al., “Hard Disk Controller: the Disk Driver’s Bain and Body”, 0-7695-1200-3/01, 2001 IEEE.

[7] Steve Jobs, “Thoughts on Music”, http://www.apple.com/hotnews/thoughtsonmusic/

[8] Knowledge@Wharton, UPenn, “The Home-video Market: Who Rents, Who Buys and Why”, February 08, 2006.

[9] George Knox, Jehoshua Eliashberg, “The Consumer’s Rent vs. Buy Decision: The Case of Home-Video”, Marketing Department, The Wharton School,University of Pennsylvania, Feb 2005

[10] Knowledge@Wharton, UPenn, “Now Showing at Blockbuster: How Revenue-sharing Contracts Improve Supply Chain Performance”, October 16, 2000.

[11] S. W. Smith and S. H. Weingart, "Building a High-Performance, Programmable Secure Coprocessor," *Computer Networks*, 31(8), pp. 831-860, April 1999.

[12] Suh, G. E., O'Donnell, C. W., Sachdev, I., and Devadas, S. 2005. “Design and implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions,” In Proceedings of the 32nd Annual international Symposium on Computer Architecture (June 04 - 08, 2005).

[13] J. D. Tygar Bennet Yee “Dyad: A System for Using Physically Secure Coprocessors,” Carnegie Mellon University Technical Report CMU-CS-91-140R, May 1991.

[14] “TCG Specification Architecture Overview,” Specification Revision 1.3, 28 March 2007, https://www.trustedcomputinggroup.org/groups/TCG_1_3_Architecture_Overview.pdf

[15] Naor, M.; Pinkas, B.; “Efficient Trace and Revoke Schemes,” Financial Cryptography ’2000, LNCS 1962. Springer-Verlag, New York. 1-20.

[16] Taban, G; Cardenas, A; Gligor, V; “Towards a Secure and Interoperable DRM Architecture,” DRM ’06, 30 October 2006. Alexandria, Virginia.

[17] Boneh, D., “The Decision Diffie-Hellman Problem,” Proceedings of the Third Algorithmic Number Theory Symposium, LNCS Vol. 1423, Springer, pp. 48-63, 1998.

[18] Grimm, R, “Privacy for Digital Rights Management Products and their Business Cases,” University Koblenz and Fraunhofer Institute for Digital Media Technology, 2005.

[19] Anlauff, M; Pavlovic, D.; “Pda – the Protocol Derivation Assistant,” Kestrel Institute, 17 July 2006

[20] Ruby B. Lee and A. Murat Fiskiran, PLX: An Instruction Set Architecture and Testbed for Multimedia Information Processing, *Journal of VLSI Signal Processing 40*, 85-108, 2005.

[21] Ruby Lee and Michael Wang, “Resolving Encoding Issues in Combining PAX and PLX Instruction Sets”, PALMS, EE Dept Princeton University, Technical Report No. CE-L2007-007.

[22] Michael Wang, Ruby Lee, “PAX-PLX 1.0 Encoding”, PALMS Lab, Princeton University, July, 2006

[23] Michael Wang, Ruby Lee, “PAX-PLX 1.0 Reference”, PALMS Lab, Princeton University, July, 2006

[24] Michael Wang, Ruby Lee, “PAX 1.1 ISA Encoding”. PALMS, EE Dept. Princeton, July, 2006.

[25] Michael Wang, Ruby Lee, “PAX 1.1 ISA Reference”. PALMS, EE Dept. Princeton, July, 2006.

[26] Michael Wang, “PAX Simulator, Assembler, and Linker: Building a Toolset for a New Processor ISA Based on the SimpleScalar Simulator and GNU Toolset”. Junior Independent Work Final Report, EE Dept. Princeton University, January, 2007.

[27] B. Schneier, Applied Cryptography: Protocols, Algorithms, and Source Code in C, John Wiley and Sons 1996.

[28] A Murat Fishiran, Ruby Lee, "PAX: A Tiny Scalable Cryptographic Processor for Wireless Devices and Servers", *Princeton University Department of Electrical Engineering Technical Report CE-L2004-002*, November 2004.

[29] Austin, Todd; Burger, Doug, "SimpleScalar Toolset, Version 2", <http://www.simplescalar.com/docs/users_guide_v2.pdf>.

[30] Austin, Todd, "A User's and Hacker's Guide to the SimpleScalar Architectural Research ToolSet", January, 1997, <http://www.simplescalar.com/docs/hack_guide_v2.pdf>.

[31] Kegel, Dan, "Building and Testing gcc/glibc cross toolchains", <http://www.kegel.com/crosstool/>.

[32] "ARM7TDMI-S Technical Reference Manual", < http://www.arm.com/pdfs/DDI0234A_7TDMIS_R4.pdf >

[33] Xilinx, Inc. "EDK Concepts, Tools and Techniques", Version 9.1i.

[34] Xilinx, Inc. "Xilinx University Program Virtext Pro Development System Hardware Reference Manual", March, 2005.

# Appendix A
# Alternative Solution for NCdisk

The SP-based SoC design of the NCdisk described in Chapter 3 represents the latest version of the NCdisk. In this appendix, we describe an earlier design of our NCdisk architecture that is ASIC-based, and compare it with the SP-based version.

***ASIC-based NCdisk Architecture***

This earlier version of the NCDisk Hardware Architecture, Figure 4-1, is an ASIC-based co-processor design. It was designed for the online movie download application – and hence has a similar design goal as the SP-based NCdisk described in chapter 3. However, its implementation is different and more complex. It has a control processor (CProc), which runs software that controls the input/output of the movie files, as well as other general purpose routines. The NCdisk also has a secure processor (SProc), which is part of a SoC that runs the security software. The NCdisk has an untrusted hardware module that consists of a regular disk, the CProc, a USB controller used to connect the NCdisk to a PC, an Ethernet controller used to connect the NCdisk to a home network router, and a wireless transceiver used to control the NCdisk with a remote-control. The movie files are stored on the regular disk. They are encrypted by security software and critical secrets that are stored on the SoC.

*A.1. Security Assumptions and Definitions for online movie download application*

Prior to looking at the ASIC-based architecture in detail, it is important that we clearly define our security assumptions and goals for the online movie download application.

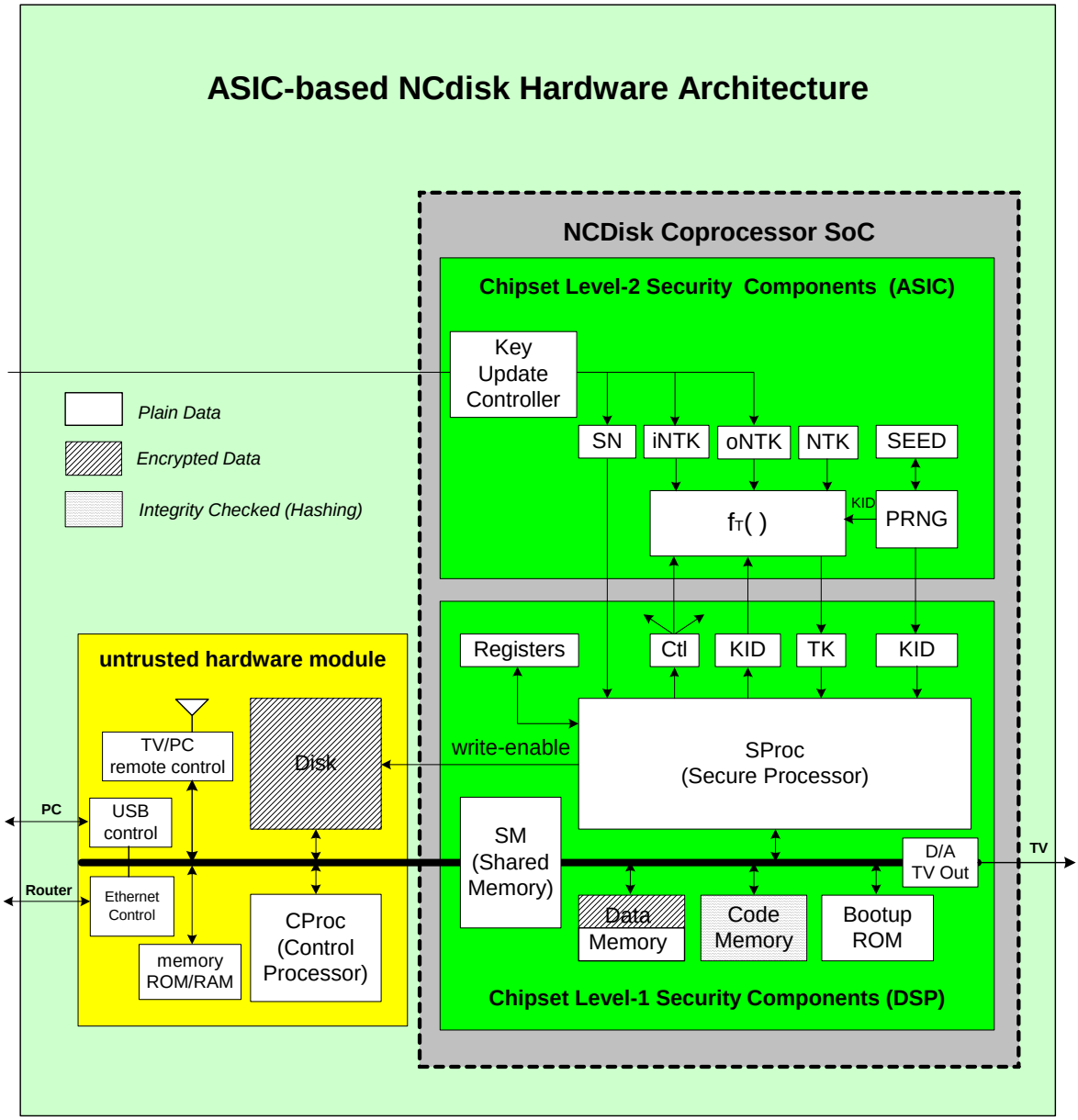



Fig A-1 ASIC-based NCdisk Hardware Architecture

First, we assume that the existing encryption algorithm, hashing function, and random key generator that we use are cryptographically strong. We also assume that the standard cryptographic protocol, such as SSL, for two-party authentication between the server and the NCdisk is secure. Moreover, we define that an encrypted movie file is broken if the movie encryption key is compromised by an attacker. We define that an NCdisk is broken if the critical secret of the NCdisk is compromised by an attacker. Moreover, an attacker can clone a device if he can obtain the critical secret of the device, in which case the attacker could produce multiple copies of NCdisks that contain the same broken critical secret.

Further, we assume that physically opening up a SoC chipset and probing the internal bus without damaging the chipset is a difficult but still feasible task for professionals. If the internal bus can be probed, then the software and data memory on the chip can be read out. This can be accomplished by connecting the internal bus to an external processor or instrument. Further, if the internal processor can be used to run attacker software, then the internal register contents can be read out through the internal bus. At this point, we are ready to give two security level definitions:

***Chipset Level-1 Security Definition:*** *a particular on-chip component has Chipset Level-1 Security if an attacker has to physically open up the chipset and probe the bus to obtain data from the component by either running software on the internal processor or external processor.*

***Chipset Level-2 Security Definition:*** *a particular on-chip component has Chipset Level-2 Security if an attacker may not obtain data from the component even if he physically opens up a chip, probes the bus, and runs software on the internal processor or external processor. The only*

*way to obtain data from the component is to perform gate level reverse-engineering of the chipset.*

The security goals of the NCdisk architecture are:

(1) Store the critical secrets of the NCdisk in Chipset Level-2 Security hardware. In other words, an attacker may not obtain the critical secrets of the NCdisk (and thereby clone the NCdisk) without performing gate level reverse-engineering of the chipset.

(2) Breaking one movie in the server database should not affect the security of other encrypted movies. In other words, if an attacker obtains the encryption key of one movie, that key should not leak information about the encryption keys of other movies.

(3) Breaking one NCdisk should not affect the security of other NCdisks. In other words, if an attacker obtains the critical secret of one NCdisk, that information should not leak the critical secrets of other NCdisks.

(4) Breaking one movie file on an NCdisk should not affect the security of other movie files on the same NCdisk. In other words, if an attacker obtains the encryption key of one movie file on the NCdisk, that key should not leak information about the encryption keys of other movie files on the same NCdisk.

(5) Only a predefined a set of player devices are able to receive the NCdisk output data.

(6) When a particular pirated movie file is found, it should be traceable to the specific NCdisk. Then, that NCdisk should be revoked.

Same as SP-based NCdisk architecture in chapter 3, these security goals also could be narrowed down to two major points: storing and protecting critical secrets, and controlling of output.

*A.2. Storing and Protecting Keys*

The security measures of storing and protecting keys are provided by the SoC in Fig A-1. The SoC contains various Chipset Level-1 Security components. It contains a boot-up ROM, which is the starting point of all code executed by SProc. The SoC also contains RAM that stores most of the secure software. This software in RAM is integrity-checked each time by ROM code before being executed. Further, the SoC contains data RAM memory. Part of this memory may be encrypted to store some long-term secrets. The other part of the memory may contain intermediate data of currently running security software. The SoC also contains on-chip registers to store intermediate data of any programs running on the SProc. These register data may spill over into the RAM data memory. Also, the SoC contains a D/A converter that immediately converts an unencrypted movie file into analog streams that can be outputted to a TV. Note that the method of using secure ROM and integrity-checked RAM to secure software is similar to the method used in extended sensor-node SP [4] and IBM Co-Processor [11].

