Université de Montréal

SCIL Processor
A Common Intermediate Language Processor for Embedded Systems

par

Tongyao Zhou

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures

en vue de l'obtention du grade de Maître ès sciences (M. Sc.) en Informatique

Mai, 2008

Université de Montréal

Faculté des études supérieures

Ce Mémoire intitulé:

SCIL Processor
A Common Intermediate Language Processor for Embedded Systems

présenté par :

Tongyao Zhou

a été évaluée par un jury composé des personnes suivantes :

Abstract

Embedded systems and their applications are becoming ubiquitous and transparent. Nowadays, the designers need to implement both hardware and software as fast as they can to face the competition. Hence tools and IPs became an important factor of the equation. In this work, we present a synthesisable softcore processor similar to the micro-architecture of Tanenbaum's IJVM processor. The processor implements a subset of Microsoft's Common Intermediate Language. We seek to accelerate the development of the embedded software by providing a platform onto which the whole .NET Framework (C#, Visual Basic.NET...) (along with its object-oriented approach) could execute. We used a Xilinx Virtex II PRO as the prototyping platform.

Keywords: Embedded processor, Softcore, CIL, SCIL Processor, Embedded System, .Net language

Résumé

Les Systèmes embarqués et leurs applications sont omniprésents et transparents actuellement. Afin d'affronter des compétitions, des designers ont besoin d'implémenter des matériels et des logiciels le plus vite possible. Des outils et des IPs donc deviennent un facteur important. Dans ce projet, nous présentons un processeur softcore dont l'architecture est inspirée par l'architecture de l'IJVM processeur de Tanenbaum. Le processeur est synthétisable et implémente un sous ensemble de CIL (Microsoft's Common Intermediate Language). Parce que CIL est le plus bas niveau langage dans Microsoft .Net Framework, toutes les .Net langages, comme C# et Visual Basic.NET, peuvent être utilisés pour les systèmes embarqués. Nous souhaitons que cette nouvelle plate-forme puisse accélérer le développement des applications logicielles embarquées.

Mots de clés : Processeur embarqué, Softcore, CIL, SCIL processeur, Système embarqué, .Net langage

Index

List of Tables

List of Figures

Table of Acronyms

ALU	Arithmetic Logic Unit
BRAM	Block Random Access Memory
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLR	Common Language Runtime
CP	Conventional Processor
CPI	Cycles Per Instruction
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
GPIO	General Purpose Input/Output
HDL	Hardware Description Language
ICF	Integrated Coupling Facility
IFL	Integrated Facility for Linux
IFU	Instruction Fetch Unit
IP	Intellectual Property
ISA	Instruction Set Architecture
JVM	Java Virtual Machine
LMB	Local Memory Bus
MIU	MicroInstruction Unit
OEM	Original Equipment Manufacturer
PCI	Peripheral Component Interconnect
PE	Portable Executable
PLD	Programmable Logic Device
RISC	Reduced Instruction Set Computer
SCIL	Simple CIL
SoC	System on Chip
VHDL	VHSIC Hardware Description Language
VLSI	Very Large Scale Integration

Remerciements

Je tiens à remercier mon directeur El Mostapha Aboulhamid pour sa direction et son support tout au long de ma maîtrise. Je tiens aussi à remercier Luc Charest qui a passé beaucoup de temps à me montrer comment rédiger un mémoire, j'en avais bien besoin.

Je dois aussi remercier mes parents et ma copine pour leurs encouragements et leur appui dans les moments difficiles.

Chapter 1 Introduction

Introduction

With embedded systems used more and more widely, new design methods and new hardware development tools are introduced and commercialized. However, embedded system designers continue to demand complete solutions to build and complete quickly their hardware and software designs. To satisfy such demands, many manufacturers provide their embedded processors and corresponding integrated embedded development environments, such as Xilinx's MicroBlaze [20] and Xilinx Platform Studio [33], as well as Altera's NoisII [11] and QuartusII Development Software [34]. By using these design tools, the embedded system designers can develop a SoC (System On Chip) starting at a relatively high level. On the hardware side, the designers choose the embedded processor and construct the embedded sub-system implementations under the development environments; on the software side, the designers develop software applications and then convert them to embedded processor instructions, which can be executed by the embedded system implementations. After that, the designers use the functionalities integrated in the development tools to modify the optimal design features, improve the design performance, and optimize area and cost of the design system. In this way, the developers can craft embedded systems quickly and easily.

In this work, we introduce a new softcore processor, *SCIL Processor*, which implements a subset of Microsoft's Common Intermediate Language (CIL) [2]. This processor makes it possible to use all primary .NET language in embedded system designs to develop software applications. In our design, because it is hard to directly implement the CIL on hardware, we consider a subset of the CIL as a simpler intermediate language, and then implement this new language on hardware.

Embedded processors

An embedded system is a special purpose computer system designed to perform one or a few dedicated functions, and it is usually embedded as part of a complete device including hardware and mechanical parts [3]. In order to shorten the period of embedded system development, almost all designers use the CPU platform. The CPU platform uses the special-purpose embedded processors, which can be purchased as part of the chip design, to construct the embedded system. By using the CPU platform, it is easy and quick for the designers to develop a chip (SoC) and create the complex embedded systems. A SoC consists of the hardware and the software. The hardware includes embedded processor, DSP (Digital Signal Processor) cores, peripherals and interfaces; and the software which is the program loaded into the memory controls operations of the hardware. The design flow for a SoC aims to develop hardware and software in parallel. The SoC designs can program on field-programmable gate array (FPGA) with all the logic, including the embedded processors.

There are two kinds of embedded processors: microprocessors (μp) and microcontrollers (μc). Microprocessor are the single VLSI chip that has a CPU and may also have some other units such as caches, floating point processing arithmetic unit, and super-scaling units. Microprocessors support their particular instruction sets. Microcontrollers are the single-chip VLSI unit, which has built-in peripherals together with some microprocessors on the chip. The use of microcontrollers can reduce the size of embedded systems because it reduces the size of control programs. Since the first microprocessor Intel 4004 [4], which requires external memory and support chips, was used in embedded systems, many microprocessors have been developed and commercialized in this field. Furthermore, in contrast to the personal computer (PC) market where only limited CPU architectures are used, there are many different CPU architectures used for embedded designs such as ARM [5], MIPS [6], Atmel AVR [7], Zilog Z80 and Z8 [8], Renesas H8 and M32R [9], PIC [10], as well as PowerPC.

Embedded processors can also be divided into hardcore processors and softcore processors. A hardcore processor is a fabricated integrated circuit that may or may not be embedded into additional logic, and usually it has a fixed unchangeable construction. A softcore processor is a microprocessor core described in a HDL, and that can be implemented using logic synthesis. It can be implemented via different semiconductor devices containing programmable logic such as FPGA. The softcore processor can be configured based on factors such as schedule, unit cost, space constraints, product lifetime, toolset, and flexibility needs. Although usually the hardcore processors can achieve better performance than that the softcore processors, the softcore processors are widely used because not all embedded applications need the high speed performance. In practice, many applications require expanded functionality and flexibility. Softcore processors usually provide a substantial amount of flexibility through the configurable nature of FPGA. The flexibility allows embedded system designers to create a custom system that contains only the needed functionalities. Furthermore, it is easy for the softcore processor systems to modify the current designs to meet future needs. Therefore, softcore processors may be used not only in a simple system, where the only functionality is limited to a simple GPIO (General Purpose Input/Output), they may also fit a complex system, where an operation system is incorporated and includes many peripherals or any other custom IP. Moreover, these softcore processors can be implemented in a much shorter amount of time than hardcore processors can. Therefore softcore processors can shorten time-to-market. At present, the most popular used softcore processors are Xilinx's MicroBlaze and Altera's NiosII.

The MicroBlaze [20] is a softcore processor optimized for Xilinx FPGAs. The MicroBlaze is based on RISC architecture. It features a 3-stage or 5-stage pipeline, with an instruction completing in each cycle. Both instruction and data words are 32 bits. The MicroBlaze can reach speeds of up to 210 MHz on the virtex-5 FPGA family. The processor can communicate via the LMB bus for a fast access to local memory, which is normally the BRAM inside FPGAs. The size of the BRAM is flexible and can change based on the demands of target systems. With the configurable definition, the MicroBlaze can be customized to the applications in many aspects such as: cache structure, peripherals, as well as interfaces. In addition, the MicroBlaze can add or remove hardware implementation for certain operations including multiplication, division, and floating-point arithmetic. In Figure 1, we present the base architecture of the MicroBlaze, and show its 3-stage pipeline.

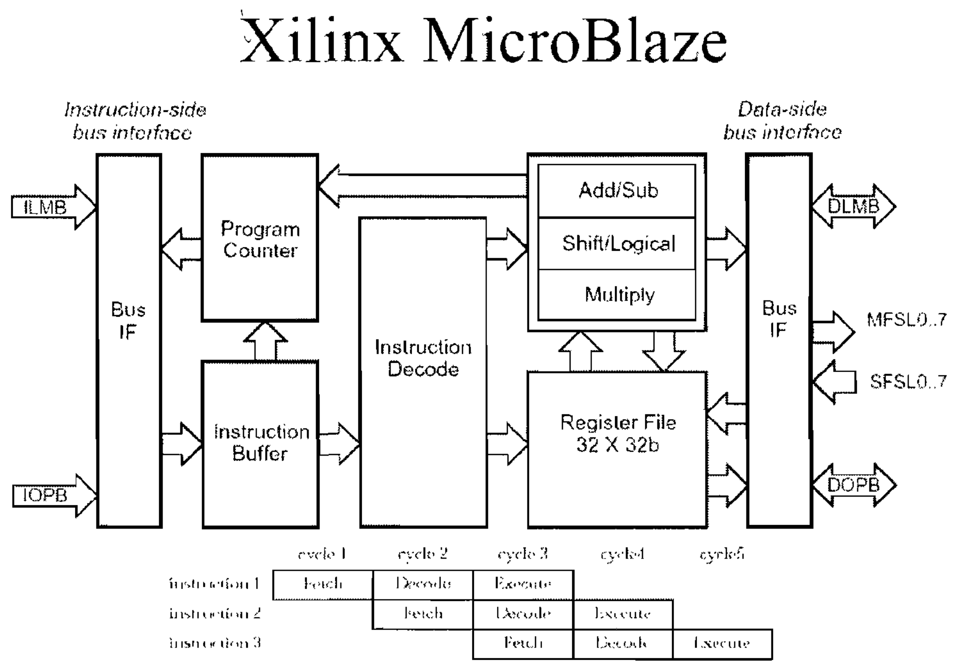


Figure 1: Xilinx MicroBlaze [20]

Motivation of project

Nowadays Xilinx Inc and Altera Corp dominate the whole PLD (Programmable Logic Device) market. Based on a business report [38], Xilinx and Altera accounted for a combined 83.4 percent market share of the PLD market in 2005. Xilinx is the PLD market leader with a 50.3 percent market share and second-place Altera captures 33.1 percent of the PLD market. The other small corporations are so far down the two giants so that almost nobody ever hears of them. As a result when people develop embedded systems, they naturally choose the product from Xilinx and Altera. Because both PowerPC and MicroBlaze from Xilinx and NiosII from Altera focus on C/C++ programs, currently C and C++ are the main programming languages which are used in embedded system designs. On the other hand, a number of researches have been to develop hardware implementations for Java. In fact, there exist many Java processors, which can support JVM or dedicated Java instructions. Hence, it is possible to use Java as the embedded system developing language with these Java processors.

However, there are few attempts to create .NET language processors and use the .NET languages in embedded system designs although .NET languages are used widely at present. Therefore, we try to develop an embedded processor for .NET language and aim at the language CIL. Because the CIL is the lowest-level language in the .NET Framework and all primary .NET languages, including C#, Visual Basic .NET, C++/CLI and J#, can compile to the language CIL, the new processor can execute all .NET programs. In this way, we can use all .NET languages as programming languages in embedded system designs with our processor.

With such an embedded processor for .NET language, the designers can use the existing programs, which are written in .NET languages, for the target applications instead of translating them to the programs in C or Java. In addition, as an IP or a co-processor, the .NET processor would be used as a dedicated unit, which is responsible for executing .NET programs, in one system. Finally, we can use this new processor to do some tests and benchmarks in multiprocessor systems. It is interesting to compare the execution results of different languages.

Introduction of Microinstruction

"Microinstruction is an instruction that controls data flow and instruction-execution sequencing in a processor at a more fundamental level than machine instructions. A series of microinstructions is necessary to perform an individual machine instruction." [21] Microinstructions help the designers to find a simple and easy method to develop the control logic for a processor. Originally, people implemented machine instructions directly in circuitry which provided fast performance. However as instruction sets became more and more complex, the corresponding circuitries became more difficult to design and needed too many hard resources. In 1951 Maurice Wilkes described using microinstructions in CPU design for the first time. By using Microinstructions, CPU design engineers can write a microprogram to implement a machine instruction rather than design a circuitry for it. It is more flexible to use microinstructions than use circuitries. Even late in the design process, designers can easily modify the context of microinstructions to adapt the changeable CPU demands. Moreover, it is possible to realize very complex instruction sets with microinstructions. The CPU designers can use microinstructions to implement many abstract and high level machine instructions. The famous CPU that uses microinstructions is the IBM System360.

The microprograms with microinstructions exist on a lower conceptual level than other familiar programs. As one single high level language statement is compiled to a series of machine instructions, one machine instruction is implemented by a series of microinstructions in the processor using microinstructions. The microinstruction exists usually in a special read-only memory instead of the main system memory. Microinstructions control the action of the processor at a very low level. For example, a single typical microinstruction might specify which register should be updated or which operation of ALU should be done in one single cycle. Microinstructions can be thought as the combination of command signals for all parts of the processor. In Example 1, we show several standard microinstructions. There are four microinstructions from M0 to M3. These four microinstructions are to implement the instruction *fetch*. The processor has to execute the four microinstructions in turn for each instruction *fetch*.

```
M0:   PC_out, MAR_in
M1:   read, pcincr
M2:   MDR_out, IR_in
M3:   decoding opcode in IR
```

Example 1: The sequence of microinstructions for the instruction fetch

Introduction of Common Intermediate Language

CIL (Common Intermediate Language) is the lowest-level human-readable programming language in the CLI (Common Language Infrastructure) of Microsoft's .NET Framework. All primary .NET languages, including C#, Visual Basic .NET, C++ and J#, are compiled to the CIL before .NET program execution. The CIL is a CPU-independent and platform-independent instruction set, and it can be executed in any environment supporting the .NET framework. Like JVM (Java Virtual Machine), the CIL has a stack-based architecture and uses bytecode instructions. Moreover, the CIL is an object-oriented language.

During execution of a .NET assembly, its CIL codes are passed through the CLI's JIT (Just-In-Time) compiler. The JIT compiler translates bytecode instructions to native codes that are immediately executable to the CPU. The procedure of compilation is performed gradually during the whole program's execution. Moreover, in a CIL program, except for CIL instructions, there are many Metadata. A .NET language compiler generates Metadata and assembles them with CIL instructions. A Metadata in CIL file begin with a "point". For example,

```
.maxstack 2
```

Metadata contain the information about compiled classes and some additional attributes. Metadata can be thought as the complementary descriptions for CIL instructions. For example, Metadata used for a method usually contain the information about the class name, the type of the return value and the type of the method parameter. The information ensures that the method can be invoked. The JIT compiler reads these Metadata during the JIT compilation. In Figure 2, we show the basic process of CLR (Common Language Runtime).

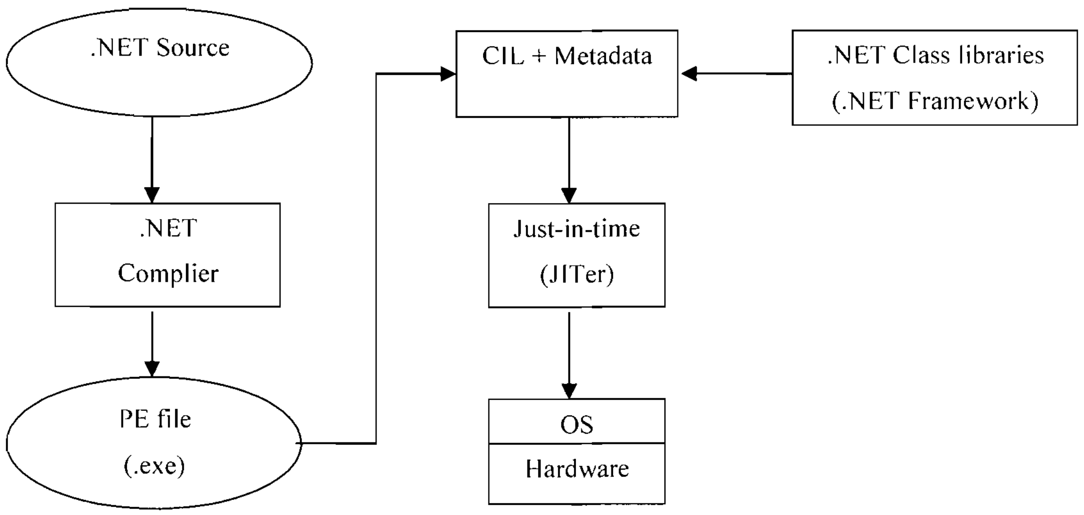


Figure 2: Common Language Runtime

The .NET compiler firstly translates a .NET programs to a PE (portable executable) file [23]. The PE file is a collection of CIL instructions and Metadata. When the PE file is executed, a JIT compiler compiles CIL instructions and Metadata to native language instructions. During the compilation, the JIT compiler refers to .NET class libraries. Finally, these new instructions can be executed on some special hardware environment with some special OS.

Introduction of SCIL processor

The SCIL processor is a synthesizable softcore processor which implements a subset of the CIL. It is a little-endian processor. The processor supports 32 bits integer calculation, and it cannot execute floating-point operations. The processor does not support object-oriented concepts at present.

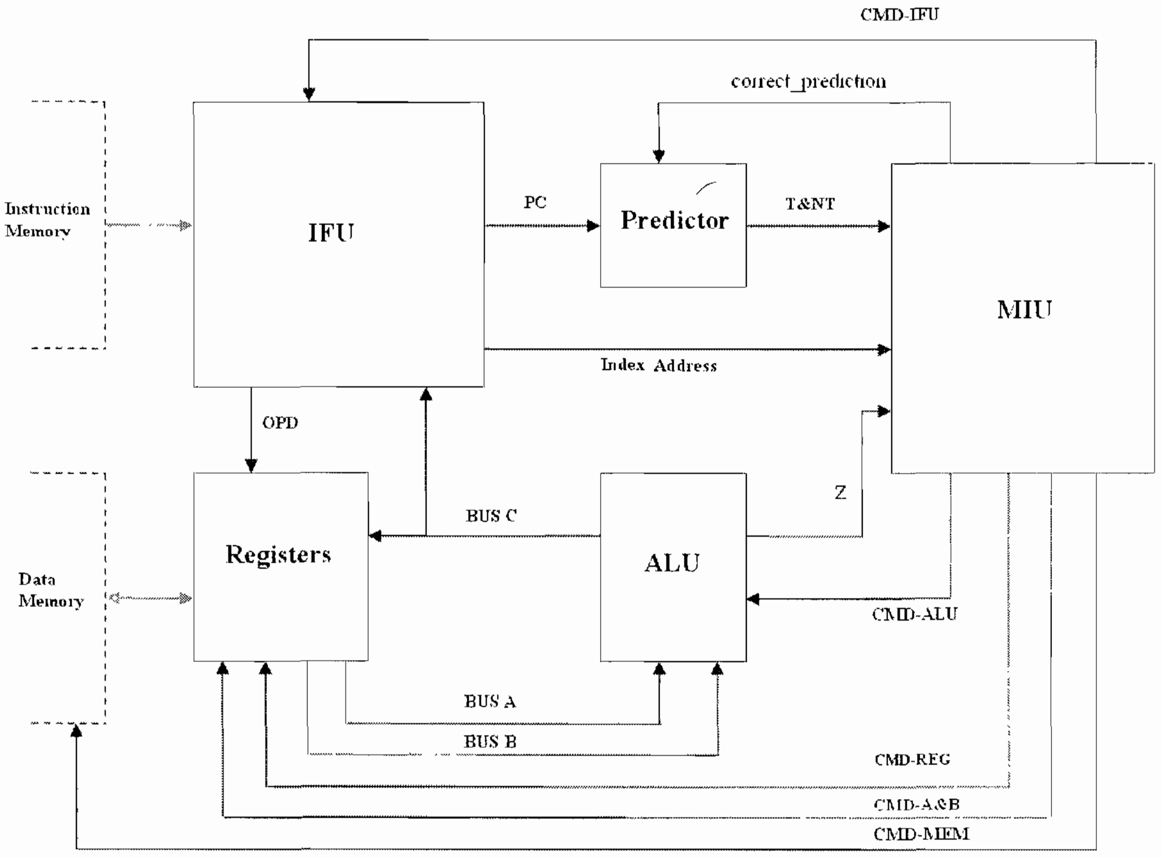


Figure 3: Block-diagram of SCIL processor

In Figure 3, we present the block-diagram of the SCIL processor. The processor consists of the following functional units: Instruction Fetch Unit (IFU), Microinstruction Unit (MIU), ALU, Predictor and Registers (local memory). The IFU fetches SCIL instruction data from memory and then decodes the SCIL instruction code; the MIU picks out microinstructions and converts them to command signals; the ALU does the basic arithmetic operations and determines whether the processor takes condition branches; the Predictor is a one-bit predictor with 128 different addresses for branch prediction; and the local memory consists of nine 32-bit registers. Furthermore, we can find three data buses (BUS A, BUS B and BUS C) and five command signal sets (CMD_A and B, CMD_ALU, CMD_REG, CMD_MEM, and CMD_IFU) which control the operations of different units. In addition, the SCIL processor directly connects two BRAMs. One is as the instruction memory and the other is as the data memory.

Outline of thesis

The remainder of the thesis is organized as follows. We will introduce the related work in next chapter. In chapter 3, we will discuss the CIL and the new language SCIL. Then in chapter 4, we will present the detailed SCIL processor architecture. We will explain the functions and characteristics of each unit of the processor. After that in chapter 5, we will show some experiment results, and we will compare and discuss the performance between the SCIL processor and the MicroBlaze processor. Finally, we will present the conclusion and future works in chapter 6.

Chapter 2 Related work

Currently most embedded processors, which target a specific programming language, focus on Java or JVM. These processors are usually called Java processors. In this chapter, we present several typical Java processors. We introduce three JVM based processors: SUN's picoJava processor, Co-Designed Java Virtual Machine processor developed by University of New Bunswick and Tanenbaum's IJVM processor. Moreover, we introduce the Lightfoot processor, which supports the instruction set interpreted from JVM. Furthermore, we introduce IBM System z Application Assist Processor (zAAP), which is unique Java processor used in large-scale commercial field.

picoJava processor

SUN's picoJava processor [12] may be the most famous Java processor although picoJava only appears in research papers and this processor is never released as a product by SUN. Now SUN provides the full Verilog code under an open source license [13]. The first version picoJava core, picoJava-I, was introduced in 1997.

Through an interpreter or through just-in-time (JIT) compilation, Java programs can be executed on a processor. However, both the interpretation and the JIT compilation have their disadvantages. The nature of interpretation involves a time-consuming loop, which affects performance significantly. A JIT compiler can reach a high speed. However, because the compiler itself and compilation require large quantity of storage, it consumes much more memory, which is a precious resource in the embedded designs, than the interpretation. Therefore, SUN developed picoJava-I processor to create a processor to the Java environment which can eliminate the disadvantages of the two traditional execution ways. The picoJava-I is a small, configurable core designed to support the JVM. In Figure 4, we present the major function units of the picoJava-I. The shading parts indicate configurability.

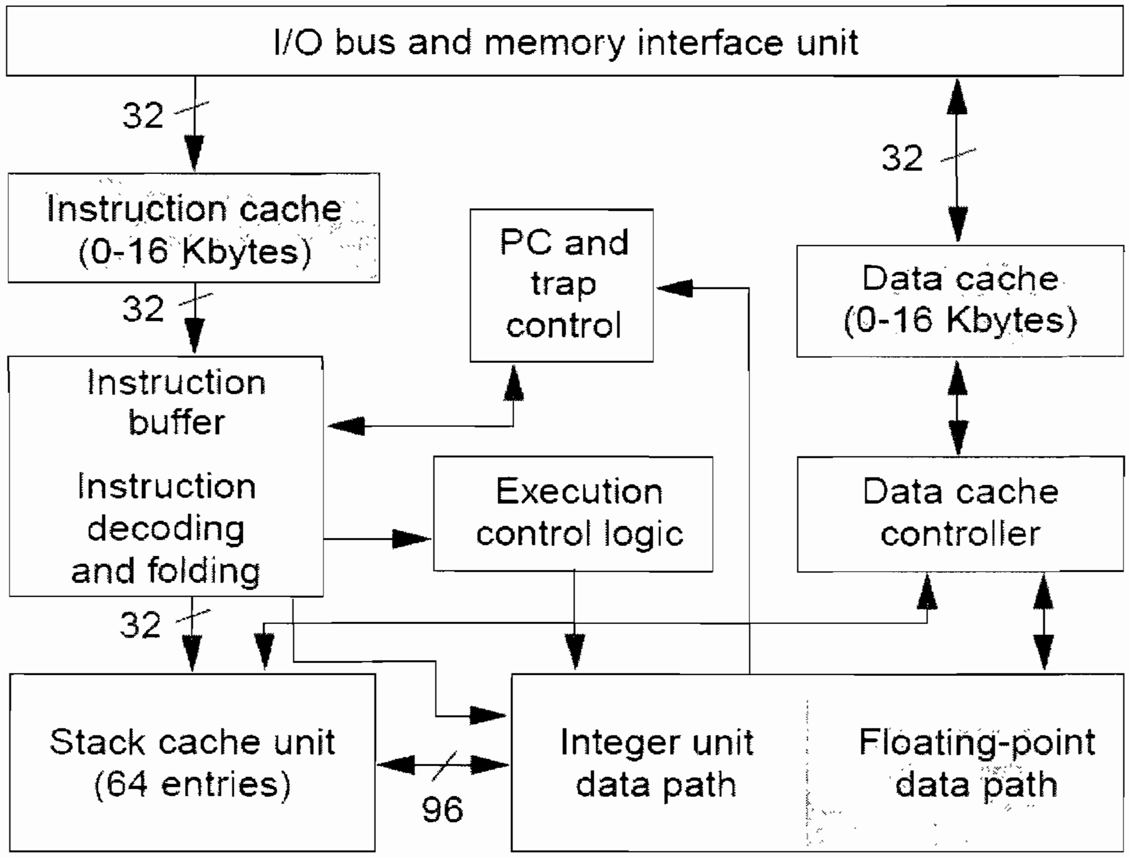


Figure 4: Block-diagram of picoJava-I processor [12]

The instruction cache is a direct-mapped cache with a line size of 8 bytes, while the data cache is a two ways, set-associative, write-back cache. Both of them can be configured between 0 and 16 Kbytes. The picoJava-I processor has a 64-entry stack cache which directly supports the JVM's stack based architecture. The stack cache is implemented as a register file and managed as a circular buffer with a pointer to the top of stack. The picoJava-I allows the option of including or excluding a floating-point unit. The picoJava-I processor includes a RISC-style pipeline and a straightforward instruction set. It implements 341 different instructions. The processor implements simple Java bytecodes in circuitry and executes them in one to three cycles. For example, either integer addition or quick loads of object fields uses a circuitry directly. The picoJava-I implements some performance critical instructions, such as calling a procedure, by using microinstructions. Furthermore, for some complex instructions, such as creating the object or garbage collection, the picoJava processor uses a trap to execute these instructions. One trap needs at least 16 cycles to complete executing. Besides, the picoJava-I processor does not have branch prediction logic. In Figure 5, we present the picoJava-I processor's four-stage pipeline.

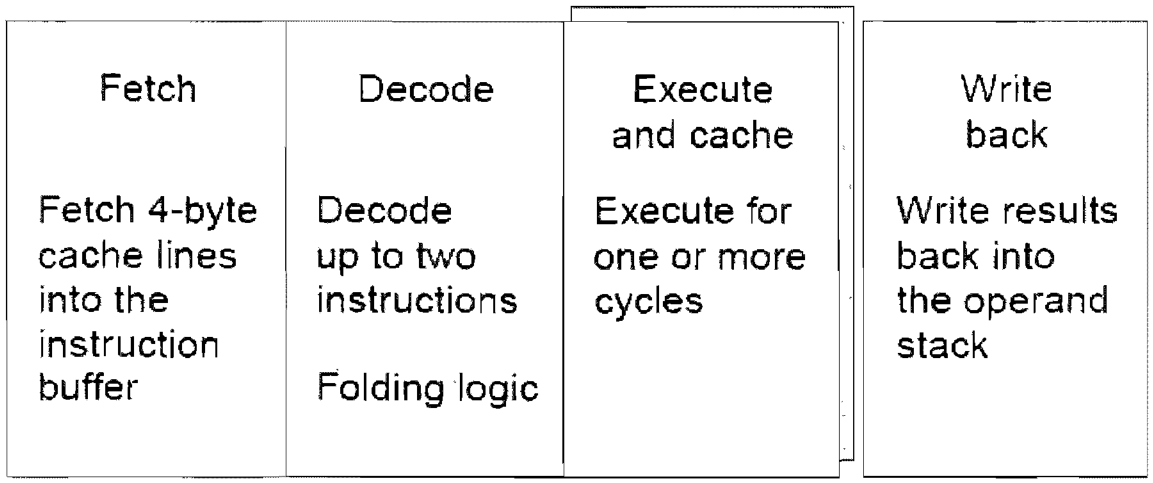
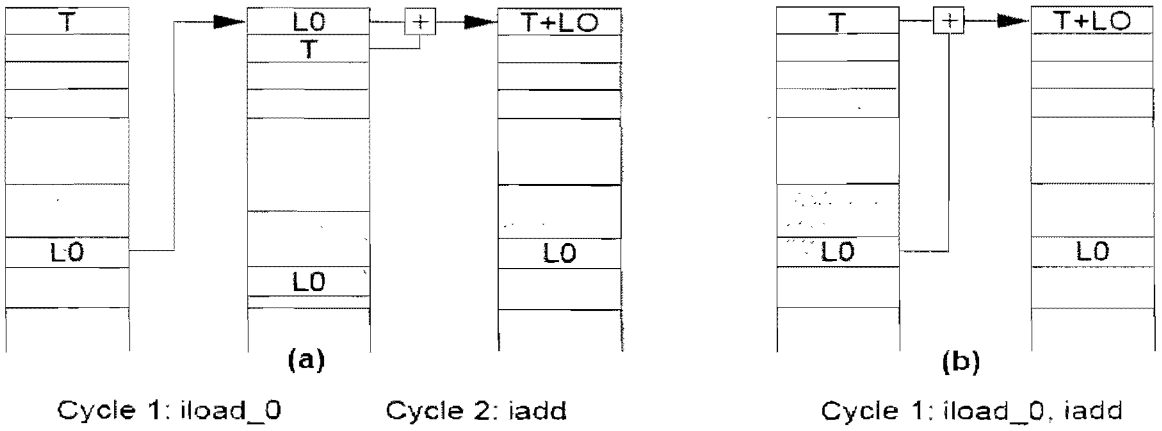

Fetch
Fetch 4-byte cache lines into the instruction buffer
Decode
Decode up to two instructions
Folding logic
Execute and cache
Execute for one or more cycles
Write back
Write results back into the operand stack

Figure 5: PicoJava-I processor's four-stage pipeline [12]

The picoJava-I processor can accelerate Java bytecode execution with a folding operation, which takes advantage of random single-cycle access to the stack cache. Example 2 shows that the processor can reduce one cycle to complete the stack operations by using folding operation.