The security protocol is similar to that in Chapter 3. The NCdisk device key DKj has three different types of iNTK, NTK, and oNTK. The iNTK is used to secure communication between the server and the NCdisk. The oNTK is used to secure communication between the NCdisk and a trusted player device. The NTK is used to protect the movie files on the NCdisk. The NTK is known only to the NCdisk. Not even the server or the legitimate user of the NCdisk knows this key. Further, the SoC stores these critical keys (NTK, iNTK, and oNTK) in Chipset Level-2 Security hardware. These three keys cannot be broken even if the SoC chipset is opened, or the bus is probed, or the SProc is used to run malicious software. Hence, it prevents cloning the NCdisk. Further, the iNTK and oNTK are initially installed and may be updateable by the

content provider. The updating process is securely executed by the Key Update Controller, which works as follows:

(1) The iNTK and oNTK may only be updated by the server, who knows the old key.

(2) The server first inputs the old key value for iNTK (or oNTK).

(3) The server then inputs the new key value for iNTK (or oNTK).

(4) The Key Update Controller uses hardware logic circuits to compare the inputted old key value with the existing key value of the iNTK (or oNTK).

(5) If the two key values are the same, then the existing iNTK (or oNTK) value is replaced by the new inputted key value.

(6) If the two key values are not the same, then the Key Update Controller runs a hardware delay before resetting the controller. This is to ensure that an attacker may not feasibly devise an automated way to keep guessing the old key value by brute force.

In addition, these three critical keys are never directly used to encrypt a movie file. Instead, they are used as input of a cryptographic hash function to derive different encryption keys for different movies. This further ensures that the critical keys are not leaked out.

The key derivation function is the $f_T( )$ function block shown in Figure A-1. The $f_T( )$ function is same as MAC function in chapter 3. This function takes as input a critical key (iNTK, oNTK, or NTK) and a movie ID (a random Key ID). Then, the $f_T( )$ function outputs a derived key TK for a particular movie. It is important that the TK values are checked to be cryptographically strong before they are used.

*A.3. Controlled Predefined Output*

No data and software inside a SoC should be accessible from outside of the SoC. Only a predefined player device can receive the output data through a set of secure API functions. Moreover, the CProc and the SProc do not share a common bus. Instead, they are separated by a shared memory module. The shared memory completely isolates the internal bus of the SoC from the outside. The communication from the CProc to the SProc is shown in Figure A-2. As described above, the secure boot up ROM and the integrity-checked RAM code ensures that only the secure software on the SoC can run on the SProc. The CProc may only call the SoC to execute a particular function through the shared memory. The CProc only knows the function IDs and the input/output data formats of the secure software library. The SProc processes the value in shared memory and outputs its result back into the shared memory. The CProc reads this result. The step-by-step communication is described as follows:

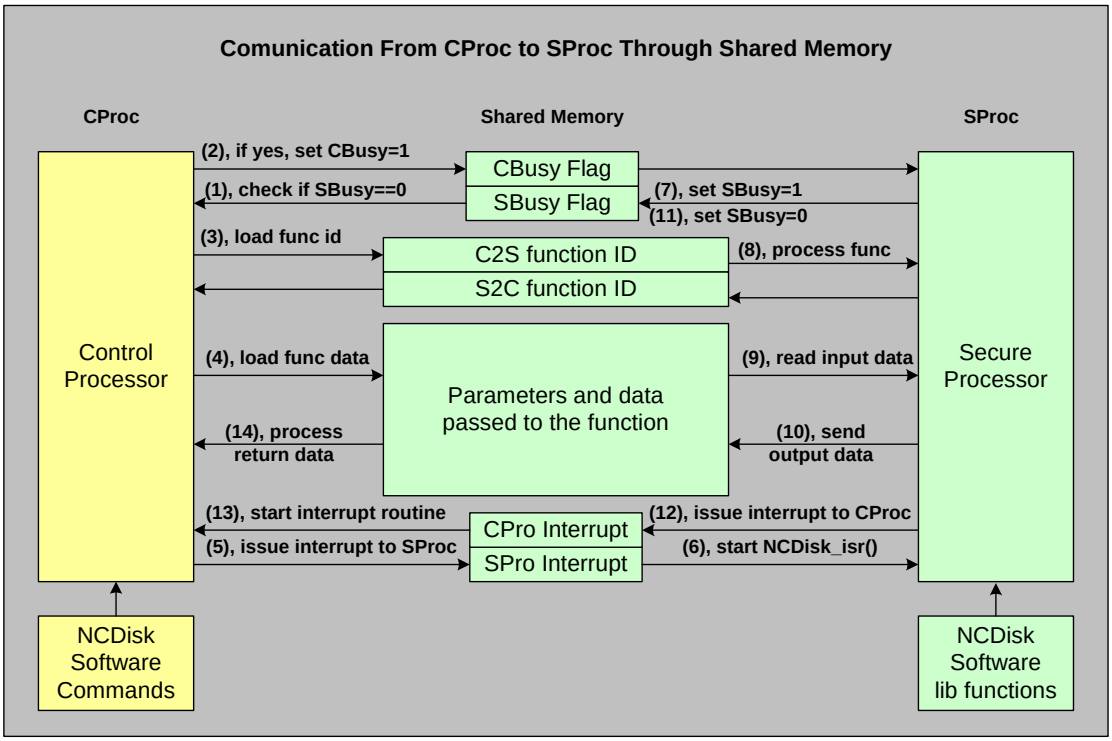



Fig A-2. Communication Between CProc and SProc through Shared Memory

(1) CProc checks the “SBusy” bit. If this bit is set, then it means that SProc is running and is using the shared memory. In this case, the CProc must wait until the bit is cleared by SProc.

(2) CProc sets the “CBusy” “CBusy” bit to claim the ownership of the shared memory.

(3) CProc loads the function ID into the “C2S function ID” word in shared memory.

(4) CProc loads the function parameters and data into shared memory. The data format is predefined for each function.

(5) CProc issues an interrupt to SProc, at which point the SProc starts to process the value in shared memory. The CProc should now clear the “CBusy” bit to release the use of the shared memory to SProc.

(6) SProc runs the interrupt service routine NCdisk_irs(), which is stored in the secure ROM.

(7) The NCdisk_irs() routine sets the “SBusy” bit to claim ownership of the shared memory.

(8) The NCdisk_irs() routine then calls the NCdisk_main() function, which validates the function ID and data format.

(9) If the shared memory input is validated, the NCdisk_main() function uses the input data to execute the function.

(10) The output of the function is sent back into shared memory based on a predefined format.

(11) SProc clears the “SBusy” bit to release the ownership of the shared memory.

(12) Sproc issues an interrupt to CProc

(13) CProc runs an interrupt service routine to process the data returned by SProc.

The above steps describe the communication from the CProc to SProc. The communication can also occur in the other direction. In other words, the SProc may also call the CProc to execute a particular function, such as error reporting and debugging functions.

The NCdisk software library functions are divided into two subsets. One is the set of core functions located in ROM. The other is the set of extended functions located in RAM. RAM functions are not allowed to directly run on SProc. RAM functions can only be called by ROM functions. The RAM code is integrity-checked by ROM. Since no one can change ROM code, the security of the system is ensured. Note that attackers are allowed to read the ROM code (although this is very difficult), but even if they do so, they will not be able to change anything. A description of some main core functions is detailed below:

- NCdisk_bootup( ) – SProc boot up function, in ROM
    - Perform boot up procedure
    - Initialize NCdisk_irs( ) interrupt routine
    - All functions run on SProc must go through the NCdisk_irs( ) routine.

- NCdisk_irs( ) – interrupt service routine, in ROM
    - This function is called by the interrupt signal issued by CProc
    - Set the SBusy flag
    - Call NCdisk_main( )
- NCdisk_main( ) – SProc main function. Links to all functions in library, in ROM.
    - Validate the command and data format in shared memory
    - If the function is in ROM and the data range is correct, call the function

- If the function is in RAM, check the function integrity, then call the function

- NCdisk_update( ) – allows server to add or delete library function in RAM, in ROM
  - Parameters
    - A flag describing whether function is to be added or deleted from RAM
    - function name and id
    - start and end memory address in RAM
    - The number of total blocks passing through shared memory to transfer this new function to the SProc RAM.
    - current block index
    - hash value of the function using iNTK, block by block
  - Functions
    - if deleting an existing function from RAM, verify that all parameters match to the exiting function in RAM
    - if add a new function, check if it matches predefined rule (such as memory mapping etc)
    - check hash value for each block in shared memory by using iNTK
    - Only server knows how to interface the new function to ROM function.
- NCdisk_reset( ) – reset NC Disk to original manufacturer setting, in ROM
  - Reset all setting to original manufacturer setting

- NCdisk _reencrypt() – decrypts and re-encrypts a movie file
    - Parameters
        - Input encrypted movie frame
    - Functions
        - Decrypts and re-encrypts the movie frame in one algorithm such the intermediate plaintext data is never stored anywhere.

- NCdisk_Convert() – decrypts and converts a movie file to a playable format
    - Parameters
        - Input encrypted movie frame
    - Functions
        - Decrypts and converts the movie frame to a playable format in one algorithm such the intermediate plaintext data is never stored anywhere.

*A.4. Comparison between SP-based and ASIC-based NCdisk Architecture*

For both the SP-based and ASIC-based NCdisk architecture, the design goals were the same: to protect critical secrets and to provide a controlled set of predefined outputs. The SP-based architecture is able to achieve these goals by taking a single processor and adding a minimal set of hardware additions onto that processor. On the contrary, the ASIC-based architecture requires a co-processor design, which is more costly. Further, the SP-based architecture uses a Trusted Software Module (TSM) that allows the specific application of the SP-based processor to be flexibly defined. On the other hand, the ASIC-based architecture restricts the application of the

SoC to be fixed to the NCdisk application. A benefit of the ASIC-based architecture is that it can be implemented into an SoC by connecting an existing processor to some ASIC logic. In contrast, implementing the SP-based SoC architecture requires changing an existing processor, although the change is only minimal.

A lot of the work done by special-purpose hardware in the ASIC-based architecture is done by the Trusted Software Module (TSM) in the SP-based architecture. In fact, all the cryptography processing and disk-controller processing can be done by the single processor in the SP-based model. We also examined the use of PAX, a general-purpose processor with special features for acceleration of cryptographic processing for this SP-based NCdisk processor. We describe the work on implementing PAX in Appendix B.

# Appendix B
# Implementation of PAX Processor

This appendix describes a two year long project that I have been working on in parallel with the NCdisk project. It is apparent that the processor used in the SoC of the NCdisk must perform a significant amount of encryption and decryption of movie contents. At the same time, the processor must also perform other general purpose functions. This project involves working on the PAX processor which has special features for accelerating cryptographic processing [21-26,28] . In the sections below, we examine my work in encoding the ISA of PAX, developing the software toolset for PAX, designing the VHDL code for the Parallel Table Lookup Unit (PTLU) of PAX, mapping the AES algorithm to PAX implementations with different word sizes, and implementing PAX on a Virtex-II Pro FPGA.

## *B.1 Encoding of PAX and PLX Processors*

PAX [28] and PLX[20] are two small, general-purpose instruction set architectures (ISA) designed by Prof. Ruby Lee and students at Princeton University, Department of Electrical Engineering. PAX is a word-size scalable processor designed to be a simple yet high-performance ISA for cryptographic processing, while PLX is a fully subword-parallel processor designed to be a simple yet high-performance ISA for multimedia information processing. This section describes the challenge in combining the two processors into one processor, and encodes a unique ISA set that covers both processors' instructions.

### *B.1.1 Background*

There are many applications, such as cell phones, laptops, palm pilots, etc, that need both cryptographic and multimedia processing. Since PAX and PLX are built-from-scratch processors designed to be fast and power-efficient at these two functions, it is desirable to combine these two instruction sets into one processor. Further, since the PLX instructions have already been encoded, we encode the PAX instruction set on top of the existing PLX encoding. Both PAX and PLX are encoded with 32-bit instructions. Six of the bits are designated for opcodes, giving 64 possible opcodes. The PLX instruction set takes up only a fraction of the 64 available opcodes, and so the PAX instruction set can be mapped into the remaining empty opcodes and the subops of the PLX encodings. During the combining of PAX and PLX, some PAX instructions sacrifice some functionality. Below, we examine the rationale behind the PAX-PLX instruction set combination. We discuss the two major encoding issues in combining PAX and PLX.

### *B.1.2. Major Encoding Issues*

#### *Making Predication Compatible*

A fundamental problem of combining the PAX and PLX instruction sets is that PLX uses predication, while PAX does not. The novel predication method used in PLX greatly reduces the performance degradation caused by conditional branch instructions during multimedia processing. However, predication is not significantly helpful for the cryptographic processing of PAX, which uses a simple set of conditional branch instructions. To implement predication, PLX has 128 1-bit predicate registers, which are divided into sixteen groups. At any one time, only one group of 8 registers is active. The first three bits of any PLX instruction is used to select one of the eight active registers. Then, the rest of the instruction executes only if the chosen register

has a value of 1. Since PAX does not utilize predicate registers, it can use the first three bits of an instruction for something else, such as an additional immediate field or subop field. However, to combine PAX and PLX instruction sets in one processor, the instructions must be encoded in such a way that hardware can easily differentiate between predicated and non-predicated instructions. We explore three different methods of achieving this and rationalize our method of choice.