Example 2: Folding operation [12]

The picoJava-I can be implemented minimal in about 440K gates [14]. Moreover, based on the experiments, the picoJava-I processor can reach 15 to 20 times faster than a 486 with an interpreter at an equal clock rate, and five times faster than a Pentium with a JIT compiler at an equal clock rate.

Co-Designed JVM processor

The co-designed JVM processor [15] is developed by Kent from University of New Bunswick. This processor uses hardware/software partitions for a JVM within the context of a desktop workstation. The motivation of Kent is to relieve performance penalty caused by the translation from Java bytecodes to machine language. The co-designed JVM processor tries to leverage the combined benefits of hardware and software. Instead of 100% on hardware, Kent implemented only part of Java bytecodes on hardware.

The co-designed approach realizes a fully functional JVM comprised of both hardware and software support in a desktop workstation environment. The dedicated hardware, which is supported directly on the workstation mainboard, uses a FPGA tightly coupled with the workstation's general purpose processor through a PCI bus. The partitioning of the design between hardware and software is interesting. The processor uses overlap partitions between hardware and software instead of maintaining disjoint partitions which are normally used in co-designed systems. This partitioning is to relax the conditions to switch execution between hardware/software partitions. The instructions that can be implemented in the hardware partition are those that can be found in traditional processors such as stack manipulation, arithmetic operators and logic operations, comparison and branching, jump and return, as well as data loading and storing. Most of accessed data structures, i.e. the method's bytecode, execution stack and local variables, are placed in the FPGA board memory. The constant pool and the heap reside in the PC's main memory. The software partition executes all object-oriented bytecodes. It supports many complex virtual machine functions, such as class loading and verification, garbage collection, exceptions, as well as memory management. For example, the instructions *new*, *checkcast*, and *instanceof* are executed in software partition. The software partition is responsible for transferring data during context switch between the hardware and software partitions. Furthermore, because some instructions are supported both in the hardware partition and in the software partition due to the overload partitioning, the software partition does a run-time decision to decide where these instructions are executed. The software partition decides during runtime which instruction sequences can be executed by the hardware. The whole system uses a single data bus and a control line to realize a simple communication protocol between the two partitions. Once the hardware partition finishes execution, it signals the software using an interrupt. Then the software partition retrieves the current state of the virtual machine from hardware and continues execution.

The tests of small benchmarks on a simulator show performance gains by a factor of 6 to 11 compared with an interpreting JVM. (Kent does not introduce the machine used to run the software JVM.) Kent does not show benchmark results on FPGA after implementing the processor. The hardware partition is coded in VHDL and the memory uses the memory space within the FPGA device. The interface with the PCI bus is Altera pci_mt64 MegaCore function. Through timing analysis, the maximum clock rate is 24 MHz. The design which implements the full partition (161 instructions) needs 37,756 logic elements with 64 entries instruction cache and data cache. When it uses 16 entries cache, the number of logic elements becomes 33,490.

Lightfoot processor

The 32-bit Lightfoot processor [16] is the product of Digital Communication Technologies. This processor can be used as a design solution of embedded system OEMs from a tiny memory footprint. It is a hybrid 8/32-bit processor based on Harvard architecture. This processor uses a 3-stage pipeline. The instruction memory is 8-bits wide while the data memory is 32-bits wide.

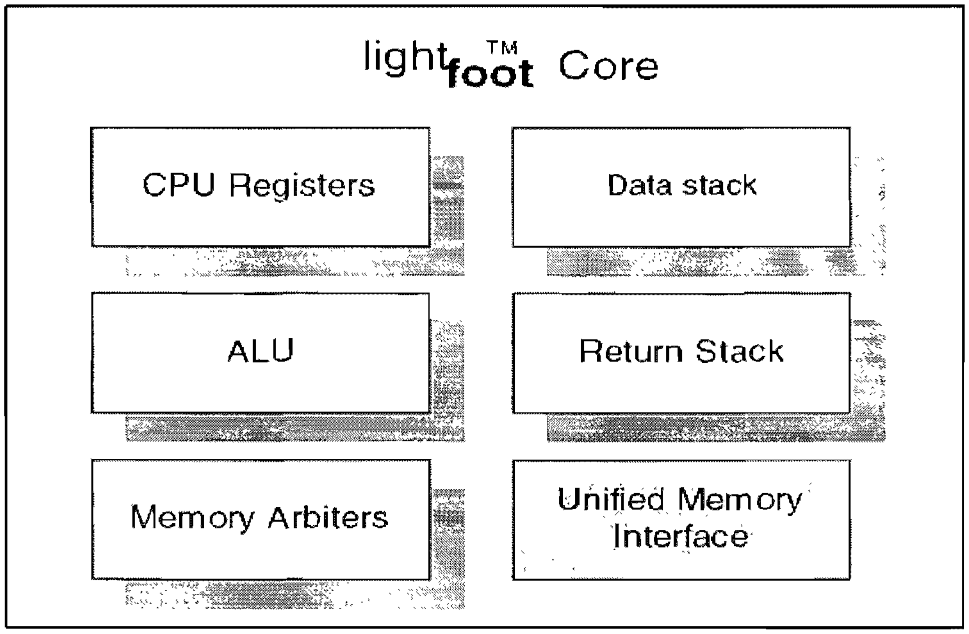


Figure 6: Block-diagram of Lightfoot processor [16]

In Figure 6, we show the key blocks of the Lightfoot processor. The shading part indicates configurability of the memory interface. The user can configure the size of memory and cache. The Control Unit is responsible for fetching, decoding and sequencing the execution of instructions in the processor. The ALU is a traditional 32-bit design. Besides the usual arithmetic and logic capabilities, it has a 32-bit barrel shifter and a 2-bit multiple step unit which can execute a 32×32 bit multiply in 16 cycles. Data stack holds temporary data. The return stack holds return addresses for subroutines. Its top-of-stack element is used as an index register to access program memory. Moreover, the return stack can be used as an auxiliary stack for programs. Both of the two stacks consist of a hardware part and a memory extension. The hardware part of the data stack consists of eight 32-bit on-chip registers while the hardware part of the return stack is four 32-bit registers. The processor has 256 words of register space. The sixteen addresses at the bottom of them are used as CUP registers such as the stack extension pointers, constant and parameter pool pointers. The other register space is for interfacing to system peripherals such as memory management unit. The processor supports the instruction set interpreted from JVM. The Lightfoot processor has three different instruction formats: soft bytecodes, non-returnable instructions and 32 single-byte instructions. The Lightfoot processor implements the 128 soft bytecode instructions in low program memory. When the processor needs to execute a soft bytecode instruction, it branches to one location where the implementation of this soft bytecode resides. The processor needs one cycle to do this operation, and it pushes the address of the following instruction to the return stack. The 32 single-byte instructions can be folded with a return operation. The 32 single-byte instructions have a return bit. When this bit is set, the processor loads the value popped from the return stack to the program counter register. This mechanism implements a zero-overhead return feature of the processor.

The frequency of the Lightfoot processor can reach 31 MHz on Spartan II FPGA family and 40 MHz frequency on Virtex II FPGA families. It can be implemented with less than 30,000 gates for the conventional form.

IBM System z Application Assist Processor (zAAP)

The zAAP [21][35] is the first large-scale commercial Java processor. The zAAP is introduced by IBM in 2004, and available on IBM System z9 and zSeries z990/z890. (Because the zAAP is a commercial product of IBM, we can not find the detailed architecture of this processor. We introduce the zAAP based on the introduction and presentation on IBM official website.) The objective of the zAAP is to integrate Java technology-based applications with mission-critical data and reduce infrastructure complexity for multi-tier applications.

The zAAP is not designed as an independent processor which works individually. Usually the zAAPs work as particular processors and do execute Java programs in IBM System z, IBM's mainframe computers. For example, IBM z990 has 10 CPs (conventional processor), 1 ICF (Integrated Coupling Facility), 2 IFLs (Integrated Facility for Linux), and 3 zAAPs. The zAAPs execute Java programs in IBM JVM under control of z/OS [36], which is IBM's flagship mainframe operating system. When a Java program is to be executed, z/OS dispatches the work units, which the zAAP can support, on a zAAP while z/OS dispatches the left work units on standard processors. In Figure 7, we show how z/OS partitions Java works to zAAPs and general processors.

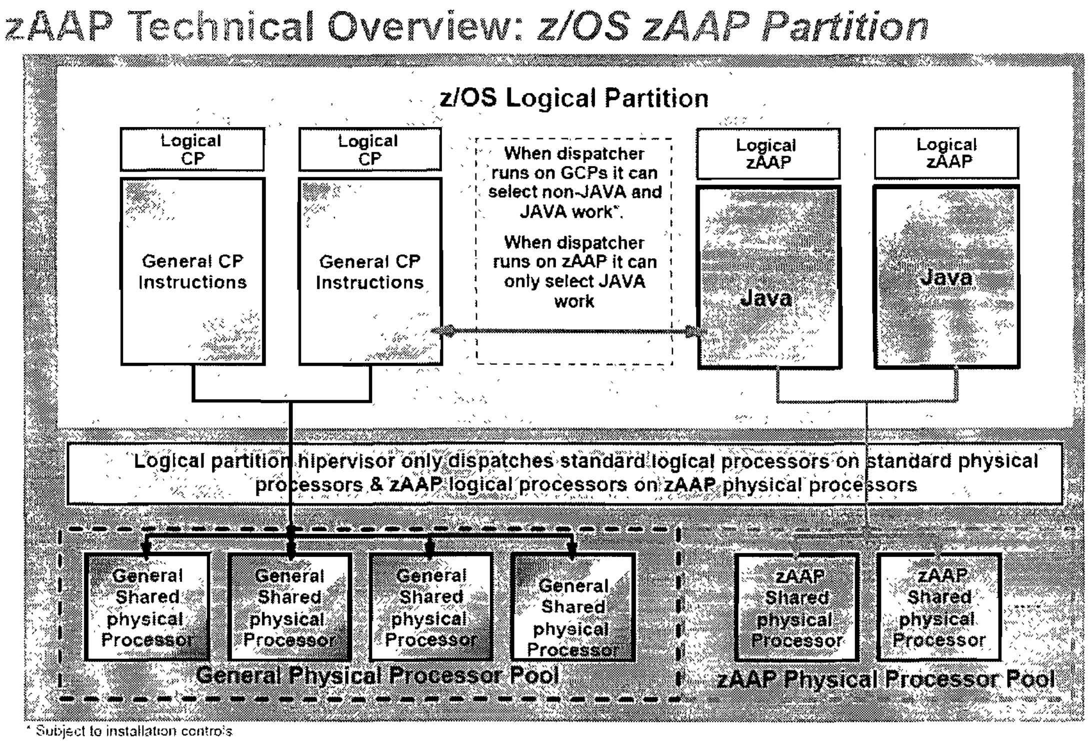

zAAP Technical Overview: z/OS zAAP Partition
z/OS Logical Partition
Logical CP
Logical CP
General CP Instructions
General CP Instructions
When dispatcher runs on GCPs it can select non-JAVA and JAVA work*.
When dispatcher runs on zAAP it can only select JAVA work
Logical zAAP
Logical zAAP
Java
Java
Logical partition hipervisor only dispatches standard logical processors on standard physical processors & zAAP logical processors on zAAP physical processors
General Shared physical Processor
General Shared physical Processor
General Shared physical Processor
General Shared physical Processor
zAAP Shared physical Processor
zAAP Shared physical Processor
General Physical Processor Pool
zAAP Physical Processor Pool
* Subject to installation controls

Figure 7: zAAP Technical Overview [35]

Because the zAAPs share many demands from general purpose processors, general purpose processors can be available for additional workloads. In Figure 8, we show a simple example. With the zAAPs, the system can reduce the standard CP capacity requirement for the application to 500 MIPS or a 50% reduction.

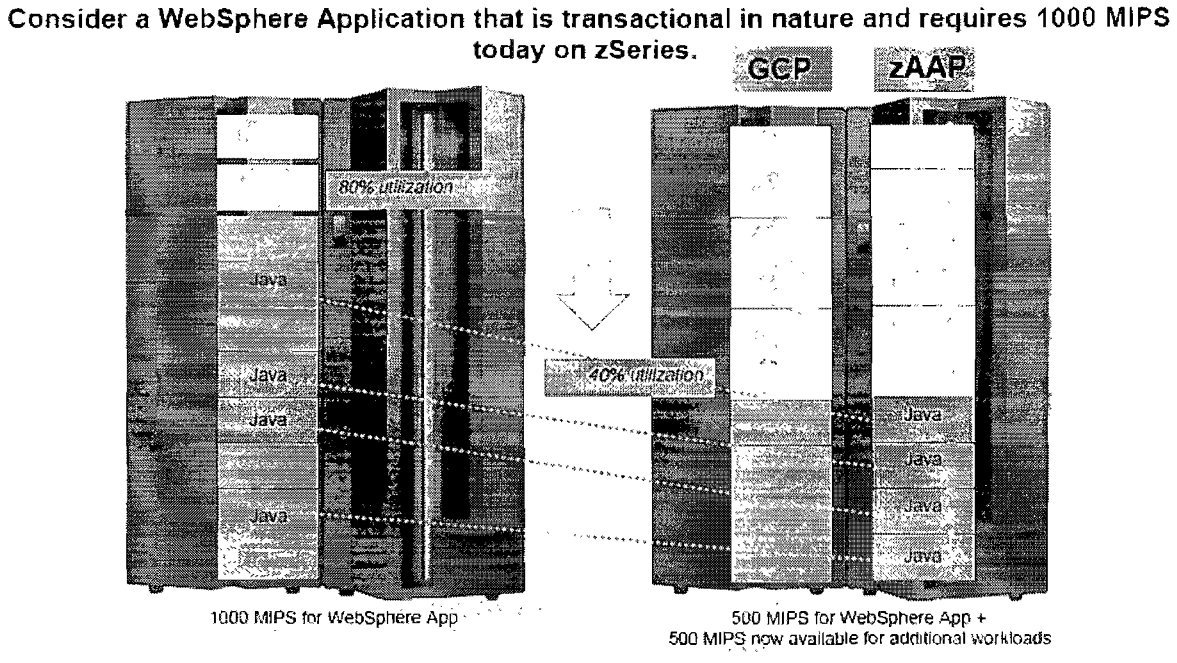

Consider a WebSphere Application that is transactional in nature and requires 1000 MIPS today on zSeries.
GCP
zAAP
80% utilization
Java
Java
Java
Java
40% utilization
Java
Java
Java
Java
1000 MIPS for WebSphere App
500 MIPS for WebSphere App +
500 MIPS now available for additional workloads

Figure 8: An example of zAAP [35]

The zAAP can execute z/Architecture ^{TM} instruction set architecture (ISA) [17][18][19]. The processor does not support all manual operator controls such as PSW (Program status word) Restart, Load or Load derivatives. Moreover, the zAAPs are supported by IBM middleware such as WebSphere, CICS and DB2.

Tanenbaum's IJVM processor

Tanenbaum's IJVM processor [1] is an implementation of micro-architecture. The IJVM processor can execute a subset of integer JVM (IJVM) on hardware. It implements only twenty-two different integer JVM instructions such as *iload* and *istore*. Moreover, the processor does not natively support object-oriented concepts. The Tanenbaum's IJVM processor uses microinstructions and has a seven-stage pipeline. The processor has three data buses and 11 local registers. It does not have prediction logic. In Figure 9, we show the basic architecture of the Tanenbaum's IJVM processor.

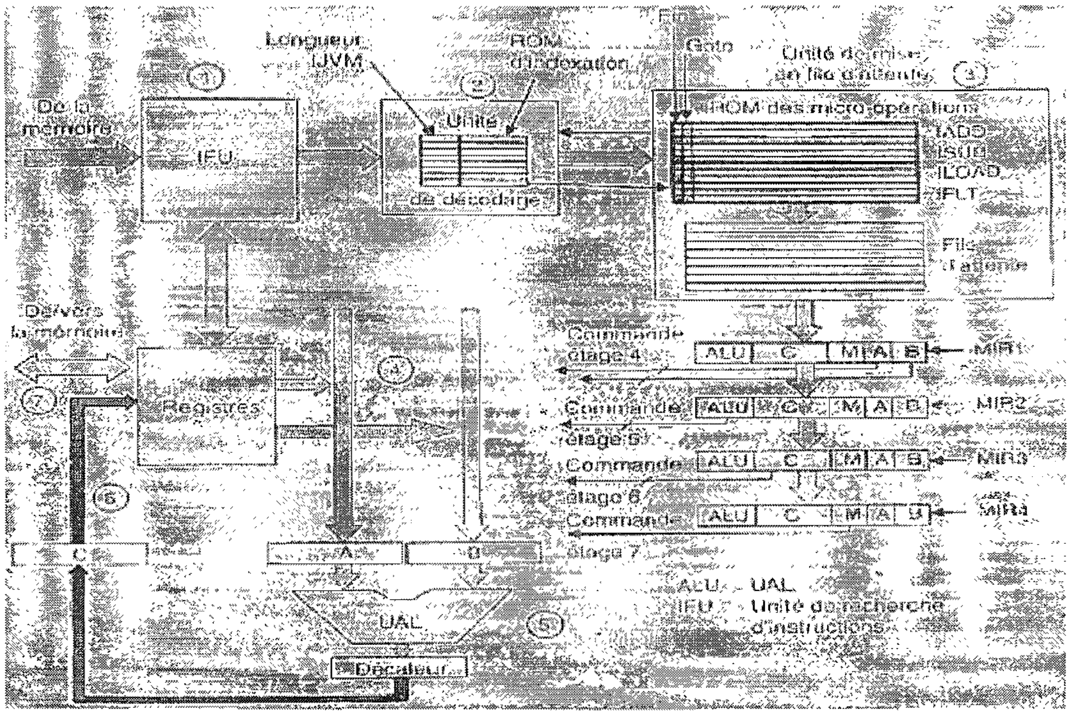


Figure 9: Tanenbaum's IJVM processor [1]

Additional Comments

We showed how the different Java processors implement an instruction set. Because implementing every instruction in circuitry needs too many hardware resources, there are few hardware processors which use this approach in practice. Instead, many processors, such as picoJava-I processor and Lightfoot processor, use the alternative approach where the processor implements simple instructions in circuitry and uses microinstructions to implement the complex instructions and native functions. This approach can reduce the amount of hardware resources and accelerate the execution of those instructions which are used frequently. Of course, the picoJava-I also uses software traps to implement some complex instructions. Using a number of traps may occur to reduce the performance of the processor because the picoJava-I needs minimum 16 cycles to complete a trap operation. The architecture of a processor with two or three different implementation approaches becomes complex.

Another solution we presented is co-design hardware/software approach used in Kent's processor. It partitions the whole implementation to hardware part and software part. Some instructions are implemented in hardware while others are executed in software. In our opinion, this solution is very interesting and suggestive. However, the potential problem is the communication overhead. It is very difficult to predict how much time the processor uses to keep the communication while it executes some programs. Moreover, the raw speedup of the FPGA solution presented by Kent is under the condition that both the clock frequency and general-purpose processor have the same clock frequency. As we all know, the clock frequency of a general-purpose processor is usually 20 to 50 faster than that of FPGA. Therefore, we doubt whether this co-design approach can be realized in practice.

Our SCIL processor adopts a simple approach: the whole instruction set is implemented by microinstructions. In fact, most of small processors use this approach, such as Tanenbaum's IJVM processor and Martin Schoeberl's JOP [37]. During execution, every SCIL instruction is translated to an address and then mapped to one set of microinstructions. The processor can complete this translation in one pipeline stage without execution overheads. Moreover, since there is no instruction implemented in circuitry, the design can be implemented with minimal hardware. In addition, because the architecture of the processor is relatively simple, the development period is shorter than with other approaches. Table 1 lists the Java processors we introduced.

	Instruction set	**Clock frequency**	**Logic usages**	**Implementation approach**
picoJava-I processor	JVM		440K gates	Circuitry Microinstruction Trap
Kent's JVM processor	JVM	24MHz	37K gates	Co-Design
Lightfoot processor	Interpreted from JVM	40MHz	30K gates	Circuitry Microinstruction
Tanenbaum's IJVM processor	IJVM			Microinstruction

Table 1: Various Java processors

The architecture of SCIL processor is inspired by Tanenbaum's IJVM processor. The two processors have some similarities. For example, both of them use three data buses. However, there are also lots of differences between Tanenbaum's IJVM processor and the SCIL processor. First of all, the instruction set our processor implements is SCIL instead of IJVM. The SCIL processor uses a new set of microinstructions, which is different from other existing sets of microinstructions, to implements the SCIL instructions. Furthermore, we change a lot the architecture in order to make the SCIL processor fit the characteristics of the SCIL. For example, the SCIL processor has a different method to deal with branches because the SCIL use absolute address to represent branch address. Moreover, in order to reduce the number of suspending cycles, the SCIL adopts a predictor and the forwarding technique which are not used in the IJVM processor.

We introduced IBM's zAAP to show that the processors for a specific programming language have their commercial usages. As we can see, the zAAP has been used in large-scale commercial field. Furthermore, the zAAP gives one possible method to use the SCIL processor. IBM uses the zAAP as co-processors in IBM's mainframe computers to accelerate the execution of Java programs. Therefore in the future, it is very possible to use the processors for .NET languages in one system to improve the execution of .NET programs.

Chapter 3 CIL vs. SCIL

CIL: high level machine language

In our design, we do not directly use the CIL instructions for our processor. Instead, we create a new intermediate language, named SCIL (Simple CIL). By using a SCIL compiler (we will introduce it later), we translate a CIL program to a SCIL program, and the two programs have the same signification. Then our SCIL processor can execute this SCIL program on FPGA. We use the SCIL to replace the CIL because it is hard to implement the CIL on hardware. The CIL is close to a machine language. It can be assembled into bytecode. At the same time, the CIL is an object language, and it supports object-oriented concepts. Therefore, we can think the CIL as a "high level machine language". In Example 3, we present a piece of CIL program to show its characteristics.

```
.class private auto ansi beforefieldinit Class1 extends [mscorlib]System.Object
{
 .method private hidebysig static int32 zzz(int32 a) cil managed
 {
  .maxstack  2
  .locals init ([0] int32 b,
         [1] int32 CS$00000003$00000000)
  IL_0000:  ldc.i4.6
  IL_0001:  stloc.0
  IL_0002:  ldarg.0
  IL_0003:  ldloc.0
  IL_0004:  add
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
  IL_0007:  stloc.1
  IL_0008:  br.s      IL_000a
  IL_000a:  ldloc.1
  IL_000b:  ret
 } // end of method Class1::zzz
 .method private hidebysig static int32 Main(string[] args) cil managed
 {
  .entrypoint
  .maxstack  2
  .locals init ([0] int32 x,
         [1] int32 CS$00000003$00000000)
  IL_0000:  ldc.i4.0
  IL_0001:  stloc.0
  IL_0002:  br.s      IL_000d
  IL_0004:  ldloc.0
```

```
    IL_0005:  ldc.i4.5
    IL_0006:  call       int32 ConsoleApplication1.Class1::zzz(int32)
    IL_000b:  add
    IL_000c:  stloc.0
    IL_000d:  ldloc.0
    IL_000e:  ldc.i4.s   100
    IL_0010:  blt.s      IL_0004
    IL_0012:  ldloc.0
    IL_0013:  stloc.1
    IL_0014:  br.s       IL_0016
    IL_0016:  ldloc.1
    IL_0017:  ret
  } // end of method Class1::Main
} // end of class Class1
```

Example 3: The "high level machine language" CIL

As we can see, the CIL is close to a machine language because every CIL instruction can be expressed in form of bytecode instruction, such as "ldarg 0" or "br.s IL_000a". However we can find that the CIL is similar to high level languages such as C++ and Java. The CIL uses "return value + function name + parameter type" to declare one procedure, and uses one pair of "{" "}" to express the beginning and end of one procedure or class. The CIL supports object-oriented programming. Usually only high level languages have these characteristics. As a result, these high level language characteristics make the CIL much different from machine languages. A CIL program is more readable than a JVM program. However, it is a real nightmare for the hardware designers to construct the hardware solutions that implement the "machine language" CIL. It is difficult for one processor to know how to deal with some complex and tedious statements such as "private hidebysig static int32 zzz(int32 p) cil managed", or identify some characters like '{' and '"'. Therefore, we give up using the CIL as the machine language for our processor. We adopt an alternative approach: before loading CIL instructions into memory, we translate CIL instructions to the real machine instructions, and then implement these machine instructions on hardware. In this way, we can avoid using the complex CIL statements and success in executing the CIL program on hardware. Furthermore, we can reduce significantly the total amount of hardware resources and shorten the development period.

SCIL: Simplifying CIL

The SCIL is designed for our processor, and it is the real machine language for the processor. The basic idea of generating SCIL instructions is to simplify the CIL. The SCIL has no Metadata, and only consists of bytecode instructions. The SCIL is equivalent to a subset of the CIL. Nowadays it only supports 32 bits signed integer, and does not support floating-point operations and object-oriented concepts.

There exist a lot of differences between the CIL and the SCIL. First of all, the instruction codes of the SCIL are completely different from their prototypes, the instruction codes of the CIL. We redefine all instruction codes of SCIL instructions. For example, the instruction code of *add* is 0x58 in CIL when the new instruction code is 0x01 in SCIL. Secondly, for some SCIL instructions, although they still own the same names as CIL instructions, their instruction operands may represent different signification. Taking the instruction *call* for example, the instruction operand of *call* in CIL represents the name of invoking procedure, however the instruction operand of *call* in SCIL represents the branch address of invoking procedure. Thirdly, because usually one SCIL instruction corresponds to several CIL instructions, the SCIL has much less instructions than the CIL has. (We will introduce these differences in following sections.)

Metadata

The SCIL has no Metadata any more. It is not easy for a processor to deal with Metadata because usually Metadata have various forms and different parameters. We remove Metadata by using three ways. Firstly, we directly delete some Metadata. Because now the SCIL is not an object language, many Metadata are not useful any more. Therefore, although we throw away these Metadata to garbage, we do not change the signification of the whole program. For example, the Metadata *.class*, this Metadata is useful for the CIL to declare the beginning of one class definition. However, the SCIL does not have the concept of class. Hence, this Metadata becomes not useful, and we can delete it without hesitation.

The second way of dealing with Metadata is to remove the Metadata but still complete this Metadata's job. For example, the Metadata *.entrypoint* represents which procedure is the main procedure in the program. For the SCIL, there is not one instruction which functions as to indicate the position of the main procedure. Alternatively, we define that all SCIL programs are executed from the first instruction, the first line of the source code. Then we move the procedures, which contains the Metadata *.entrypoint* in CIL, to the beginning of the SCIL programs. In this way, we can delete the Metadata *.entrypoint* when the SCIL programs still know the position of the main procedure.

The third way is that we use SCIL instructions to replace some Metadata. For example, we replace the Metadata *.locals* with the SCIL instruction *local opd*. We show an example for Metadata *.locals*.

```
.locals init ([0] int32 a,
              [1] int32 b)
```

In this example, the Metadata *.locals* expresses that there are two local variables *a* and *b* in the current procedure. For the SCIL, it is not necessary to remember the name of variables because the SCIL always use a number to represent a local variable. Instead, the number of local variables is very useful for the SCIL. For the above example, the SCIL will name variable *a* as 0^{th} variable *b* as 1^{st} variable. Therefore it is necessary to keep the number of variables from the Metadata *.locals*. We use a SCIL instruction *local opd*, where *opd* represents the number of variables, to replace the Metadata *.locals*. Therefore, we can replace the Metadata *.locals* in the above example with the SCIL instruction *local 2*. In this way, the SCIL can keep the information the CIL Metadata contain by only using SCIL instructions.