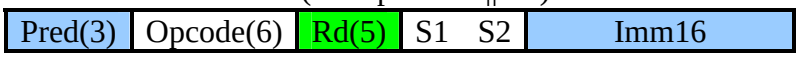
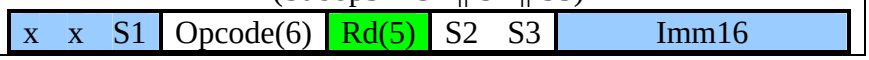
| Instructions | How is it affected? |
|---|---|
| *ALU Immediate*:<br>addi<br>subi<br>andi<br>ori<br>xori<br>*Memory Access*:<br>load<br>store | PLX version only requires 13-bit immediate field:<br>  <br>Non-predicated PAX version requires 16-bit immediate field:<br>(Imm16 = Imm16a \|\| Imm16b)<br>  |
| Loadi *(see section 2.2)* | PLX requires 2-bit selection field:<br>(Subop2 = S1 \|\| S2)<br>  <br>PAX requires 3-bit selection field:<br>(Subop3 = S1 \|\| S2 \|\| S3)<br>  |

Table B-1. How Predication of PLX affects some PAX instruction encodings

One method is to make all PAX instructions predicated. This method has the advantage that all instructions in the PAX-PLX combined processor are predicated, and so, the hardware has no need to differentiate between predicated and non-predicated instructions. The disadvantage is that the first three bits of any PAX instruction will have to be used to specify predicate registers. Since these three bits could have been used for something more important, a predicated version of PAX may sacrifice some performance. Table B-1 illustrates these issues. First, the ALU immediate and memory access instructions require a 13-bit immediate field in PLX. However, these instructions in PAX would benefit from a 16-bit immediate field. If the

first 3 bits of the instruction are used to specify predicate registers, then there is not enough room for a 16-bit immediate field. Second, the loadi instruction (See Section 2.2) in PLX requires a 2-bit selection field to specify one of four possible locations, while the PAX versions of loadi requires a 3-bit selection field to specify one of eight possible locations. Once again, there is not enough room for the extra bit of selection because of the space taken up by the 3 bit predicate field.

A second method to make predication compatible between PAX and PLX is to create a new predication mode. Prior to examining an instruction, first check the mode. If the mode is predication-enabled, then the instruction is treated as a PLX instruction. In this case, treat the first 3 bits as a predicate field, and examine the opcode field only if the predicate register is true. If the mode is predication-disabled, then the instruction is treated as a PAX instruction. In this case, bypass the predicate-check and look directly at the opcode. Depending on the opcode, the first three bits can be either an immediate field or a subop field. The advantage of this solution is that many pairs of PAX and PLX instructions can share the same opcodes, although they use the first three bits differently. Also, it is very simple to switch between the PLX and PAX instructions in a single program. The following pseudo-code explains this convenience:

```
pred_on                 // instruction that switch to predication-enabled mode
P0 addi Rd, Rs, 0x16    // addi treated as PLX instruction. Hence, addi
                        // executes only if P0 (predicate register) is true and it has a
                        // 13-bit immediate field.
pred_off                // instruction that switch to predication-disabled mode
addi Rd, Rs, 0x16       // addi treated as PAX instruction. Hence, addi
                        // has a 16 bit immediate field.
```

*addi PLX version*

| P (3) | Op (6) | Reg (5) | Reg (5) | Imm13 |
|---|---|---|---|---|

*addi PAX version. Imm16a and imm16b are concatenated to form Imm16.*

| Imm16b | Op (6) | Reg (5) | Reg (5) | Imm16a |
|---|---|---|---|---|

Another advantage of this solution is that the two processors maintain their original properties: all PLX instructions are predicated, while all PAX instructions are non-predicated. The disadvantage of this solution is that a 32-bit instruction does not include all information needed to execute this instruction. We must look at the predication mode prior to examining any instruction.

| *opcode* | *Bit 1* | *Bit 2* | *Bit 3* | *Bit 4* | *Bit 5* | *Bit 6* |
|---|---|---|---|---|---|---|
| 2C | 1 | 0 | 1 | 1 | 0 | 0 |
| 2D | 1 | 0 | 1 | 1 | 0 | 1 |
| 2E | 1 | 0 | 1 | 1 | 1 | 0 |
| 2F | 1 | 0 | 1 | 1 | 1 | 1 |
| 3C | 1 | 1 | 1 | 1 | 0 | 0 |
| 3D | 1 | 1 | 1 | 1 | 0 | 1 |
| 3E | 1 | 1 | 1 | 1 | 1 | 0 |
| 3F | 1 | 1 | 1 | 1 | 1 | 1 |

Table B-2. 8 non-predicated

The two methods examined so far are, in a sense, polar opposites. Method 1 calls for making all PAX instructions predicated, while method 2 calls for maintaining the non-predicated property of all PAX instructions. The third method is a compromise between the first two methods and is suitable for the PAX-PLX combined instruction set. Some PAX instructions are predicated, and the rest remain non-predicated. Of the 64 possible opcodes, eight opcodes are selected to be non-predicated, Table B-2, and the other opcodes are all predicated. The eight non-predicated opcodes are specially chosen so that the processor only has to examine three of the six opcode bits to differentiate between predicated and non-predicated opcodes. If bits 1, 3, and 4 are all ones, then the opcode is non-predicated, and the first three bits of the instruction are treated as either an immediate field or a subop field. If bits 1, 3, and 4 are not all ones, then the opcode is predicated, and the first three bits of the instruction are treated as a predicate register field.

The eight non-predicated opcodes do not cover all PAX instructions. Many PAX instructions are mapped into the predicated opcodes. The advantage of this method is that with

the eight un-predicated opcodes, we can cover most of the important PAX opcodes. Further, it is much simpler to implement than method 2. We choose this method to map the PAX and PLX instruction set into one opcode set, Section 3.

***Use of the loadi instruction***

PAX has a load immediate instruction, which loads a 16-bit immediate to an aligned 16-bit field of Rd. Since PAX is wordsize scalable up to 128 bits, there are up to 8 possible positions to load the 16-bit immediate field in Rd. Furthermore, PLX has a different version of the load immediate instruction, which can only load a 16-bit immediate to one of four aligned 16-bit fields in the lower 64 bits of Rd. The PAX-PLX instruction set needs to support both types of load immediate instructions. There are two methods. The first method is to have a 3-bit subop, which gives 8 different possibilities, one for each of the positions in a 128-bit Rd. Using this instruction, PAX can load a 16-bit immediate to any of the 8 locations in Rd, and PLX can load to any of the 4 locations in the lower 64 bits of Rd. The other method is to have only a 2-bit subop, which gives 4 different possibilities. In this case, loadi can only load to the lower 64 bits of Rd, which is suitable for PLX. For PAX, to completely load a 128-bit register, one would load the lower 64 bits of two separate registers and then use the "mix 8 byte" instruction to combine the two registers into one. The two methods are illustrated below:

*Method 1: loadi has a 3-bit subop. Loading an entire 128-bit register R1 requires 8 instructions, as shown in assembly language below:*

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| loadi.z.0 R1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| loadi.z.1 R1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| loadi.z.2 R1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| loadi.z.3 R1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| loadi.z.4 R1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| loadi.z.5 R1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| loadi.z.6 R1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| loadi.z.7 R1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

*Method 2: loadi has a 2-bit subop. Loading an entire 128-bit register R3 requires 8 loadi instructions and a mix instruction, as shown in assembly language below:*

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| loadi.z.0 R1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| loadi.z.1 R1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| loadi.z.2 R1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| loadi.z.3 R1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| loadi.z.4 R2 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| loadi.z.5 R2 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| loadi.z.6 R2 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| loadi.z.7 R2 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| mix.8.r R3, R1, R2 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

As illustrated above, these two methods only differ by one instruction. To use method 1, we have to use 8 non-predicated opcodes to encode the loadi instructions with 3-bit subops. Since there are only 8 non-predicated opcodes, method 1 is wasteful. To save opcodes, we use method 2 and share the loadi instruction between PAX and PLX.

### *B.1.3 The Encoding Results*

The solution is published as a technical report [21]. The full version of PAX encoding is described in document [24,25], and the complete encoding and references for the PAX-PLX instruction set is in document [22,23].

## *B.2 Development of PAX Assembler, Linker and Simulator*

This section discusses the development of the PAX-32 toolset, which consists of a simulator, assembler, and linker. The PAX simulator is based on the SimpleScalar simulator, and the PAX assembler and linker are based on the GNU toolset. The development method of the PAX toolset discussed in this section can be extended to develop similar toolsets for other new processor ISAs. The more detailed results are described in the technical report [26]. We used this toolset to write assembly code for one round of the AES-128 encryption algorithm, assemble and link it, and simulate it on the SimpleScalar simulator. Then, we ran a similar program with an ARM toolset. We noticed a 10.84 times speedup in the PAX-32 processor compared to the ARM processor when running the encryption algorithm.

### *B.2.1 Introduction*

After the ISA of PAX has been designed and encoded, the next step is to develop a toolset consisting of a simulator, compiler, assembler, and linker. There are two approaches to creating the toolset. One approach is to construct the toolset from scratch, and the other approach is to port PAX onto an existing toolset. The advantage of the first approach is that it is often easier to write the toolset from scratch rather than to learn the code structure of an existing toolset. Nevertheless, in an effort to make PAX as portable as possible, we chose to build the PAX toolset based on a popular toolset that has an easily portable code structure.

The goals of this research are three-fold. First, we describe the development of the PAX toolset, which is based on the GNU toolset and SimpleScalar Simulator [29,30]. This section discusses the development of the simulator, assembler, and linker, but does not discuss the compiler. Second, although the file names and code structures discussed in this paper are specific

to PAX, the development technique used may be generalized to write a toolset for any processor ISA. Finally, we examine the performance results that are obtained for PAX from using this toolset.

### *B.2.2 Methodology of Building a Toolset for a New Processor ISA*

An ISA toolset allows researchers to study the performance of a processor ISA by using only software. The main framework of the toolset is shown in Fig B-1. Using this toolset,

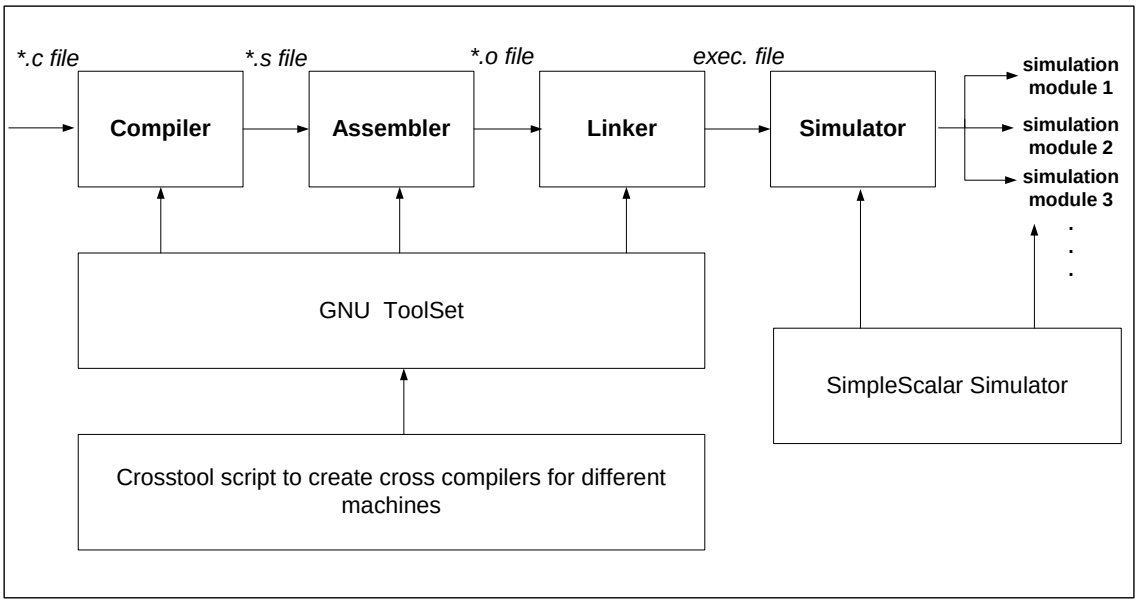



Fig B-1: Structure of toolset for a new processor ISA that is based on GNU toolset and SimpleScalar simulator.

researchers can write c-code or s-code, then produce executable code, and finally run the code on the simulator. There are many variations of simulators, and each one is implemented as a simulation module. Types of simulation modules range from functional simulators, which implement the architecture of the processor, to complex performance simulators that implement the micro-architecture of the processor. By using various types of simulation modules, researchers can study the performance of the processor ISA from many different perspectives.

This way, the strengths and weaknesses of the processor may be carefully analyzed before committing the time and money necessary to design and manufacture the hardware version of the processor. In this paper, we do not cover the development of a compiler for a processor ISA, but this is a necessary part of future research. This paper discusses the development of an ISA toolset that allows researchers to write s-code, assemble it, link it, and simulate it on a functional simulator[1]. The rest of this section discusses the reason for choosing the GNU toolset and the SimpleScalar simulator [29,30] as the base platform, and how to set up the base platform.

***Base Platform of the Toolset***

The reason we chose the GNU toolset as the base platform for the compiler, assembler, and linker is that GNU is a free, open source software[2] that is widely used in both academia and industry. Currently, the GNU Compiler toolset (which includes the compiler, assembler, and linker), called GCC, supports a long list of commonly used machines, including ARM, i386, MIPS, PowerPC, etc. The code structure of GCC is designed so that it can be easily ported to different machines.