Regrouping Instructions

We do some simplifications for CIL instructions while translating them to SCIL instructions. We reduce the number of CIL instructions. Usually, several CIL instructions correspond to only one SCIL instruction. For example, loading an integer value to the top of stack is an often-used operation in the CIL, and the CIL has a series of instructions to do loading an integer with different value. (See Table 2)

CIL instruction	**Comment**
ldc.i4.m1	Load integer -1 to the top of stack
ldc.i4.0	Load integer 0 to the top of stack
ldc.i4.1	Load integer 1 to the top of stack
ldc.i4.2	Load integer 2 to the top of stack
ldc.i4.3	Load integer 3 to the top of stack
ldc.i4.4	Load integer 4 to the top of stack
ldc.i4.5	Load integer 5 to the top of stack
ldc.i4.6	Load integer 6 to the top of stack
ldc.i4.7	Load integer 7 to the top of stack
ldc.i4.8	Load integer 8 to the top of stack
ldc.i4.s opd	Load integer opd to the top of stack

Table 2: Various CIL loading instructions

If we kept this series of loading instructions without any change, our processor should have implemented them as ten different instructions. Now the SCIL only uses one instruction, *loads opd*, where *opd* represents the value of the integer loaded, to replace all these ten CIL loading instructions. For example, the CIL instruction *ldc.i4.m1* can be represented by the SCIL instruction *loads –1*; and the CIL instruction *ldc.i4.0* can be represented by the SCIL instruction *loads 0.* As a result, our processor can only use almost 1/10 hardware resources which are needed to implement all ten loading instructions. Such a simplification also refers to the CIL instructions such as *ldarg*, *ldloc* and *stloc.*

In fact, if the processor can implement some instructions in circuitry, the speed of execution may be improved. For example, the SCIL can use three instructions to represent the various CIL loading instructions: *loads 0*, *loads 1* and *loads opd*, and the processor implements *loads 0* and *loads 1* in circuitry. As a result, because the first two instructions are used frequently, the processor can accelerate execution of programs. Moreover, just implementing two instructions in circuitry do not need lots of additional hardware resources. However, the problem is how many and which instructions could be implemented in circuitry. Furthermore, the architectural of the processor with many circuitries becomes more complex than that of our current processor. We think maybe the SCIL processor could implement part of instructions in circuitry in future woks.

Absolute address

When translating CIL branch instructions such as *br.s*, *bge* and *call* to SCIL instructions, we change the instruction operands of these CIL branch instructions. We use the absolute branch addresses as the instruction operands of the SCIL instructions.

The JIT compilers allocate memory for CIL instructions at application run time. (The different JIT compilers may use different methods to allocate memory address. We use the document Microsoft's .NET Framework Developer's Guide [24] as our reference.) Before CIL instructions are executed, all CIL instructions are kept in the PE file. When these CIL instructions are executed, one .NET Framework JIT compiler is responsible to convert them to native code. During the execution, the JIT compiler does not convert all CIL instructions to native code at one time. The compiler does not load one procedure (or method) until this procedure (or method) is needed. When it is the first time to invoke one procedure, the JIT compiler converts this block of instructions, all of the CIL code for this procedure, to native code. Then the JIT compiler locals the native code in memory. Subsequent calls of the compiled procedure are proceed directly to the native code that was previously generated. The benefit is that some code which never gets invoked during the execution is not loaded in memory. Rather than using time and memory to convert all the CIL to native code, the JIT compiler only converts the CIL needed during execution and stores the resulting native code.

In Example 4, we show a piece of CIL program. For each CIL instruction, it has a label like *IL_xxxx*, which are generated by the CIL compiler. In this example, we suppose that the first instruction *ldc.i4.6* is the beginning of one procedure, and all instructions are in the same procedure. When this procedure is invoked, the JIT compiler converts all CIL instructions in this procedure to native code, and loads them to memory. The numbers in parentheses before each CIL instruction in the example represents the memory address, which is allocated by the JIT compiler.

```
(0x0105)    IL_0000:  ldc.i4.6
(0x0106)    IL_0001:  stloc.0
(0x0107)    IL_0002:  ldarg.0
(0x0108)    IL_0003:  ldloc.0
(0x0109)    IL_0004:  add
(0x010A)    IL_0005:  stloc.0
(0x010B)    IL_0006:  ldloc.0
(0x010C)    IL_0007:  stloc.1
(0x010D)    IL_0008:  br.s      IL_0002
```

Example 4: Branch instruction in CIL

The instruction *br.s IL_0002* is a branch instruction, which represent that the program goes to the label IL_0002 when the first element is bigger than the second. The label *IL_0002* in the branch instruction does not represent the absolute branch address but the relative displacement to the beginning of current procedure. Therefore, in order to obtain the target branch address, the compiler has to do a calculation based on this relative displacement. It adds the displacement (IL_0002 – IL0000 = 2) to the first instruction's memory address of the current procedure (0x0105), and then the compiler can obtain the branch address (0x0107).

Unlike the CIL, the SCIL uses static allocation to allocate SCIL instructions in memory. All SCIL instruction will be loaded in memory whatever they are executed or not. Before being loaded in memory, we can know the memory address of every SCIL instruction. The first instruction of the main procedure always occupies the memory address 0x0000. The SCIL compiler calculates the address of every instruction in memory. Clearly it is not a Just-In-Time compilation. However, this kind of memory allocation can reduce the workload of the SCIL processor because such an expression enables the processor to avoid branch address calculation on the fly. When the processor obtains an SCIL instruction, it can immediately know where the next instruction in memory. Hence the processor does not need any operations to calculate memory addresses.

The SCIL uses the absolute address to represent branch addresses. When the SCIL compiler translates CIL instructions to SCIL instructions, it calculates the branch address for all branch instructions. Then the SCIL branch instructions use branch addresses as their instruction operands. In Example 5, we show the SCIL instructions equivalent to the CIL instructions presented in Example 4. We suppose that the address in parentheses is the memory address in instruction memory for the SCIL processor. The last instruction br 0x002E is equivalent to the instruction br.s IL_0002 in Example 4. Now the instruction operand 0x002E represents the branch address. When the processor executes this instruction, the processor knows the memory address of the next instruction is 0x002E.

```
(0x0029)      loads  6
(0x002C)      stloc  0
(0x002E)      ldarg  0
(0x0030)      ldloc  0
(0x0032)      add
(0x0033)      stloc  0
(0x0035)      ldloc  0
(0x0037)      stloc  1
(0x0039)      br     0x002E
```

Example 5: Branch instruction in SCIL

For the CIL, the *call* instruction includes the invoking procedure name, the type of parameters and the type of return value. When the JIT compiler executes a *call* instruction, the compiler searches the list of procedures to check the procedure name, the type of parameters and the type of return value. Then the JIT compiler examines the CIL instruction and Metadata to determine whether the code is type safe, which means a reference to a type is strictly compatible with the type being referenced. Only appropriately defined calling operations can invoke a procedure.

In Example 6, we show a piece of CIL program. We also add the memory address allocated by the JIT compiler for each instruction in parenthesis. We suppose that the first part of instructions (the first five instructions) is in the main procedure, and the second part of instructions is in another procedure named zzz. In the main procedure, there is a call instruction, call int32 Test.TestClass::zzz(int32), which invokes the procedure zzz. We can see that the call instruction provides lots of information to the JIT compiler.

```
.
.
```

```
(0x0120) IL_000e: ldloc.1
(0x0121) IL_000f: stloc.2
(0x0122) IL_0010: ldloc.2
(0x0123) IL_0011: call      int32 Test.TestClass::zzz(int32)
(0x0125) IL_0016: stloc.2
         .
         .
.method private hidebysig static int32
     zzz(int32 p) cil managed
 (0x0307)    IL_0000: ldarg.0
 (0x0308)    IL_0001: ldloc.0
 (0x0309)    IL_0002: add
         .
         .
```

Example 6: Procedure call in CIL

For the SCIL, the SCIL compiler does the job of searching the target procedure. Unlike the JIT compiler searching the invoking procedures during the execution, the SCIL compiler finds out the position of the invoking procedures before SCIL instructions are loaded in memory. Furthermore the SCIL uses the branch address as the instruction operand of the SCIL instruction call. The branch address is the memory address of the first instruction in the invoking procedure. In Example 7, we show the SCIL instructions equivalent to the CIL instructions presented in Example 6. We suppose that the address in parentheses is the memory address in instruction memory for the SCIL processor. The new call instruction becomes call 0x0027. The operand 0x0027 is the branch address which is the memory address of the first instruction in the procedure zzz.

```
         .
         .
(0x0002)      loads  1
(0x0005)      stloc  1
(0x0007)      loads  2
(0x0009)      call   0x0027
(0x000B)      stloc  2
         .
         .
(0x0027)      local  2
(0x002E)      ldarg  0
(0x0030)      ldloc  0
(0x0032)      add
         .
         .
```

Example 7: Procedure call in CIL

By using absolute addresses to represent branch addresses, all branch instructions can tell the processor where the next instruction in the instruction memory. The processor does not need calculate branch addresses during the execution, and the processor can hence obtain faster speed than with ordinary CIL. However, the method of absolute address can only used in simple embedded system designs. If we make the processor support object-oriented concept, we still need to use dynamic branch calculation. At this time, the processor has to know the position of instructions in some class instantiations, and the branch addresses are different for the same branch instructions.

Three types of SCIL instructions

Based on the length of bits which one SCIL instruction requires, we divided all SCIL instructions into three types. The Type 1 SCIL instruction occupies 8 bits and does not have instruction operand. It needs one word (8 bits per word) in the instruction memory for the SCIL processor. The Type 2 SCIL instruction needs 16 bits and two words in the memory. The first 8 bits represent the instruction code, and the rest bits represent the 8 bit signed integer operand. The Type 3 SCIL instruction demands 24 bits and three words in memory. The first 8 bits are for instruction code and the others are as the instruction operand, a signed 16 bits integer. Furthermore, in order to discriminate the types of SCIL instructions easily, we use the first two bits of instruction code to identify the different types. The Type 1 instructions begin with two bits "00"; the Type 2 instructions begin with "01"; and the first bit of the Type 3 instruction is "1". In Table 3, we show three examples for the three types SCIL instructions, and we also show them in binary form when they are loaded in the instruction memory.

	Examples	Binary form in memory
Type 1	add	00000010
Type 2	ldarg 2	01000101 00000010

Type 3	call 0x02	10000010 00000000 00000010

Table 3: Three types SCIL instructions

SCIL compiler

We use a simple compiler to translate CIL programs to SCIL programs. The SCIL compiler is written in language C++. (There is no any special reason why we use C++ rather than other languages.) The compiler has two files: compiler.cpp and compiler.h. By using the Visual Studio .NET tool ildasm.exe, which is usually in the path "..\Microsoft Visual Studio .NET \SDK\v1.1\Bin\ildasm.exe", we can obtain the CIL file from any .NET PE file. Then the SCIL compiler executes this CIL file, and converts the CIL to the SCIL. The result will be saved as a SCIL file (.scil file). In the SCIL file, all SCIL instructions are decoded in binary form, and each line is 8 bits. Finally, we use this SCIL file as the initial file for the instruction memory. In Figure 10, we show the process of converting a PE file to a SCIL file, and then loading it into FPGA.

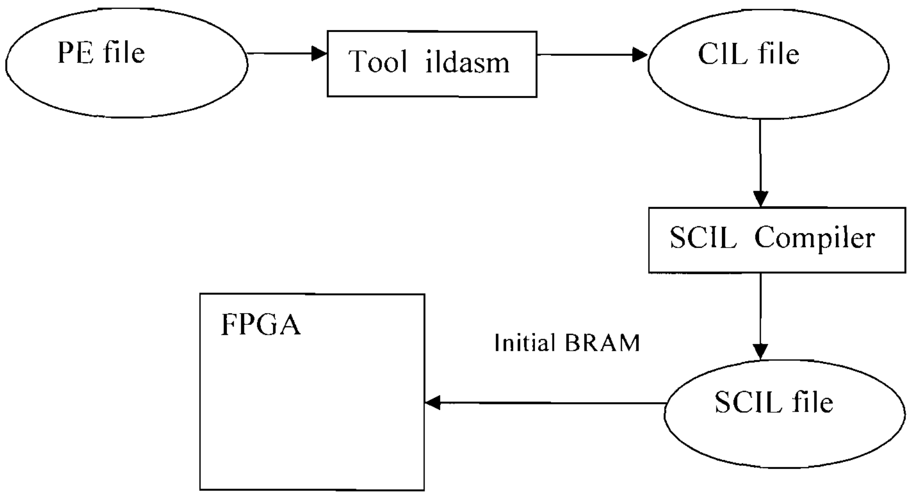


Figure 10: Convert PE file to SCIL file

A SCIL example

The SCIL program shown in Example 8 is equivalent to the CIL program we presented in Example 3. We use the SCIL compiler to execute the CIL program and then obtain the corresponding SCIL program. The two programs have the equivalent signification.

```
(0x0000)    local  2
(0x0002)    loads  0
(0x0005)    stloc  0
(0x0007)    br     0x0015
(0x000A)    ldloc  0
(0x000C)    loads  5
(0x000F)    call   0x0027
(0x0012)    add
(0x0013)    stloc  0
(0x0015)    ldloc  0
(0x0017)    loads  100
(0x001A)    blt    0x000A
(0x001D)    ldloc  0
(0x001F)    stloc  1
(0x0021)    br     0x0024
(0x0024)    ldloc  1
(0x0026)    ret_main
(0x0027)    local  2
(0x0029)    loads  6
(0x002C)    stloc  0
(0x002E)    ldarg  0
(0x0030)    ldloc  0
(0x0032)    add
(0x0033)    stloc  0
(0x0035)    ldloc  0
(0x0037)    stloc  1
(0x0039)    br     0x003C
(0x003C)    ldloc  1
(0x003F)    ret
```

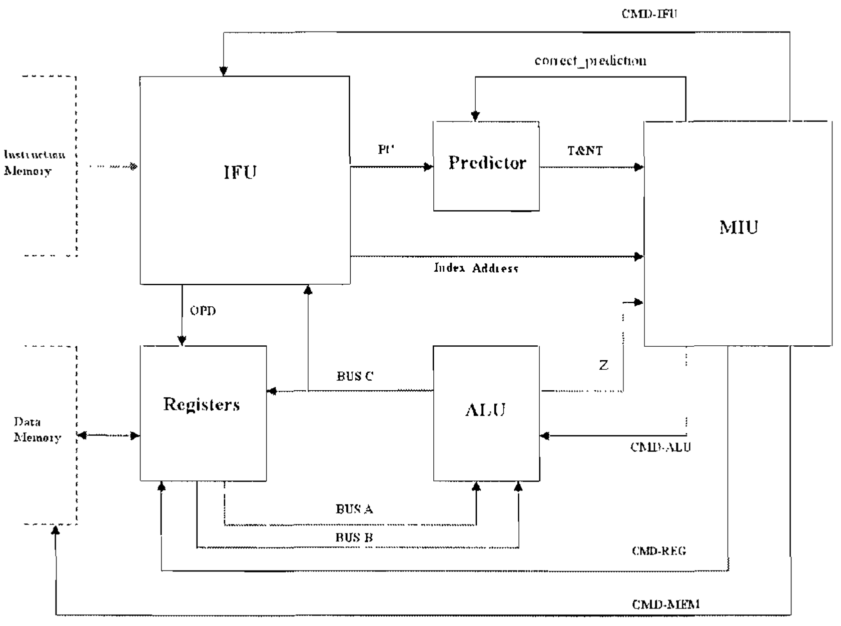
Example 8: SCIL program equivalent to CIL program in Example 3

Chapter 4 SCIL Processor

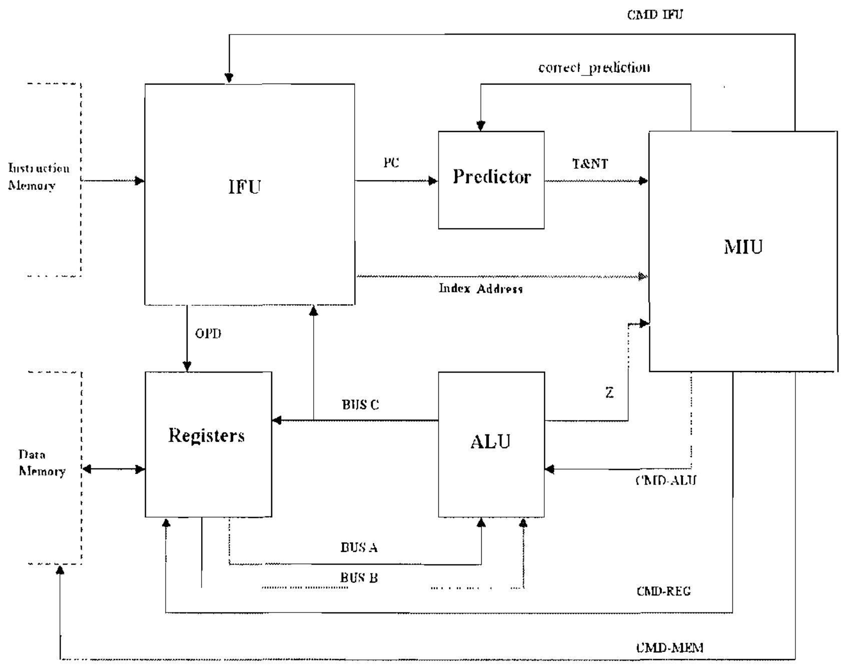
In this chapter, firstly, we will introduce the basic data flow and six-stage pipeline of the SCIL processor. Then we will present the format of microinstructions for the SCIL processor. After that, we will introduce the architecture of the SCIL processor in detail. We will discuss all the principal units of the processor one by one and point out characteristics of them.