Next, the reason we chose the SimpleScalar simulator as a base platform for the simulator is that SimpleScalar is a popular, well-respected simulator used in the academic arena. SimpleScalar was originally written to simulate a sample ISA called PISA, which stands for Portable ISA. PISA is a 64-bit processor that includes a set of commonly used instructions. SimpleScalar is popular for its powerful set of simulation modules, Table B-3. The code structure of SimpleScalar is designed so that researchers who want to use the simulator can conveniently port their processor ISA to SimpleScalar. Currently, SimpleScalar supports a wide

[1] This paper does not emphasize the design of different simulation modules, but instead focuses on the design of the overall structure of a software toolset for a processor ISA.
[2] http://www.gnu.org/

selection of machines ranging from specialized processors designed in universities to popular processors used in industry such as ARM and PowerPC.

| **Simulator** | **Function** |
|---|---|
| Sim-safe | Functional simulator |
| Sim-fast | Functional simulator. Optimized version of Sim-safe |
| Sim-profile | Generates program profiles, by symbol and by address |
| Sim-cache | Generates one- and two-level cache hierarchy statistics and profiles |
| Sim-outorder | Detailed performance simulator |

Table B-3 SimpleScalar Simulator Suite

In order to port a processor to this base platform, one must first pick an existing processor—supported by the base platform—that is most beneficial to use as the starting point. In the case of PAX, that processor is ARM [32]. Then, in both the GNU toolset and the SimpleScalar simulator, we find the ARM related files, create a copy of them, and change them to fit PAX exactly. See Sections B-3 and B-4. Note that each step of the toolset in Fig B-1 can be independently designed. One can pick different processors as the starting points for each stage of the toolset.

One important similarity between ARM and PAX is that they both have 32-bit instructions[3]. This is important because it allows the two processors to share a similar structure in the assembler, linker, and SimpleScalar loader, which is responsible for loading an executable file into the simulator memory. The ARM assembler converts ARM assembly language to ELF-format object files. If we use ARM as a starting point in writing the PAX assembler, then our major task in porting the PAX assembler is to code the PAX instructions, instead of worrying about the structure and format of the object file. On the contrary, if I based PAX on a 64-bit processor, then I would have to change the assembler such that it generates 32 bit instructions in

[3] Note that although PAX is wordsize scalable to 32, 64, and 128 bits, the instruction size is always 32 bits.

the object file rather than 64-bit instruction. This is not a trivial task. Further, if PAX and ARM have similar object file formats, then the PAX linker would be the same as the ARM linker. This is a major benefit of using ARM as a starting point. Similarly, if PAX and ARM share the same linker, then the resulting executable file would be very similar, and this in return means that the ARM SimpleScalar loader and the PAX SimpleScalar loader could be the same.

Moreover, ARM uses the TIS standard ELF file format, which defines the format of the object files. The ELF file format is widely used and has better support in GNU compared to other object file formats such as ECOFF. Since I will have to write a PAX assembler in GNU, it is a good idea to use the well-supported ELF file format.

Now that we have chosen ARM as the starting point processor, the next step is to build the SimpleScalar ARM simulator and the GNU-ARM toolset. SimpleScalar ARM or other SimpleScalar simulators can be downloaded from the SimpleScalar 4.0 website[4]. The readme file included in the download fully describes how to install the simulator.

***Building a Cross-Compiler for Target Processor***

Next, building the GNU-ARM toolset requires the construction of a cross-compiler, which allows one to compile software for a target machine on a host machine of a different type. This is because we are running the GNU-ARM toolset on a linux machine, instead of an actual ARM machine. More importantly, GNU-ARM is only the starting point, and we ultimately need to have a GNU-PAX toolset. Since PAX does not yet exist as hardware, we must use a cross-compiler to run it on a host machine.

Creating a cross-compiler can be a very tricky task. One way to obtain the ARM cross-compiler is to download the version on the SimpleScalar 4.0 website[4]. Currently, this cross-

[4] http://www.simplescalar.com/v4test.html

compiler does not use the newest version of the GNU toolset. Another way is to use the Crosstool script [31] created by Dan Kegel to build the cross-compiler. Users simply specify which machine to target and what version of GNU to use and Crosstool script automatically builds the GNU cross-compiler toolset in a couple of hours.

The results of Crosstool include executables programs for the GCC compiler, assembler, and linker, as well as the source codes from the GNU toolset. We change the ARM-specific files in the GNU assembler source code to port it to PAX. Afterwards, we need to rebuild the GNU assembler. Note that we do not need to rebuild the entire cross-compiler since only the assembler files are changed. Instead of re-running the time-consuming Crosstool script each time that we need to rebuild the assembler, we write a new script that simply rebuilds the assembler in about one minute. We write this script by noting that building a GNU assembler will require the following standard sequence of codes that build the GNU binary utilities:

```
${BINUTILS_DIR}/configure $CANADIAN_BUILD --target=$TARGET --host=$GCC_HOST
-- prefix=$PREFIX --disable-nls   ${BINUTILS_EXTRA_CONFIG}
$BINUTILS_SYSROOT_ARG

make $PARALLELMFLAGS all

make install
```

All of the capitalized parameters above are processor- and system-specific variables that are needed to build the binary utilities. The Crosstool script detects and generates the values for these parameters during run-time. We dump these values to a file and use them for our own script to only build the binary utilities, without running the entire Crosstool script. Now that we have built the GNU-ARM toolset and the SimpleScalar ARM simulator for the base platform, we are ready to port the GNU-ARM toolset to PAX.

### *B.2.3 Building the Assembler*

#### *GNU Assembler File Structure*

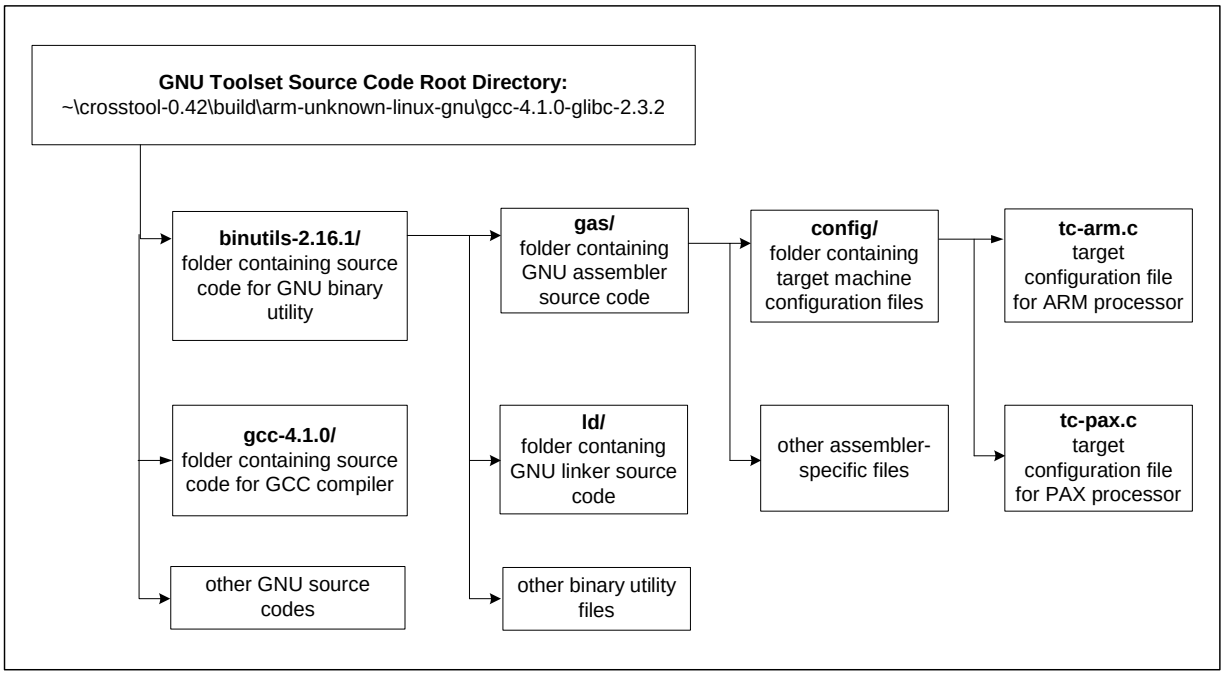



Fig B-2 GNU Toolset File Structure

The Crosstool folder contains the GNU Toolset source codes that were used to build the cross compiler. The file structure of these source codes is show in Fig B-2. The root directory is subdivided into subfolders such as binutils-2.16.1/ and gcc-4.1.0/. The gcc-4.1.0/ folder contains the source code for the GNU Compiler version 4.1.0. The binutils-2.16.1/ folder contains the source code for the GNU Binary Utility version 2.16.1. The Binary Utility consist of the assembler, linker, files that take care of the object file formats, configuration files, and more. The GNU assembler related files are contained in the gas/ folder of binutils-2.16.1/. Further, all the GAS target machine configuration files, which is used to port a target machine to the GNU assembler, is contained within the config/ folder under gas/. To port the GNU-ARM assembler to PAX, we create another copy of the existing tc-arm.c file, which is the ARM configuration files for GAS; change the file name to tc-pax.c; and edit this file so that it fits the PAX design exactly.

***GNU Assembler Code Structure***

Fig B-3 shows the code structure for the GNU assembler. Although the code is specific to PAX, the code structure can be generalized to any processor ISA. Further, we wish to explain the code structure of the GNU assembler with an emphasis on how to port a processor ISA. This is not a complete discussion of the GAS code structure.

The main GAS program is contained in as.c. This program contains a main function, which calls the perform_an_assembly_pass function to carry out the actually assembling process. The assembling process can be roughly subdivided into two parts. One part deals with reading in an assembler file, figuring out the object file format of the target processor, and setting up and configuring the output object file accordingly, such as initializing the various object file sections and taking care of symbol relocation. The other part involves actually translating a line of assembly code such as “addi r8, r8, #0” to a sequence of binary code “0x10210000”. Since PAX and ARM share the same object file format, we do not concern ourselves with the first part of the assembling process.

The perform_an_assembly_pass function calls the md_begin function in tc-pax.c to store the PAX instruction names and the registers into symbol hash tables. The purpose of this will be clear soon. Afterwards, the read_a_source_file function in read.c is called to read in an assembler file and assemble it. Besides configuring the object file format, the read_a_source_file function parses individual lines of the assembler file and sends it as input to the md_assemble function in tc-pax.c, which converts the line of assembler code into binary code. This process is best illustrated with an example. Assume that the md_assemble function takes as input the following PAX instruction:

```
addi r2, r3, #0x08
```

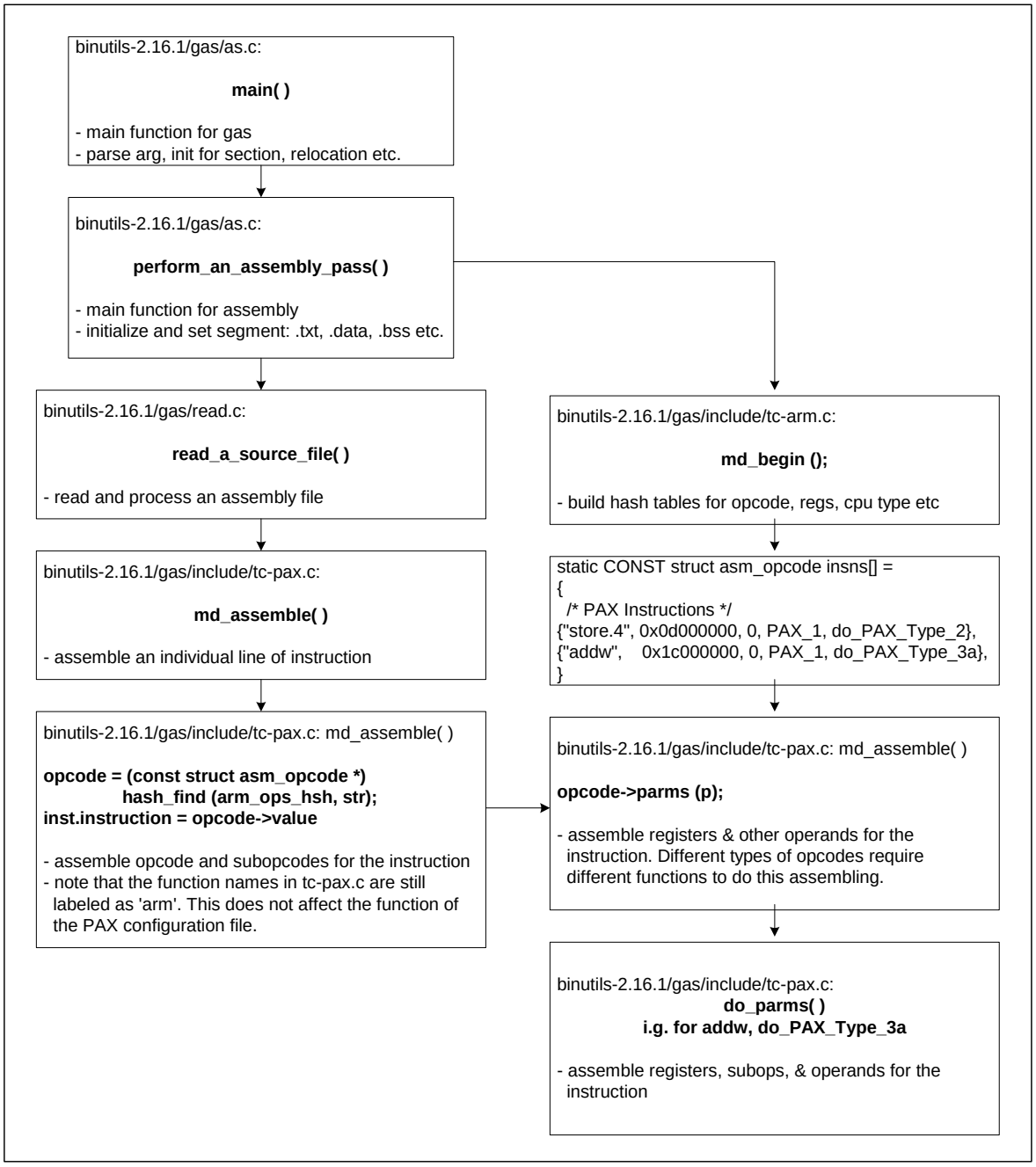



Fig B-3 GNU GAS Code Structure for PAX Processor

This instruction tells the processor to add 8 to the content of r3 and send the result to r2. At this point, the instruction name and register hash table created by the md_begin function becomes useful. The instruction name hash table stores all the PAX instructions with their corresponding

opcodes, subopcodes, instruction types, and more. The md_assemble function searches the "addi" instruction from the hash table to assemble the opcode and subopcode for "addi". Then, given that the "addi" instruction has the instruction type 2, the do_PAX_Type_2 function is called to assemble the operands. The assembling of the register operands r2 and r3 requires the use of the register hash table.