Data flow

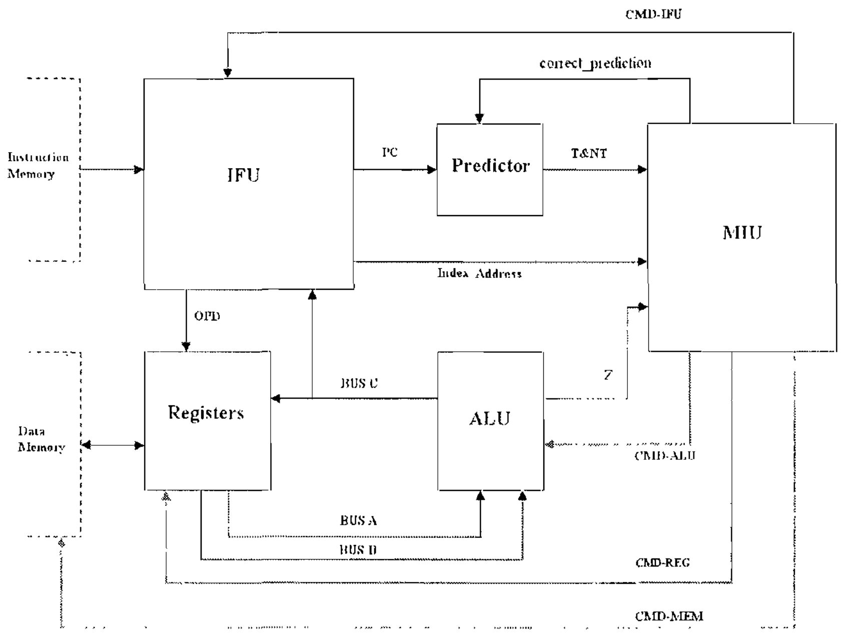
In Figure 11, we present the basic data flow of the SCIL processor. The SCIL processor takes data from the instruction memory (a), and the data successively pass through the unit IFU (b), and the unit MIU (c) to find out microinstructions. Then according to command signals derived from these microinstructions, the SCIL processor sends the value of the registers to the ALU though BUS A and BUS B (d). Then the ALU does the arithmetic calculations. After that, the ALU outputs the result of calculation on BUS C (e). At this time, the processor updates the value of registers with the data on BUS C. Finally, the processor writes or reads the data memory (f).



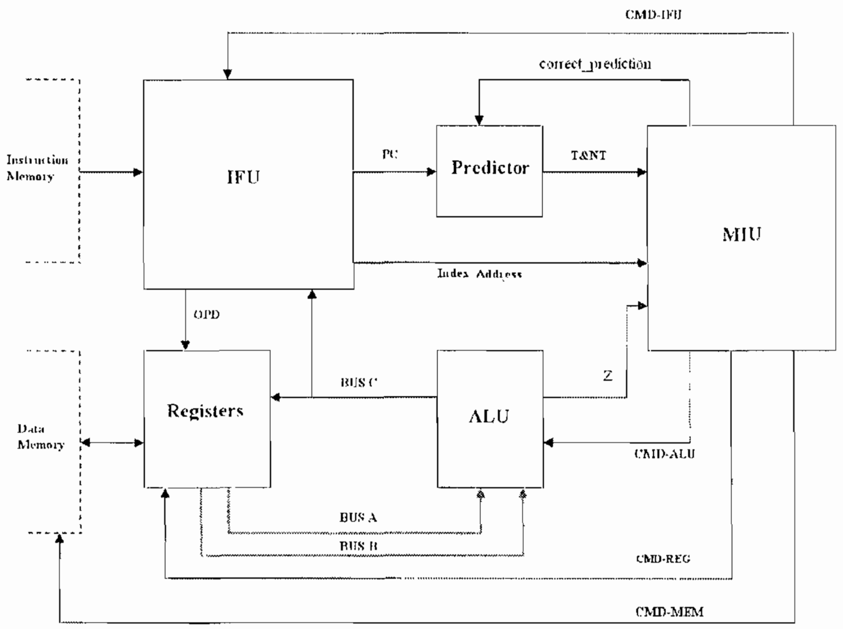
(a)


CMD IFU
correct_prediction
Instruction Memory
IFU
PC
Predictor
T&NT
MIU
Index Address
OPD
Data Memory
Registers
BUS C
ALU
Z
CMD-ALU
BUS A
BUS B
CMD-REG
CMD-MEM

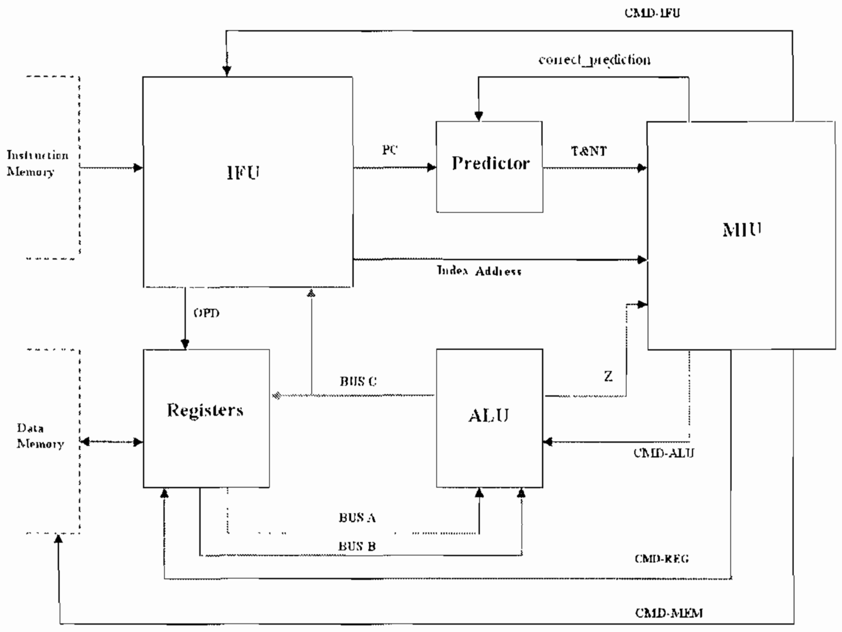
(b)


CMD-IFU
correct_prediction
Instruction Memory
IFU
PC
Predictor
T&NT
MIU
Index Address
OPD
Data Memory
Registers
BUS C
ALU
Z
CMD-ALU
BUS A
BUS B
CMD-REG
CMD-MEM

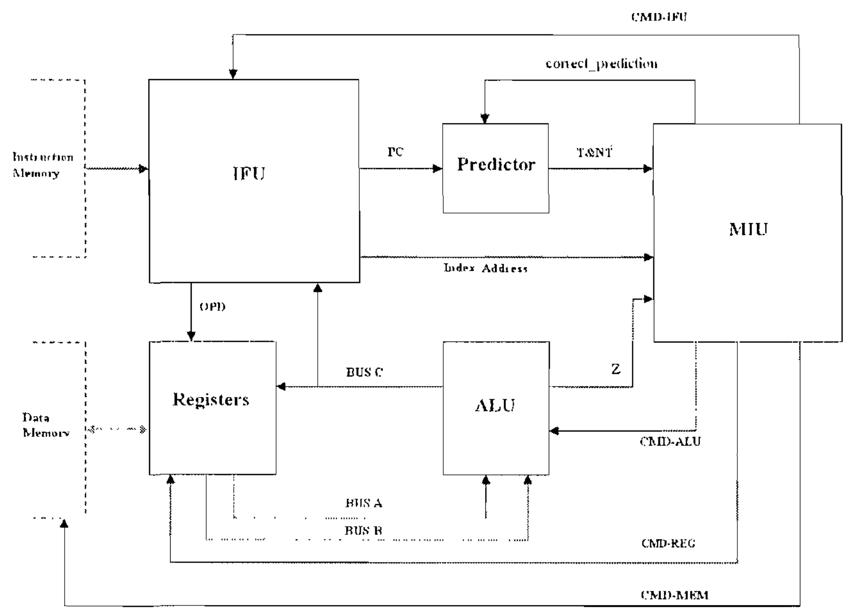
(c)


CMD-IFU
correct_prediction
Instruction Memory
IFU
PC
Predictor
T&NT
MIU
Index Address
OPD
Data Memory
Registers
BUS C
ALU
Z
CMD-ALU
BUS A
BUS B
CMD-REG
CMD-MEM

(d)


CMD-IFU
correct_prediction
Instruction Memory
IFU
PC
Predictor
T&NT
MIU
Index Address
OPD
Data Memory
Registers
BUS C
ALU
Z
CMD-ALU
BUS A
BUS B
CMD-REG
CMD-MEM

(e)



(f)

Figure 11: Data flow of SCIL processor

Six-stage pipeline

Based on relative-independent actions of the processor, we divide the whole data flow into six steps. We make each step as one stage of the pipeline. Therefore in the current design, the SCIL processor uses a six-stage pipeline. We present the pipeline in Figure 12. The six pipeline stages are: *Fetch*, *Decode*, *Register Read*, *Execution*, *Register Write-back* and *Memory Access*.

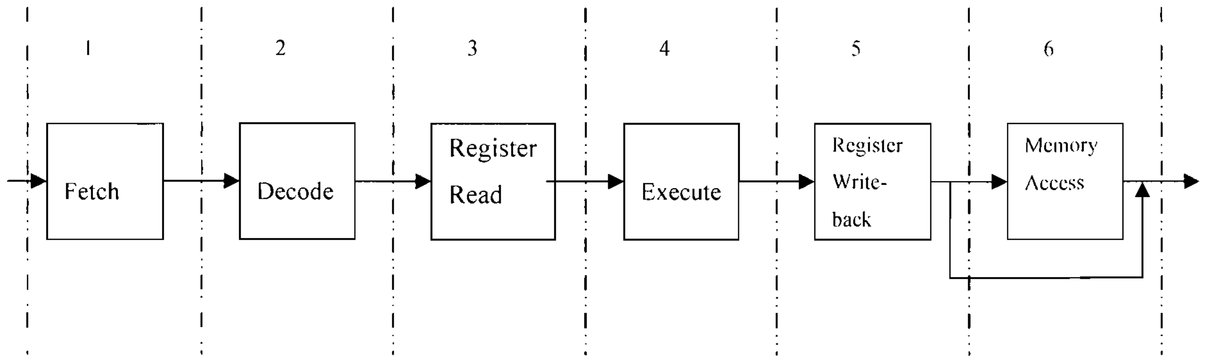


Figure 12: Six-stage pipeline

In the stage *Fetch*, the IFU converts the data taken from the instruction memory to instruction code and instruction operand, and then sends the corresponding index address to the unit MIU. In the second stage *Decode*, the MIU uses the index address to search the microinstructions, and generate command signals. In the third stage *Register Read*, the processor takes the data of two registers and sends them to the ALU. In the fourth stage *Execute*, the ALU does one arithmetic calculation. In the fifth stage *Register Write-back*, the processor updates the value of registers with the calculation result. In the last stage *Memory Access*, the SCIL processor writes or reads data memory.

Comparing with Tanenbaum's IJVM processor using a seven-stage pipeline, the SCIL processor reduces one stage of pipeline. We combine two stages of the IJVM processor's pipeline into one stage. In the IJVM processor, the first stage is responsible to take data from memory, and the second stage is to convert instruction codes to index addresses. The SCIL processor's first stage *Fetch* does the tasks of the IJVM processor's first and second stages now. With the six-stage pipeline, the SCIL processor can begin to execute the microinstructions one cycle earlier. The SCIL processor needs at least 2 cycles to prepare the microinstructions for a new SCIL instruction. The first cycle is to take an instruction from memory and get the index address; the second cycle is to search the corresponding microinstruction set. Therefore, the SCIL processor can begin to execute the microinstructions at the third cycle. When the processor used a seven-stage pipeline, the processor would need at least three cycles before the new SCIL instruction's microinstructions can be executed. The first cycle is to take the instruction; the second cycle is to get the index address; the third cycle is to search microinstruction set. The processor could begin to execute the microinstructions at the fourth cycle. Hence the processor can use less cycles with the six-stage pipeline. Furthermore, although we combined two stages into one stage, we do not change frequency of the processor. Based on timing analyse for the SCIL processor, the unit MIU, the unit for the stage *Decode*, is the unit that needs the most time. The second stage *Decode* needs more time than the first stage *Fetch*, and the time used by the second stage decide the clock frequency of the processor. Therefore, we make the SCIL processor use the six-stage pipeline instead of the seven-stage pipeline.

Microinstructions for SCIL Processor

For different micro-architectures, the microinstructions are different. In this section, we introduce the microinstructions target for the SCIL processor.