As discussed above, the only part of the GAS source code that we need to change is the part that involves translating individual lines of assembly code into binary code. After studying the code structure of GAS, it seems like we only need change tc-arm.c to tc-pax.c by replacing the ARM-specific configurations with PAX-specific configurations

### *B.2.4 Building the Simulator*

#### *SimpleScalar File Structure*

The root directory of the SimpleScalar simulator is ~/simplesim-pax/[5], as shown in Fig B-4. Directly under this root directory, there is a program called main.c, which is the starting point for the simulator. There is also a separate program for each of the simulation modules that SimpleScalar supports, Table B-3.

Further, there is a sub-directory for each target processor that SimpleScalar supports. These sub-directories contain a standard set of files that should be changed or written to port the target to SimpleScalar. For example, the target-arm/ directory contain the ARM-specific configuration files, and the target-pax/ directory contain the PAX-specific configuration files. The file pax.h is the header file for the target processor that defines the data structure of the

[5] I added the "pax" in the directory name to signify that this is the version of SimpleScalar that is ported to PAX.

processor, including the register structure, the functional units, and different instruction bit fields. These definitions in the header file are used by pax.c and pax.def, as well as the simulator files. The file pax.def contains a list of macro functions and definitions that define the PAX instruction set, the instruction format, and the implementation functions (decoder). The file pax.c contains a set of utility functions that is related to the instructions, registers and disassembler. In addition, the files loader.c loads an executable program into the simulator memory, and the files elf.c and symbol.c take care of the object file format of the target processor.

In order to port PAX to SimpleScalar, we need to use the target-arm/ directory as a starting point for the target-pax/ directory. We modify the arm.h, arm.c, and arm.def files by adding in PAX-specific code to create the pax.h, pax.c, and pax.def files. Since PAX and ARM share the same object file format, we do not need to edit the loader.c, elf.c, and symbol.c files. Finally, we make some minor changes in the simulation module files.

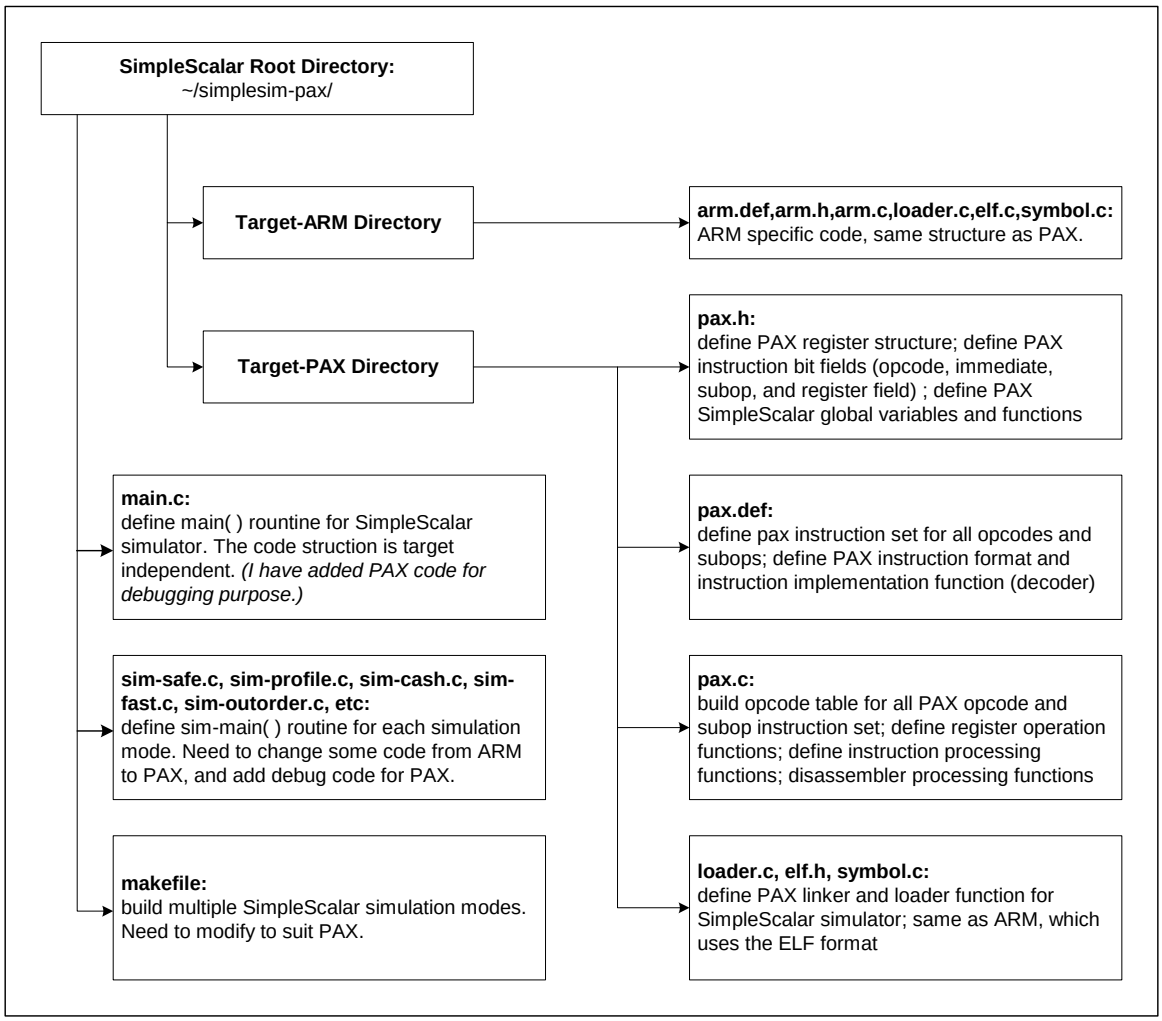



Fig B-4 SimpleScalar File Structure for Porting PAX

### *SimpleScalar Code Structure*

Fig B-5 shows the code structure for the main.c file, which is the starting point of the simulator. First, main.c initializes register statistics, which include a set of variables that record run-time data about the register. Also, each simulation module may require various command line options, and so the main.c file initializes these options. Further, a decode table, which is used in decoding input instructions, is generated using the pax.c, pax.h, and pax.def files. Next, a particular simulation module is initialized. This involves creating the register memory of the

processor. Note that the main.c file is compiled separately for each simulation module. Then, the executable program is loaded into memory. Finally, main.c initializes more simulation statistics, sets the simulator start time, runs the simulation by calling a simulation module, and prints out the log data.

The simulation modules differ in the way they analyze the run-time information, but the code structure is similar. We examine the code structure for the functional simulator sim-safe.c, as shown in Fig B-6. Many of the initialization functions called in main.c actually belong in the simulation module file (main.c calls functions in these files). After initializations are complete, sim-safe.c enters a while loop that fetches an instruction from memory, decodes it, updates the simulator and register statistics, and fetches another instruction. Other more complicated simulation modules analyze the data in more detail, but this while loop is always needed.

The main.c and sim-safe.c code structure presents a good overview of how the SimpleScalar simulator is organized. As we have seen, all the processor-specific information resides in pax.c, pax.h, and pax.def.

### *B.2.5 Extending the Toolset*

In the previous Sections, we have demonstrated how to build a GNU assembler, GNU linker, and SimpleScalar simulator for a new processor ISA. Although we ported the PAX processor by using the ARM processor as the starting point, the methodology can be generalized to build a toolset for other processors. Using the steps described above, we can further extend upon the existing toolset to add new instructions, define new register memory, and create new functional units and instruction flags.