Notation of Microinstructions for SCIL Processor

In Example 9, we show several microinstructions for the SCIL processor. These microinstructions are in binary form and each line represents one microinstruction. It is not easy to understand what one microinstruction represents.

```
000001111000011110101 1
```

Example 9: Microinstructions in binary form

Therefore, here we use a kind of notations to represent microinstructions in order to be convenient to express their meanings. We will use these notations in the following paragraphs. In Example 10, we show two typical microinstructions written in notation form.

```
(a) SS=SS+1
(b) MDR=TOS=TOS+TPR; Wr
```

Example 10: Microinstructions in notation form

The new expression makes microinstructions be similar to a high level language. The capital form terms, such as "SS" and "MDR" represent the registers. The operation symbols, such as "+" and "-", represent the operations of the processor's ALU. The terms "Wr" and "Rd" represent writing and reading the memory respectively. The equal mark represents using the value of right side to update the register of left side. Moreover, it is possible to use more than one equal mark in one microinstruction. (See the Example 10(b)) It means the rightmost value is used to update several registers of left side at one time.

In the Example 10(a), the microinstruction refers to only one register named SS, and the task of this microinstruction is to increase the value the register SS with 1. In the Example

10(b), the microinstruction refers to four registers: MDR, TOS, TOS and TPR. The task of this microinstruction is to update the registers MDR and TOS with the sum of TOS and TPR. Moreover, the "Wr" represents that this microinstruction includes a writing memory operation.

Implement SCIL instructions with Microinstructions

For the SCIL processor, every SCIL instruction corresponds to a set of microinstructions, and the SCIL processor can complete the functionality of one SCIL instruction with executing a set of microinstructions in turn. Usually such a set includes 2~9 microinstructions and the average number is about 4.5. In Example 11, we show the set of microinstructions for the SCIL instruction *add*. The set for the SCIL instruction *add* includes 3 microinstructions.

```
0100000001010000110000        -- MAR=SS=SS-1
0000110000000101010000        -- TPR=TOS; rd
0010101000101000010110        -- MDR=TOS=MDR+TPR; Wr
```

Example 11: Set of microinstructions for SCIL instruction *add*

Currently, one microinstruction for the SCIL processor needs 22 bits. The microinstruction consists of five command signal sets, and different command signal sets occupy different fixed places. In Table 4, we present the length and fixed places of the five command signal sets in one microinstruction.

CMD_ALU	CMD_REG	CMD_MEM	CMD_A	CMD_B	
5 bits	7 bits	2 bits	4 bits	4 bits	= 22 bits

Table 4: Length and fixed places of command signals sets

These different command signal sets are responsible to control different parts of the SCIL processor. The command signal set CMD_ALU (5 bits) is to control the actions of the ALU; the command signal set CMD_REG (7 bits) works as to update the registers; the command signal set CMD_MEM (2 bits) is to communicate with the data memory; the command signals set CMD_A (4 bits) and CMD_B (4 bits) are to choose the data resource of BUS A and BUS B. In the Example 12, we present two microinstructions.

```
(a) MAR=0x01                     "1001000000010000000000"
(b) MDR=TOS+TPR; Rd              "0010000000100101010110"
```

Example 12: Two microinstructions (a) and (b)

For the microinstruction (a), the task is to update the register MAR with 0x01. We split this microinstruction into five command signal sets, and state what these command signal sets represent.

```
10010    0000001    00    0000    0000
  |         |        |      |       |
  |         |        |      |       =>   no data for BUS B
  |         |        |      =>   no data for BUS A
  |         |        =>   no memory operation
  |         =>   update register MAR with data on BUS C
  =>   ALU outputs 0x01
```

For the microinstruction (b), it refers to an addition operation, a reading memory operation and operations of registers. We split this microinstruction into five command signal sets, and state the meaning of each command signal set.

```
00100    0000010    01    0101    0110
  |         |        |      |       |
  |         |        |      |       => put register TPR's data on BUS B
  |         |        |      =>  put register TOS's data on BUS A
  |         |        =>   read the data memory
  |         =>   update register MDR with data on BUS C
  =>   ALU does addition operation
```

Architecture of SCIL Processor

In Figure 13, we present the detailed architecture of the SCIL processor. Moreover, we also show the signals among the instruction memory, the data memory and the SCIL processor.

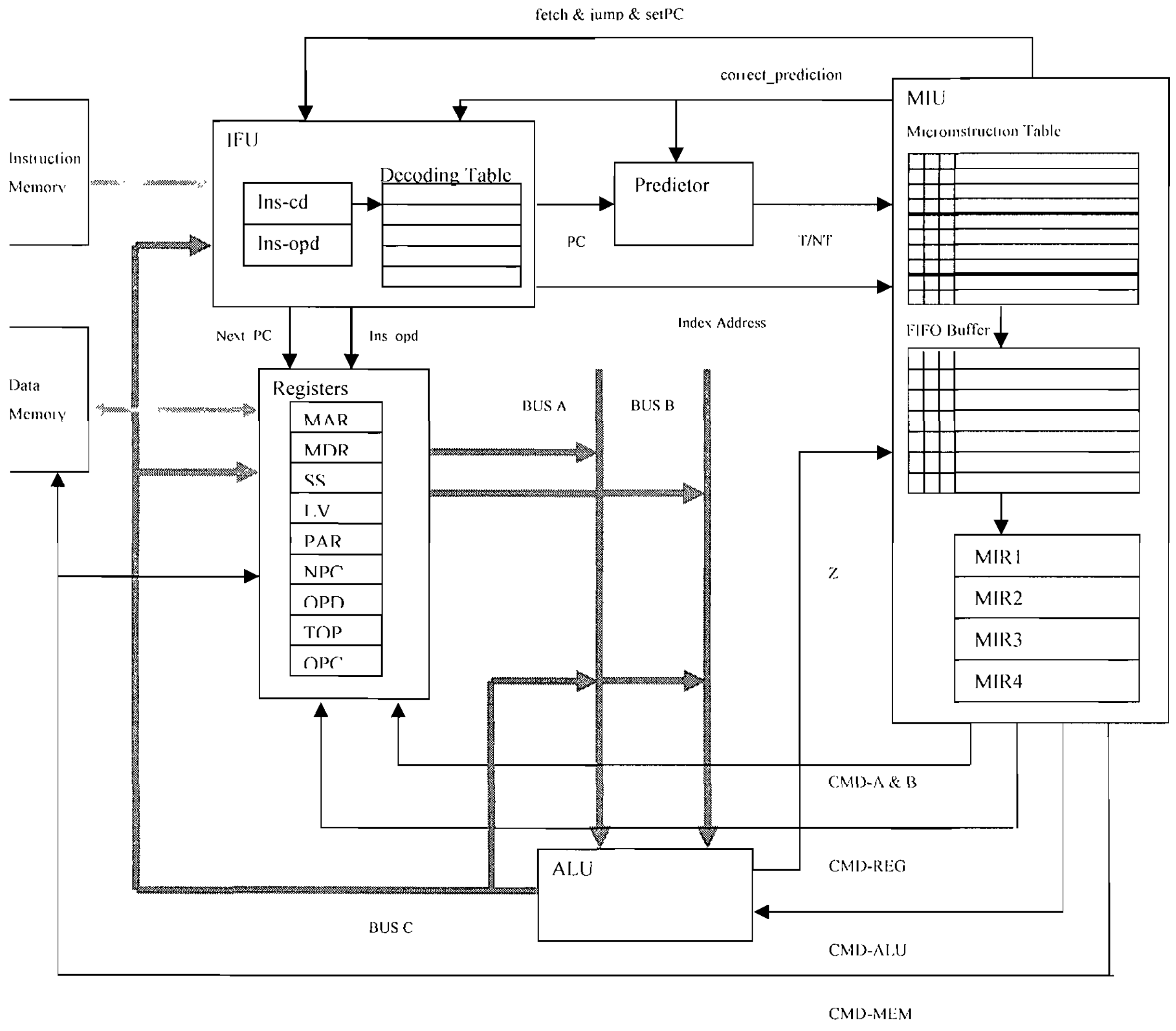


Figure 13: Architecture of SCIL Processor

IFU (Instruction Fetch Unit)

The task of the IFU is to take data from the instruction memory, and then the IFU extracts SCIL instruction codes and instruction operands from the data. After that, the IFU puts instruction code and instruction operand into two particular registers. Finally, the IFU decodes SCIL instruction codes to index addresses. With the index address, the processor can find out the corresponding set of microinstructions in the unit MIU for every SCIL instruction.

Architecture of IFU

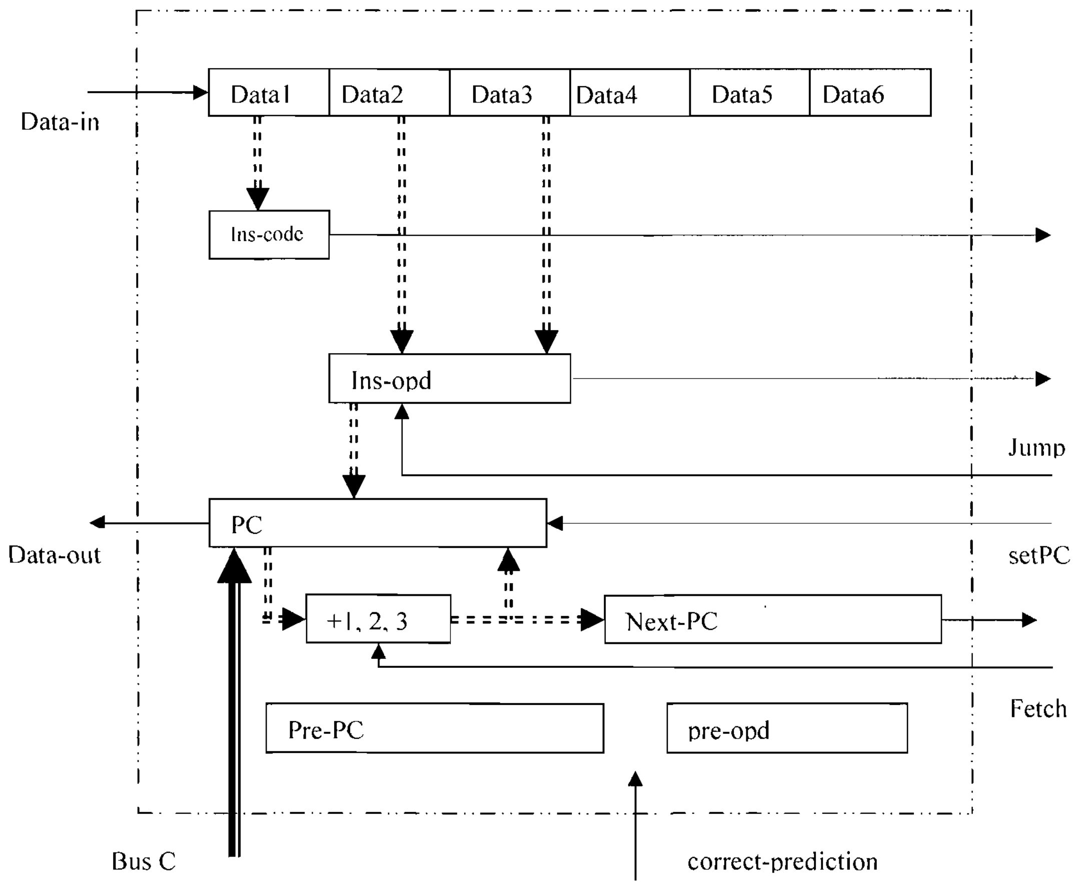


Figure 14: Architecture of IFU.

In Figure 14, we present the architecture of the unit IFU. The register *PC* (Program Counter) is 16 bits. The register *Next_PC* (16 bits) conserves the memory address of the instruction that is next to the current PC. We use value of the register *Next_PC* as the return address when the processor finishes invoking a procedure. The register *Ins_code* (8 bits) is responsible to store the instruction code, while the register *Ins_opd* (16 bits) is used to conserve the corresponding instruction operand. In addition, there are six registers, named as *Data1*, *Data2*, *Data3*, *Data4*, *Data5* and *Data6* (8 bits for each). These six registers respectively conserve the data of instruction memory with the address PC, PC+1, PC+2, PC+3, PC+4, PC+5. These six registers work as a data buffer. The IFU puts data taken from the instruction memory into these registers firstly. When the IFU fetches the instruction code and instruction operand, the IFU uses the data in the six registers instead of reading data from the memory. The IFU can receive three kinds of command signals: *fetch*, *jump* and *setPC*. The *fetch* command asks the IFU to take the SCIL instruction that is next to the current PC; the *jump* command informs IFU there is a branch; and the *setPC* command asks the IFU to update the value of PC with the data on BUS C.

Furthermore, inside the IFU, there is another two registers: *pre_PC* and *pre_opd*. These two registers are used for the prediction. The *pre_PC* conserves the old value of PC when the value of PC changes; similarly, the *pre_opd* conserves the old value of the *Ins_opd* when the value of the *Ins_opd* changes. In addition, the IFU receives a signal named *correct-prediction* which is generated by the MIU. As same as the two registers *pre_PC* and *pre_opd*, this signal is used for the prediction. This signal states whether the previous prediction is correct or not. When it is correct, the value of the signal is '0'; otherwise, the value is '1'. (We will introduce the branch prediction and the predictor in the following sections.)

Fetch SCIL instructions

The IFU takes SCIL instruction codes and the corresponding instruction operands from the six data registers. As we introduced, all of SCIL instructions code occupies 8 bits. Therefore, the IFU always uses the 8 bits of the register *Data1* as the instruction code.

Because the length of instruction operand is various (three types SCIL instructions), the IFU has to determine the instruction type before it takes correct length bits as the instruction operand. The IFU identifies the type of one SCIL instruction with checking the first two bits of instruction code. When the instruction is a Type 1 SCIL instruction, the IFU does not update the value of the register because there are no instruction operands for Type 1 SCIL instructions. When the instruction is a Type 2 SCIL instruction, the IFU uses the 8 bits data of the register *Data2* as the instruction operand. When the instruction is a Type 3 SCIL instruction, the IFU uses 8 bits of the register *Data2* and 8 bits of register *Data3* as the instruction operand. As a result, because the instruction operand was conserved in the register *Ins_opd*, the processor can directly use the instruction operand in the register without caring about the length of it when the processor deals with the SCIL instruction. It is not necessary for the SCIL processor to use additional command to require the IFU to take some bits data as instruction operands any more.

After the IFU updates the registers *Ins_code* and *Ins_opd*, the IFU shifts the six data register, and use the unused data to replace the used data. For example, when the IFU used the data of first three data register *Data1*, the register *Data2* and the register *Data3*, the IFU copies the value of the register *Data4* to the register *Data1*, the register *Data5* to the register *Data2*, as well as the register *Data6* to the register *Data3*. After that, the IFU checks whether there are enough valid (unused) data for the next instruction. Because the Type 3 SCIL instruction, which is the longest instruction among three type instructions, needs 24 bits, the IFU needs at least three data registers with valid data for the next instruction. Otherwise, the IFU reads 32 bits data from the instruction memory to refill the data registers when the number of unused registers is less than 3. In Figure 15, we present the state machine that describes how the IFU operates the six data registers and when the IFU reads new data.