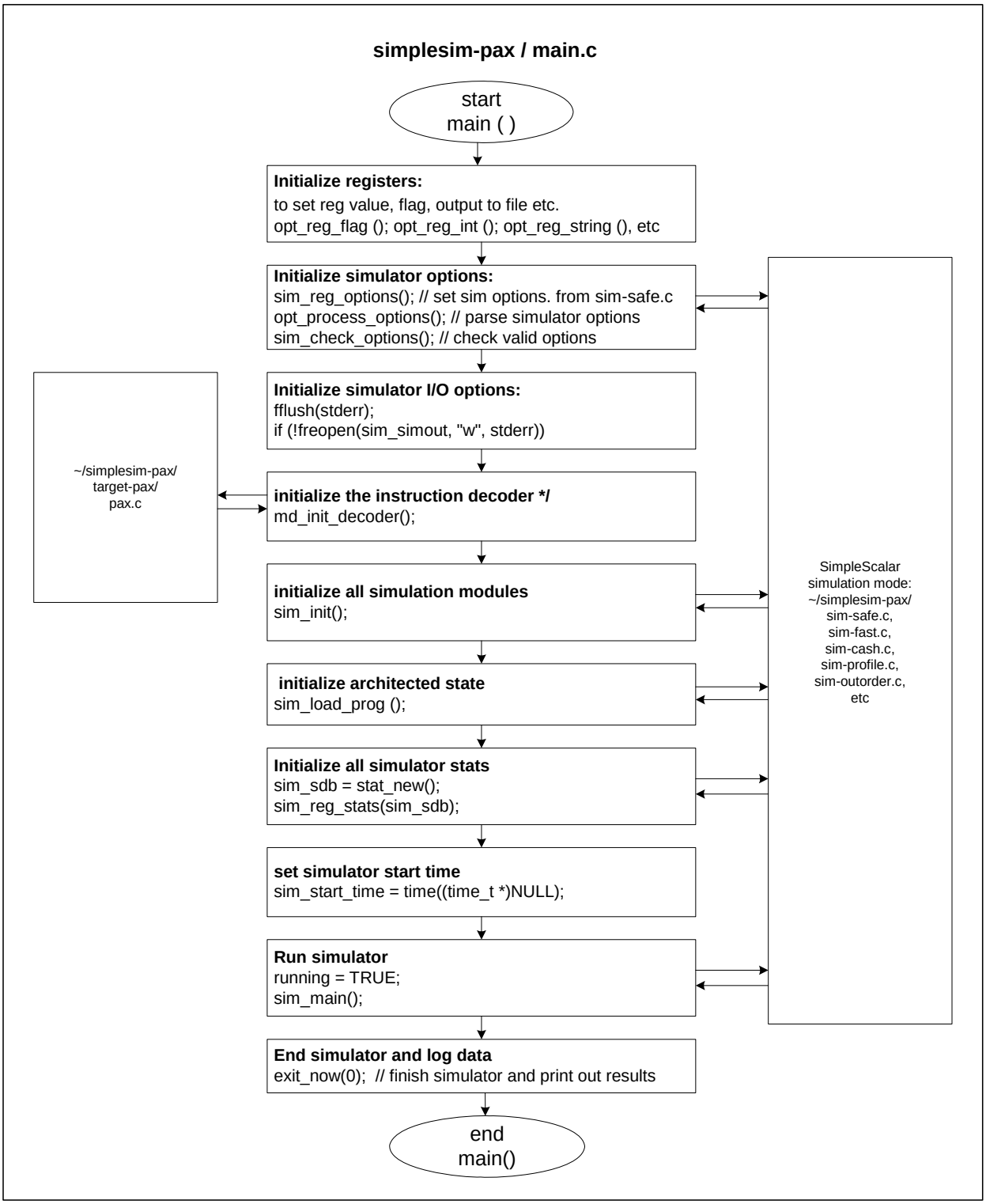



Fig B-5 SimpleScalar Main( ) Code Structure for PAX Porting

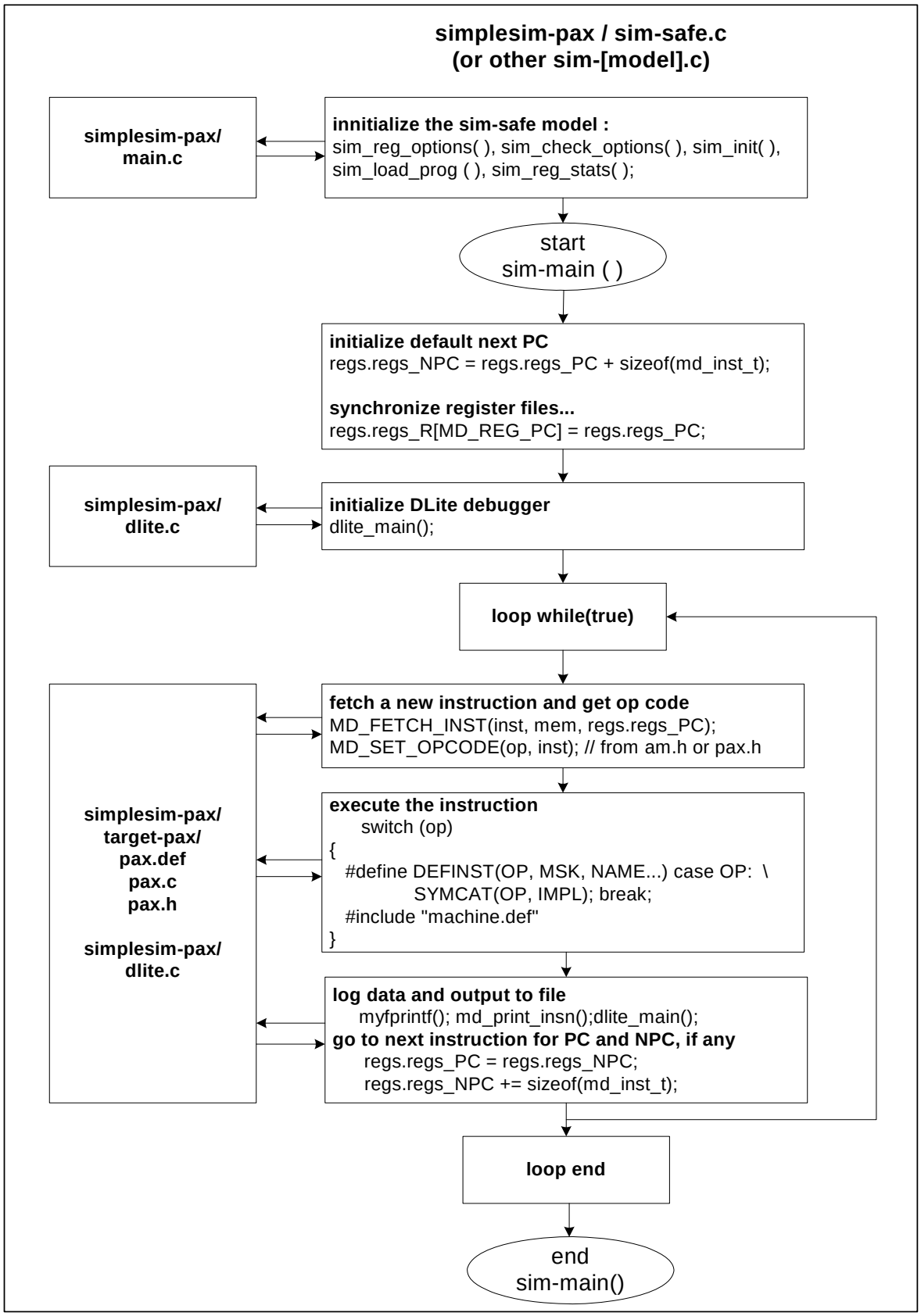



Fig B-6 SimpleScalar Sim-Safe Code Structure for Porting PAX

Moreover, many processors come in different word-sizes, and so, an interesting task is to extend the existing toolset to different word-sizes. For example, the current toolset supports PAX-32. Since this processor is designed as word-size scalable: including PAX-32, PAX-64, and PAX-128, we would eventually like to have toolsets for PAX-64 and PAX-128. The major work necessary to achieve this is to change the data structure definitions from the existing word_t (32-bit integer) to 64-bit or 128-bit integers. Further, all instructions that manipulate these data structures—such as the ALU instructions—must be changed accordingly.

Finally, as we have shown, the SimpleScalar simulator makes it very convenient to add new simulation modules. To thoroughly analyze PAX, we will need to write new modules in the future.

## *B.3 Design of VHDL for the PTLU functional unit for the PAX Processor*

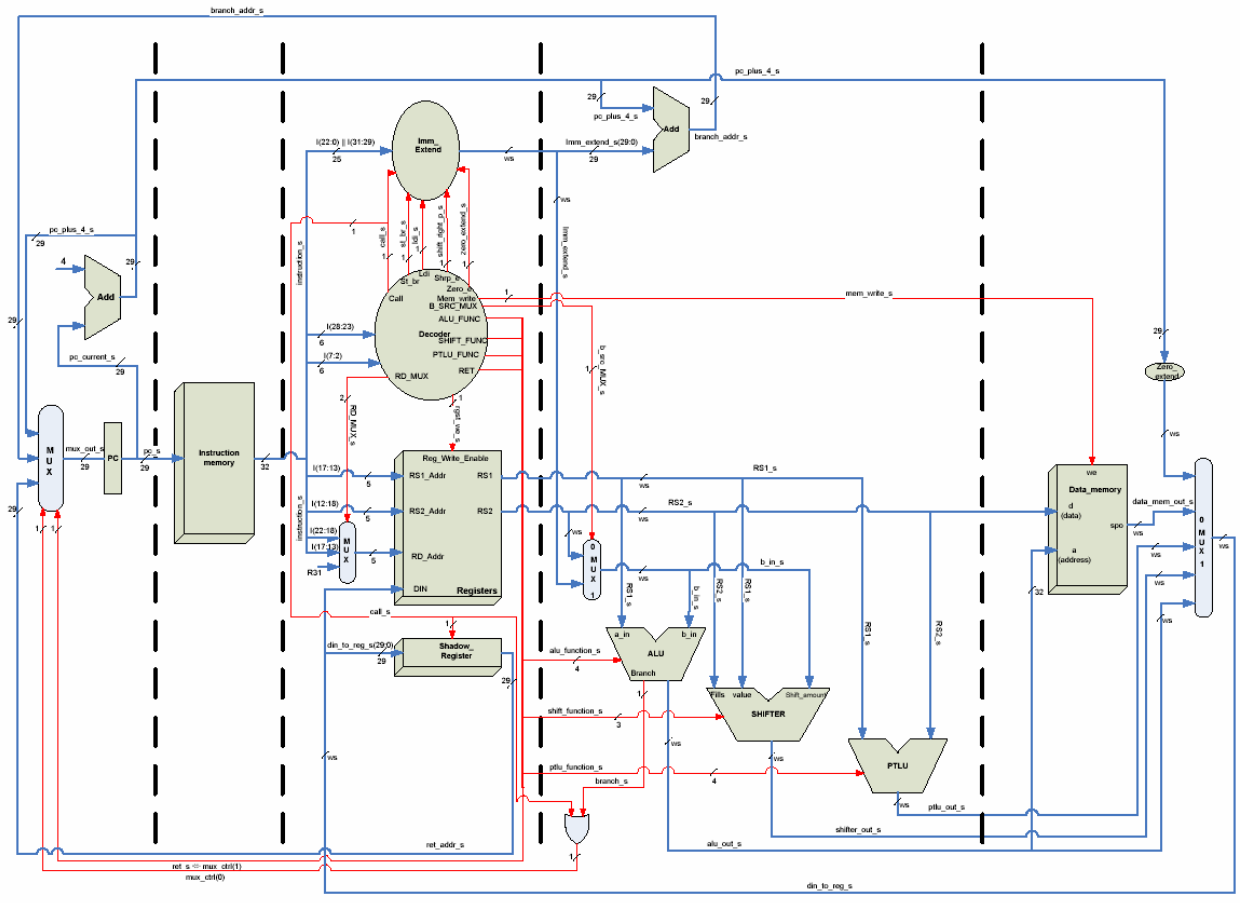



Fig B-7 PAX 5-Stage Pipeline Architectural Block Diagram

Fig B-7 shows the architectural block diagram for a 5-stage pipelined PAX processor. An important feature of PAX is the Parallel Table Look-Up (PTLU) module, which was designed to accelerate the table lookups used in symmetric key ciphers. The PTLU module is scalable for PAX-32, -64, and -128. It consists of w/8 small blocks of memory that can be read in parallel, where w is the wordsize of the processor. A PTLU instruction reads two source registers and writes one result register.

The benefits of the PTLU can be seen by running AES-128 on PAX. Using a 32-bit ARM processor, one frame of AES-128 encryption takes over 800 cycles. PAX can run a frame of AES-128 using much fewer clock cycles as shown in the table B-4.

| **Processor Type** | **Execution Cycles for 1 Frame of AES-128 Encryption** | **Memory Increase** |
|---|---|---|
| ARM-32 | 864 | 0 |
| PAX-32 | 248 | 4 Kbytes |
| PAX-64 | 104 | 8 Kbytes |
| PAX-128 | 21 | 16 Kbytes |

Table B-4 AES-128 Performance Comparison for ARM and PAX Processor

The more memory that the PTLU uses, the fewer cycles it takes to complete one frame of AES-128. However, there is a tradeoff between performance and memory size. As an example, the PTLU module for PAX-64 is shown in Fig B-8..

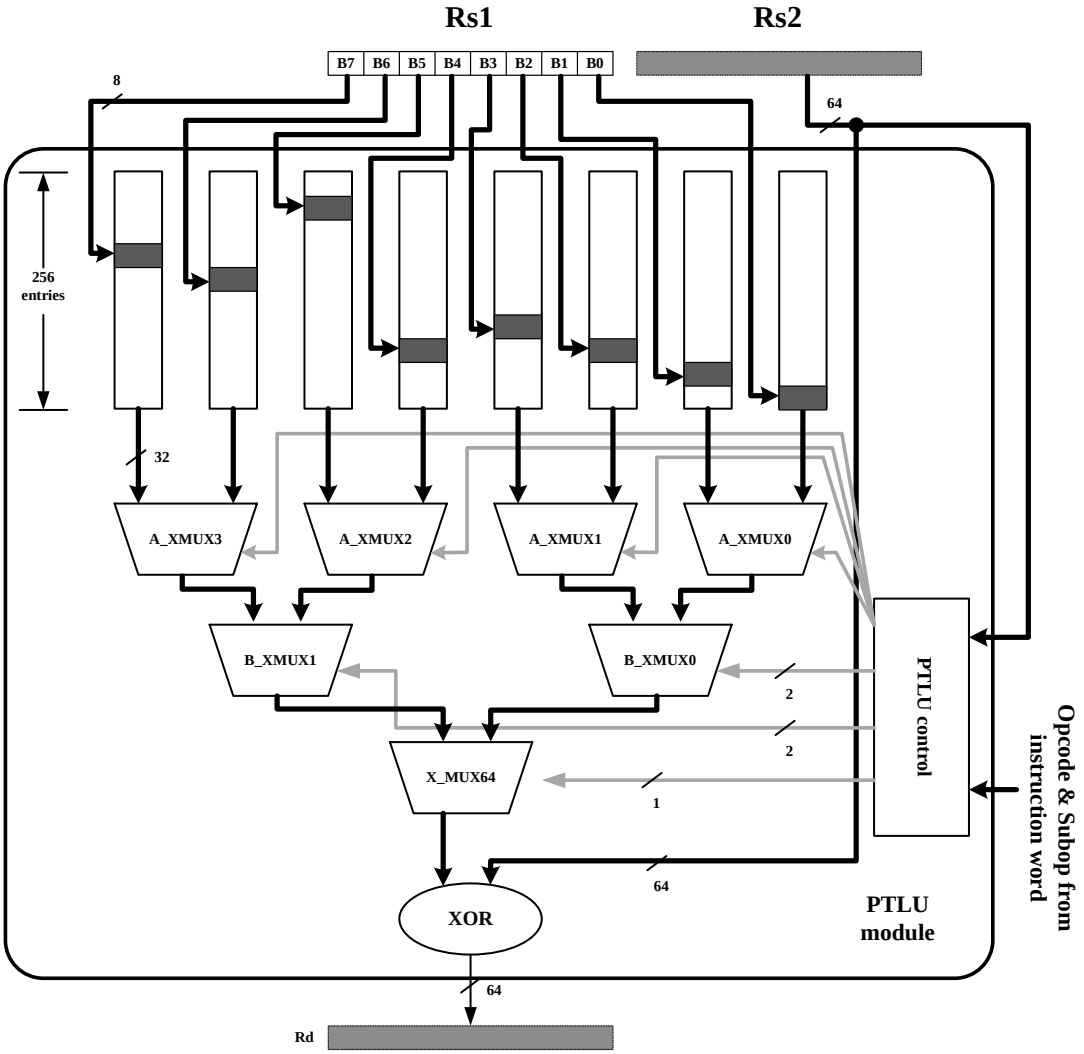



Fig B-8 PAX-64 PTLU Block

**Mapping AES-128 to PAX-32, PAX-64 and PAX-128**

I also performed the detailed mapping of the AES algorithm to PAX-32, PAX-64 and PAX-128. This is non-trivial, since PAX achieves its fast execution time for AES by performing byte permutations of the indices into the parallel tables in the PTLU module, before using the PTLU to do a parallel table lookup. The AES tables are different for AES encoding versus AES decoding. Furthermore, the AES table lookup algorithms are different for PAX-32, PAX-64 and PAX-128, since they have 4, 8 and 16 parallel PTLU tables, respectively, each 32-bits wide.

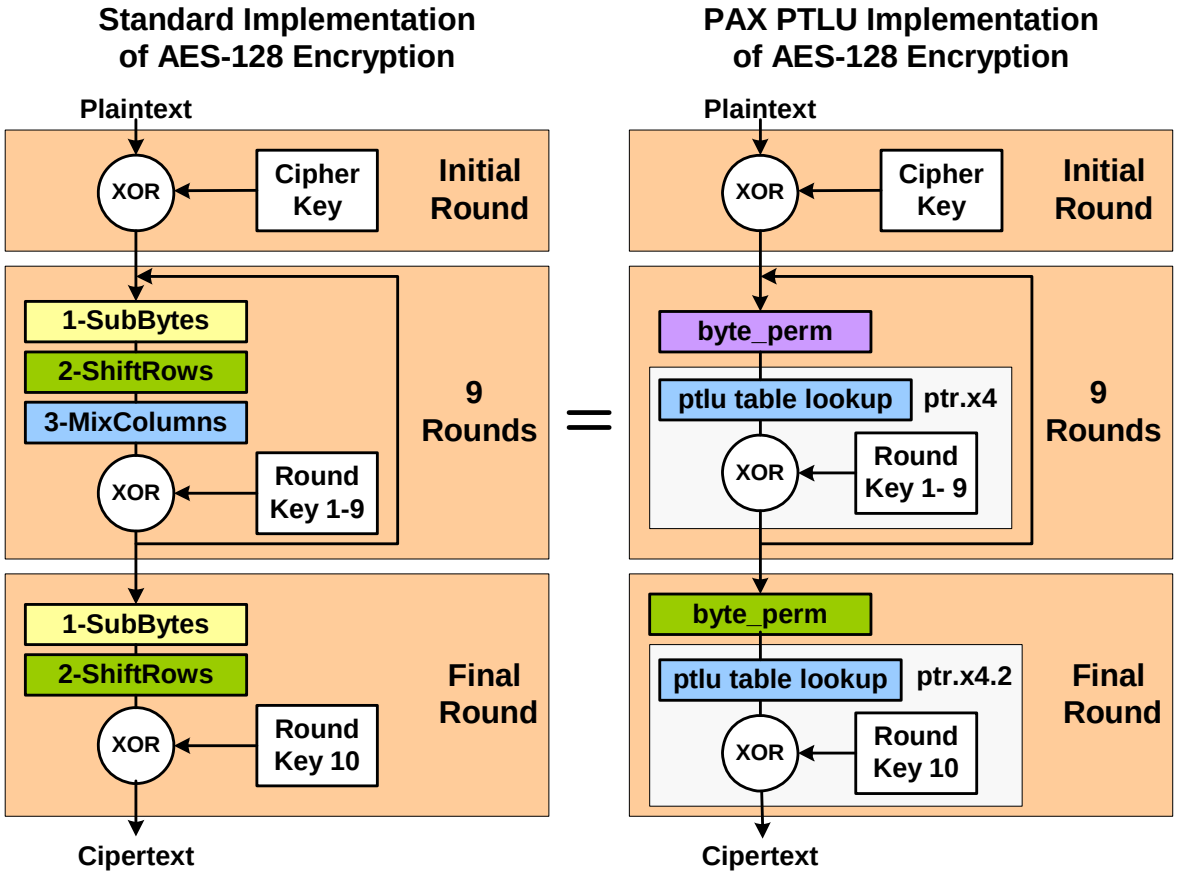



Fig B-9 Mapping AES-128 to PAX PTLU Operations

Performing the mapping required detailed understanding of how the AES cipher works, so as to combine 3 steps per AES round (Substitute Byte, Mix Columns, and Add Round Key) into a table lookup operation with the PTLU read instruction. Then the cyclical Shift Rows step of AES has to be translated into byte permutation operations over the registers, for the next round. This is quite difficult for the different sized registers for PAX-32 and PAX-64. The 6 sets of PAX code for AES encrypt and decrypt for PAX-32, PAX-64 and PAX-128 are available at the PAX web-pages (palms.ee.princeton.edu/PAX**).** Also, Fig B-9 compares how one frame of AES-128 is implemented using its standard operations and how it is done through PAX.

## *B.4 Implementation of PAX FPGA*

After designing the simulator, assembler, linker, and VHDL code for the PAX processor, we implement PAX on a Xilinx Virtex-II Pro FPGA. We note that implementing a full processor on an FPGA is not a trivial task. FPGA in student projects typically implements only a functional unit, not a pipelined processor like PAX. The first milestone is to verify that the PAX VHDL code is synthesizable on the FPGA. To achieve this, we initialize the PAX processor with an AES-128 encryption program onto the instruction memory and a set of input data onto the

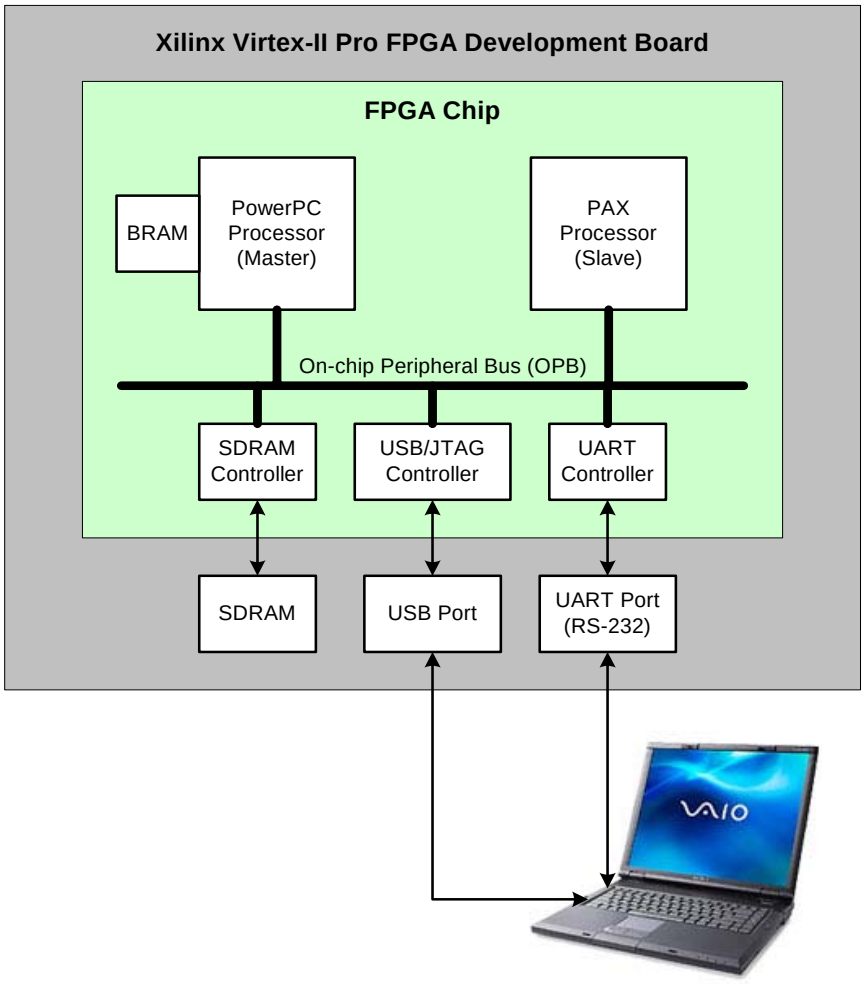



Fig B-10 PAX FPGA High-level Block diagram

data memory. The second milestone is to verify the functionality of PAX by loading and running different programs that is generated by PAX assembler. Fig B-10 shows the high-level hardware architecture of the PAX FPGA. The PowerPC processor is used to read the information on the data memory and send it to the HyperTerminal screen of an attached computer. We use the PowerPC to provide I/O functions and connections to external memory. This saves us from having to first define all these I/O functions for PAX before we can give it input or debug it by seeing information on the display.

### *B.4.1 Hardware Design*

The detailed hardware architecture for connecting the PAX processor to the PowerPC on the FPGA is shown in Fig B-8. We use PowerPC as peripheral controller for PAX processor and user PC. We use the Xilinx provided BUS and control peripherals IP for PAX. We need to write VHDL to interface the Xilinx peripheral IP to PAX that connects PAX to PowerPC, and then we can use PowerPC to write to control PAX operation. The architecture is organized into three major components – the PowerPC and its peripherals, the custom functionality interface, and the user peripheral interface.

The PowerPC component, shown on the lower portion of Fig B-8, is a processor that is integrated into the FPGA chip. There are actually two PowerPC processors on the Virtex-II Pro board, but we only need to use one of them. The PowerPC is connected to a set of peripherals through the OPB bus. One peripheral is the multi-port memory control, which we use to connect to a 512 MB SDRAM. It is also connected to an Ethernet port, which we do not use for the moment. Further, PowerPC is connected to a USB port, which we use to upload the bitstream to synthesize the FPGA. Finally, it is also connected to a UART port, which we use to send results

and debug information to the HyperTerminal screen. Currently, we store all of the PowerPC software on the 32 MB BRAM. This is enough for the moment, but if we run out of space, we can easily expand the software on the 512 MB SDRAM.

The user peripheral interface is shown in the upper right-hand side of Fig B-11. The component is coded in the user_logic.vhd file. This component consists of the custom IP, and this is the file where we instantiate the data memory and connect it to PowerPC. The

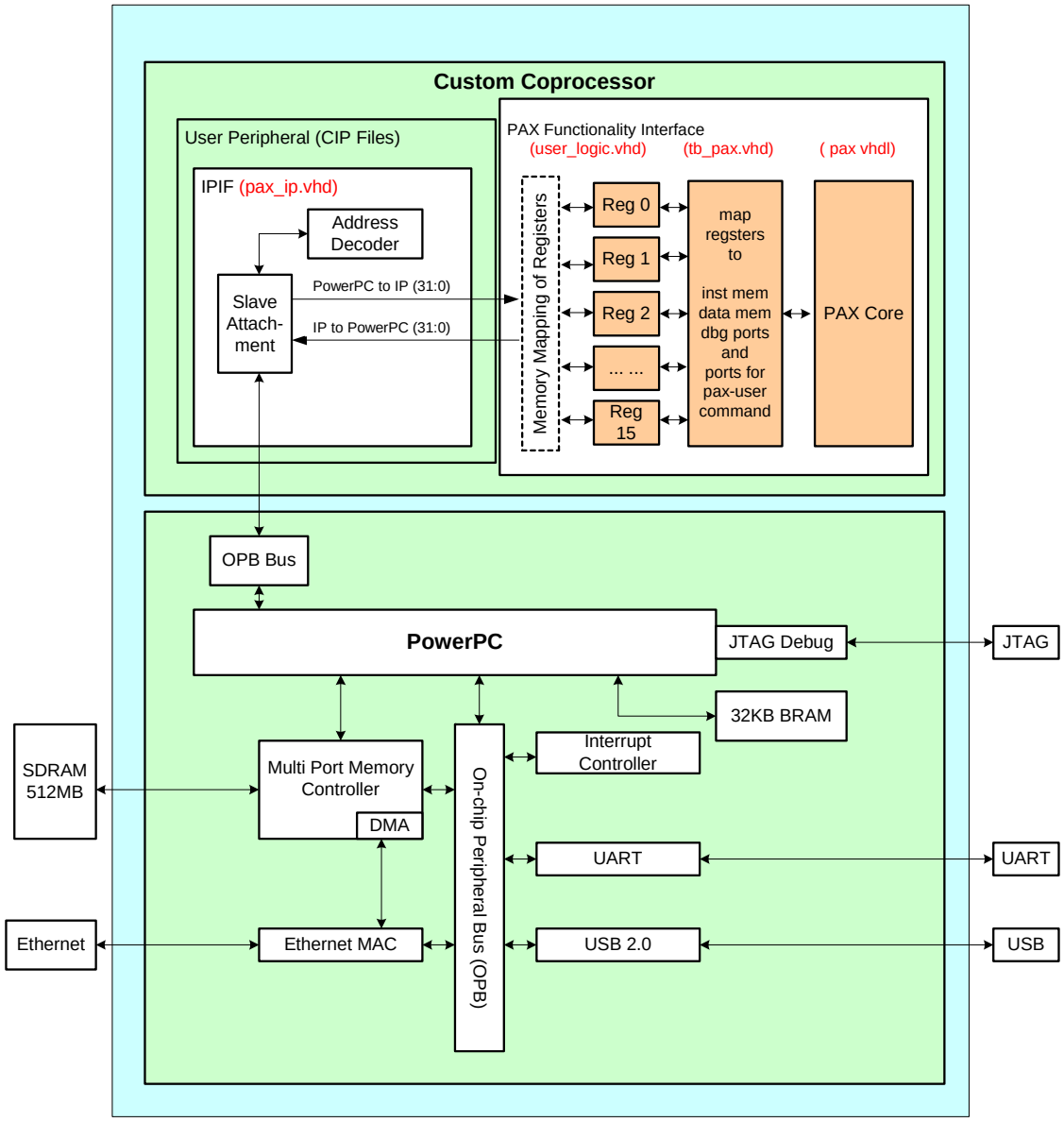



Fig B-11 . FPGA Architecture on Xilinx Virtex-II Pro Development Board

user_logic.vhd file was initially generated by the Xilinx software with a simple, exemplary custom IP that consists of 16 input and 16 output registers. These 16 pairs of registers are connected to PAX instruction memory, PAX data memory and other PAX processor ports for debugging purposes. These 16 pairs of registers are then connected to the PowerPC processor through the user peripheral interface (discussed below). Then, PowerPC can be used to read and write to these registers by simply reading and writing to the specific memory addresses of those registers. We define the communication link between PAX and PowerPC through the following register mapping as show in Table B-5:

| Register | Map to PAX port |
|---|---|
| Register0_in | User sends command to PAX processor. There are 32 bits to define different commands such as:<br>• Start_PAX<br>• Stop_PAX<br>• Get_Current_PC<br>• Get_Current_Instruction<br>• Read_Data_Mem<br>• Write_Data_Mem<br>• Read_Instruction_Mem<br>• Write_Instruction_Mem |
| Register0_out | PAX reports status to user. It also has 32 bits to define a set of PAX status |
| Register1_in<br>Register1_out | Map to PAX instruction memory address port for read and write. |
| Register2_in<br>Register2_out | Map to PAX instruction memory data port for write.<br>Map to PAX instruction memory data port for read. |
| Register3_in<br>Register3_out | Map to PAX data memory address port for read and write. |
| Register4_in<br>Register4_out<br>To<br>Register7_in<br>Register7_out | Map to PAX data memory address port for write.<br>Map to PAX data memory address port for read.<br><br>PAX32 needs one 32-bit register (register4)<br>PAX64 needs two 32-bit registers(register4 and register5)<br>PAX128 needs four 32-bit registers (regsiter4 – register7) |
| Register8_out | Map to PC port to read current instruction address |
| Register9-out | Read out current PAX VHDL code version |
| Others | Reserved for developer for debugging purpose. |

Table B-5 Register mapping table for PAX FPGA

The connections of the PAX ports described above will connect up the PAX processor to the PowerPC processor such that PowerPC would be able to read from and write to the PAX components for controlling the PAX operation.

Although the registers are created in the user_logic file, the actually memory mapping of these registers reside within the user peripheral interface component, shown in the upper left-hand side of Fig B-8. This component is coded in the pax_ip.vhd code. This component connects the custom IP in the user_logic.vhd file to the PowerPC processor. Although there are many ports in this component, two ports in particular are important to us, as shown in the Fig B-8. One port carries a 32-bit data signal from the PowerPC to the custom IP, while the other port carries a 32-bit data signal from the custom IP to the PowerPC. These signals are connected to the correct registers based on the mapping defined in the pax_ip.vhd file.

### *B.4.2 Software Design*

In above FPGA design, we connect the PAX processor to PowerPC through a set of "peripheral interfaces". We also map a set of registers to the PAX processor, through which PowerPC can control PAX. Now we look at C programs that are required to allow PowerPC to communicate with PAX through the register set.

Xilinx FPGA development kit provides an interface file, which I call pax128_ip_selftext.c file. This is the top level C program that instructs how PowerPC should communicate with the data memory component. The functions I developed for PAX are listed bellow:

```
void InitCmd()
void SetCmd(Xuint32 flag);
void ClearCmd(Xuint32 flag);
void GetPaxVhdlVersion();
void GetPAXStatus();
void StartPAX();
void StopPAX();
void WritePaxInstMem(Xuint32 addr, Xuint32 d, Xuint32 disp);
void ReadPaxInstMem(Xuint32 addr);
void WritePaxDataMem(Xuint32 addr, Xuint32 d1, Xuint32 disp);
void ReadPaxDataMem(Xuint32 addr);
void GetCurrentPaxPC();
void GetCurrentPaxInst();
void Delay(Xuint32 num);
void LoadPaxProg();
void DetectPaxProgramDone(Xuint32 timeout);
void ClearPaxDataMemory();
```

These set of utility functions provide a way for a user to load a PAX program, run a PAX program, and debug PAX for further improvement. A picture of the FPGA implementation is shown in Fig B-12. The PAX FPGA development system is shown in Fig B-13. The FPGA test result on Hyper Terminal is shown in Fig B-14.

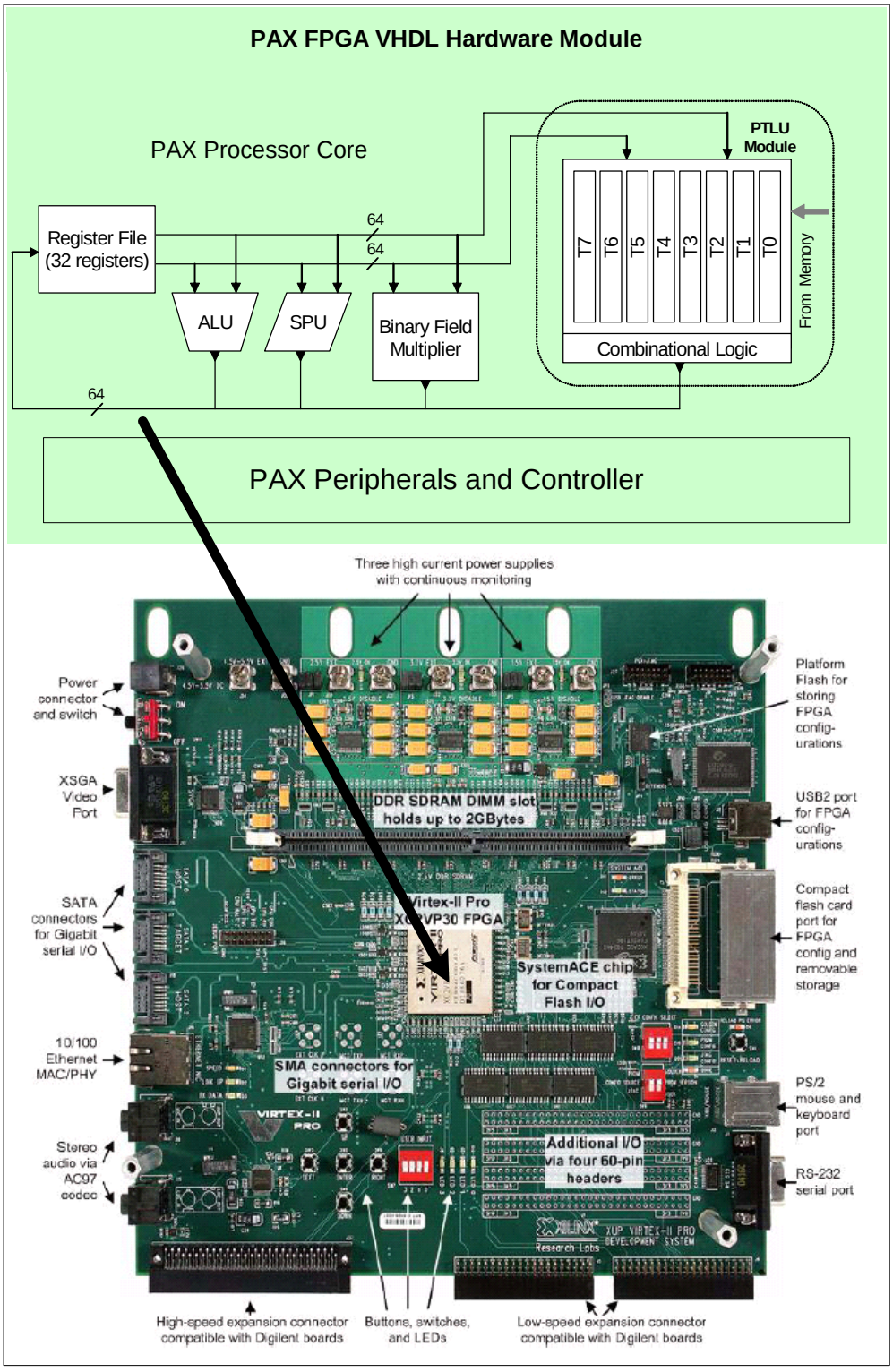



Fig B-12 PAX FPGA Board

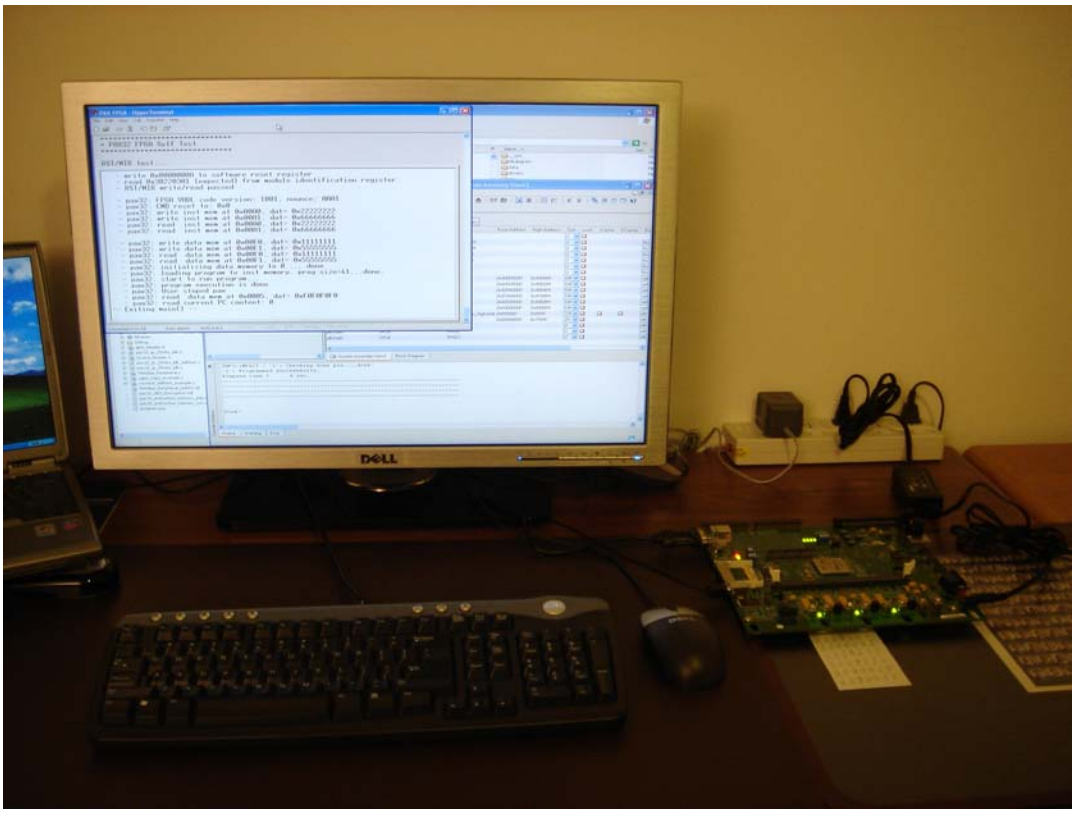



Fig B-13 PAX FPGA Development System

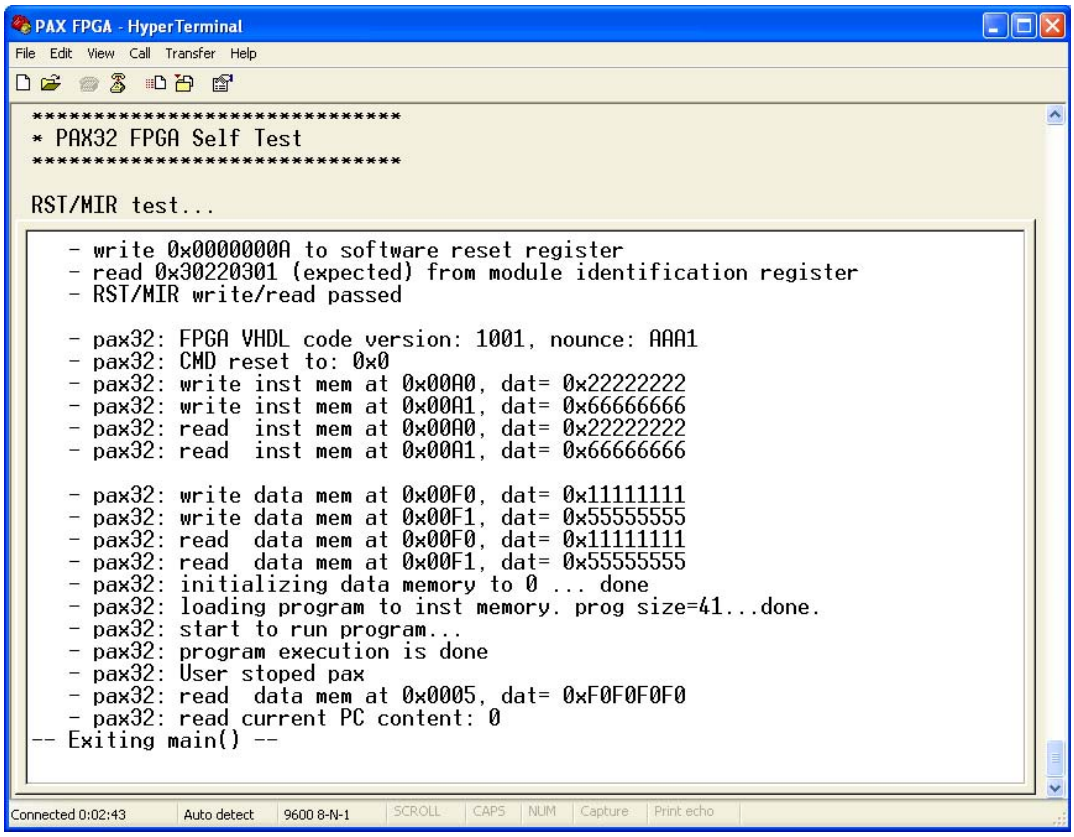



Fig B-14 Hyper Terminal Screen for PAX FPGA Test