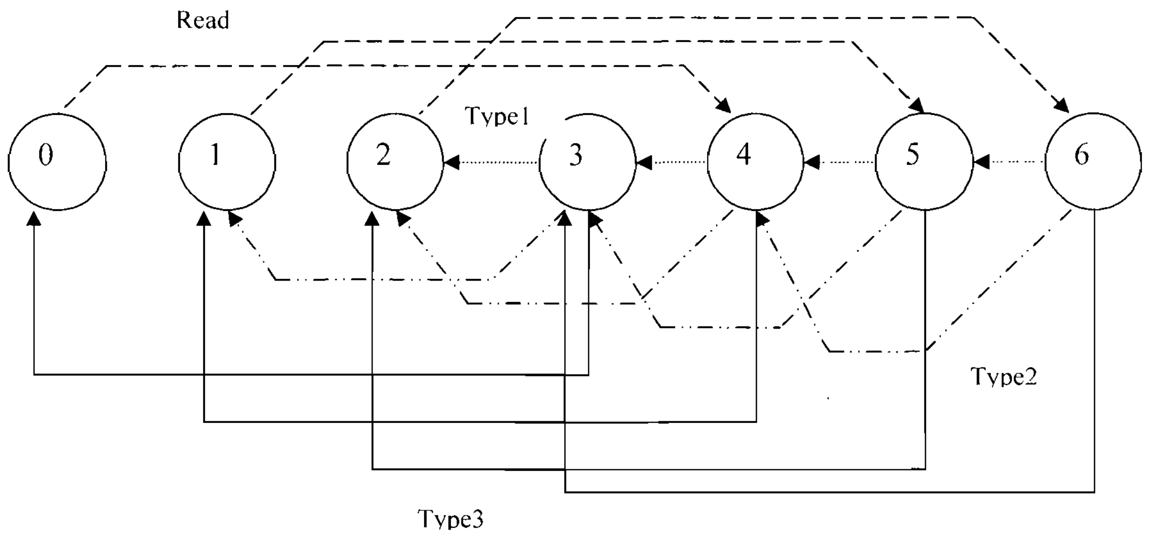


Figure 15: State machine for six data registers

When all the data registers are filled with valid data, the size of buffer (number in the cycle) is 6. Every times, when the IFU gets a Type 1 SCIL instruction, the size of buffer reduces 1. For example, it supposes that the size of buffer is 5. When the IFU gets an instruction *add*, which is a Type 1 SCIL instruction, the size of buffer becomes 4. Similarly, the size of buffer subtracts 2 or subtracts 3 when the IFU get a Type 2 SCIL instruction or a Type 3 SCIL instruction respectively. The IFU reads new data from the instruction memory when the size of buffer is smaller than 3. After the IFU reads the memory, it increases the size of buffer with 4.

In fact, the IFU can remove the six data registers. At this time, we make the IFU suspend when it finishes fetching the first instruction, and restart to work when the processor asks for the second instruction. However, as we introduced in the section for microinstructions, one SCIL instruction corresponds to more than one microinstruction, and each microinstruction at least needs one cycle to execute. There is hence an interval (several cycles) between the processor asking the IFU for the first instruction and the processor asking the IFU for the second instruction. In our design, the IFU uses this interval to prepare the new instruction in advance. During the interval, the IFU checks whether there is enough valid data for the next instruction. When there is not enough valid data, the IFU takes data from the memory automatically. Then when the IFU receives the new command, the IFU already prepared the data for the new instruction. (It supposes that there is no branch.) As a result, the IFU now has a simple pre-fetch function to accelerate fetching instructions. The IFU can take the next instruction from memory before the processor requires a new instruction. This pre-fetch function improves the performance of the processor significantly. Certainly, when the processor meets a branch, the pre-fetch function does not work at all. At this time, the IFU clears up all data registers and reads the new data from the instruction memory. (The method of fetching data is similar to the method the IJVM processor uses.)

Actions of IFU

The actions of the IFU are controlled under the command signals generated by the unit MIU. (We will introduce the MIU in following section.) The IFU has three basic actions corresponding to three commands. When the IFU receives a command *fetch*, it means the next instruction sequences with the current instruction in memory. Therefore, the IFU increases the value of *PC* depending on the type of the current instruction. For example, when the current SCIL instruction is a Type 2 instruction, the IFU increases the value of *PC* with 2. After that, the IFU can fetch the new instruction code and instruction operand from the data registers due to the pre-fetch function. Then the IFU updates the register *Ins_code* and *Ins_opd*, and the IFU updates the value of the register *Next_PC* according to the new value of the *PC*. Finally, the IFU shifts the six data registers and checks whether it needs to read new data from the instruction memory.

When the IFU receives a command *jump*, it means that there is a branch, and the current instruction is a Type 3 instruction whose instruction operand represents the branch address. The IFU needs to update the value of *PC* with this branch address. Because the operand is conserved in the register *Ins_opd*, the IFU copies the 16 bits of the register *Ins_opd* to the register *PC*. Moreover, because the pre-fetch function dose not work for branches, the IFU reads new data from the instruction memory to refill the data registers. After that, the IFU fetches the new instruction code and instruction operand, and updates the registers *Ins_code*, *Ins_opd* and *Next_PC*. Finally, the IFU shifts the six data registers and check whether the IFU needs to read new data from the memory again.

When the IFU receives a command *setPC*, it means that the IFU needs to update the value of PC with the data on BUS C. For example, when the processor executes an instruction *call*, the processor pushes (writes) the value of the register *Next_PC* to the stack (the data memory). When the processor executes the corresponding return instruction, the processor pops (reads) the old *Next_PC* value from the stack (the data memory), and then it places this value on BUS C. At this time, the processor sends a command *setPC* to the IFU. When the IFU receives a command *setPC*, the IFU uses the data on BUS C to update the register *PC*. Then the IFU does the same operations as it receives a command *jump*.

However, when the signal *correct-prediction* states the previous prediction is incorrect, the actions *fetch* and *jump* have a little difference. At this time, the IFU does correct previous wrong prediction. When the IFU receives a command *fetch*, it means that the incorrect prediction is "Take", and now the processor needs to do "Not Take". The IFU firstly picks out the value of the register *pre_PC*. This register conserves the PC of the branch instruction for which the previous prediction is done. By using this old PC value, the IFU can figure out the memory address of the instruction, which is next to the branch instruction in memory. After that, the IFU uses this calculation result as the new PC. When the IFU receives a command *jump*, it means that the incorrect prediction is "Not Take", and the processor needs to do "Take". The IFU uses the value of the register *pre_opd*. This register recodes the instruction operand of the instruction for which the previous prediction is done. The value of this register is exactly the target branch address. Hence the IFU uses it to update the value of PC. When the IFU finished resetting the new PC, the IFU clears up the data registers, and begins to do the normal operations.

Decode SCIL instruction code

Besides fetching the data, the IFU is responsible to map the SCIL instruction codes to the corresponding sets of microinstructions. Inside the IFU, there exists a mapping table, named as Decoding Table. For each SCIL instruction code, the Decoding Table records an entrance address. Furthermore, inside the unit MIU, there exists another table, named as Microinstruction Table. The Microinstruction Table conserves all sets of microinstructions for SCIL instructions, and every SCIL instruction has one and only one set of microinstructions in the Microinstruction Table. When the IFU gets a new SCIL instruction code, the IFU sends it to the Decoding Table. The table returns a map index, which we call as *index-address*. Every index-address represents an address for the Microinstructions Table, and it points to the first microinstruction in one set of microinstructions. In Figure 16, we present the relationship between the Decoding Table and the Microinstruction Table.

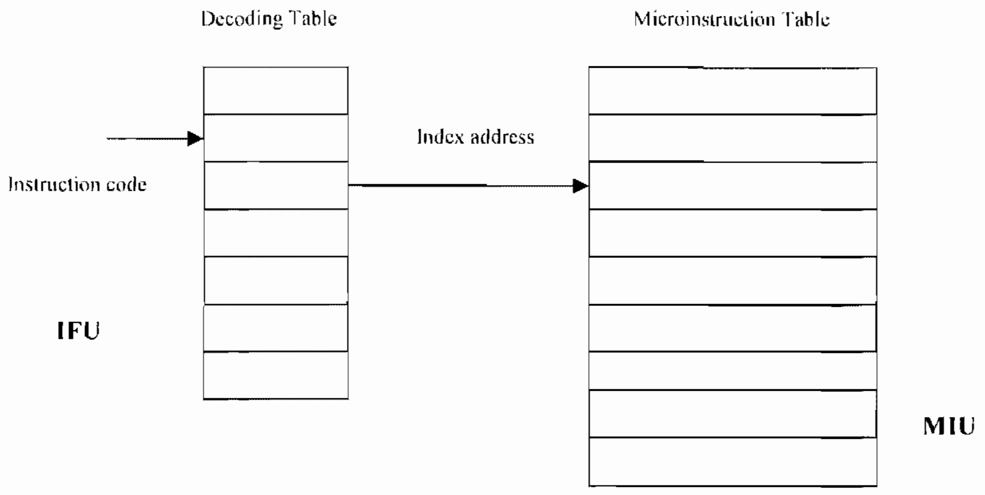
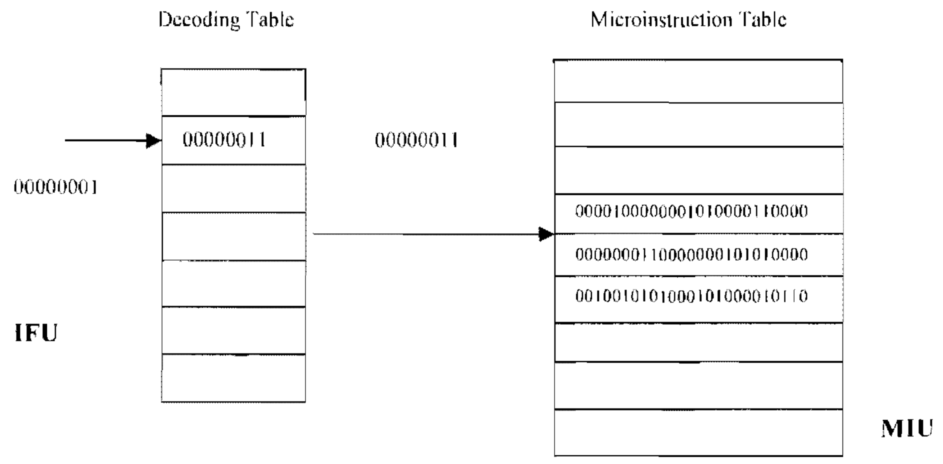


Figure 16: Relationship between two tables.

In Example 13, we show the process of mapping the SCIL instruction *add* to its corresponding set of microinstructions.



Example 13: Map instruction *add* to the set of microinstructions

The instruction code of *add* is "00000001". The IFU uses "00000001" as the address to search in the Decoding Table. Then the IFU uses the result "00000011 as the index-address and outputs it to the MIU. When the MIU receives this index address, it uses "00000011" as the address to search in the Microinstruction Table. As a result, the MIU moves the internal pointer to the fourth element, which stores the first microinstruction in the set of microinstruction for the instruction *add*. The first microinstruction for the instruction *add* is "00001000000001010000110000".

Compare with IJVM processor

Tanenbaum's IJVM processor also uses a unit named IFU to fetch IJVM instructions from memory. Furthermore there are some similar mechanisms between the IFU of the SCIL processor and the IFU of the IJVM processor. For example, both of them use data registers as data buffer, and have similar state machines for management of data registers. However, we give the IFU of SCIL processor some new functions, which the IJVM processor does not have. Firstly, the IFU of SCIL processor can automatically fetch instruction operands while it obtains instruction codes. For the IJVM processor, the processor needs to send a particular command to the IFU in order to fetch instruction operands. Therefore, for one IJVM instruction, the IJVM processor sends two times of commands to the IFU, and the IFU does fetching jobs twice. The IJVM processor sends the first command to ask IFU to output instruction codes; and it sends the second command to ask the IFU for the instruction operands. In our work, we avoid the second time of asking for the IFU. The SCIL processor can obtain both the instruction code and its instruction operand from the IFU at only one time. Therefore the SCIL processor uses less times of taking data than the time the IJVM processor needs. Secondly, because the SCIL processor has the prediction function, the IFU of the SCIL processor becomes more complex than the IFU of the IJVM processor. We added some new registers and signals to implement the prediction function. For example, the IFU have the registers *pre_PC* and *pre_opd*, which are to conserve the state before predictions. Thirdly, the IFU of SCIL processor is responsible to decode instruction code. For the IJVM processor, another unit does this job. As we introduced, the SCIL processor combines two pipeline stages into one stage. Therefore, the IFU has two pieces of works at the same time.

MIU (Microinstruction Unit)

The unit MIU is another principal unit for the whole processor. First of all, the MIU has the responsibility to control the actions of the unit IFU; secondly, the MIU is to search the set of microinstructions in the Microinstruction Table, and then the MIU arranges microinstructions in certain order; thirdly, based on these microinstructions, the MIU is enable to generate kinds of command signals, which control the operations of the SCIL processor.

Control IFU

As we introduced in the previous sections, the IFU can receive three different commands, and the MIU is responsible to send these commands. The IFU generates the commands based on the information conserved in the Microinstruction Table. Each line of the Microinstruction Table consists of one microinstruction and three flag bits: *npc*, *jmp* and *end*. These three flag bits represent the state of the current microinstruction. With acquiring the state of microinstructions, the MIU decides to send which command to the IFU. In

Table 5, we show one line of the Microinstruction Table. Each line of the Microinstruction Table has 25 bits.

npc	jmp	End	Microinstruction
1 bit	1 bit	1 bit	22 bits

Table 5: One line of Microinstruction Table

The flag bit *end* represents whether the current microinstruction is the last microinstruction in the set of microinstructions. Because all sets of microinstructions are limit (2~9 items for each set) and the processor executes these microinstructions sequentially, the last line in one set of microinstructions always has an active flag bit *end*. In Example 14, we show the set of microinstructions for SCIL instruction *add*. The first three bits are flag bits, and the flag bit *end* of third line is '1'. It means that the third microinstruction is the last microinstruction in this set of microinstructions.

```
   0000100000000101000011000    -- MAR=SS=SS-1
   0000000110000000101010000    -- TPR=TOS; rd
   0010010101000101000010110    -- MDR=TOS=MDR+TPR; wr
    |||
    /\\
npc jmp end
```

Example 14: Microinstruction set for *add* in Microinstruction Table

The flag bit *jmp* is active when the set of microinstructions correspond to one condition branch instructions, such as instructions *blt*, *beq* and *ble*. Inside such a set of microinstructions, one flag bit *jmp* is active. In Example 15, we show the set of microinstructions for the SCIL instruction *blt*. We can see that there are two active flag bits in the fifth line. The first is the flag bit *end* because the fifth microinstruction is the last microinstruction in this set of microinstructions. The second is the flag bit *jmp* because the current microinstruction refers to a condition branch. (The notation "JMP(Z)" represents that the processor takes the branch when Z is true.)

```
   0000100000000101000011000    -- MAR=SS=SS-1
   0000100000000101010011000    -- MAR=SS=SS-1; rd
```

```
    000000011000000001000010000        -- TPR=MDR; rd
    000100000000000000001100101        -- Z=TPR cmp TOS
    011000010100000000000010000        -- TOS=MDR; JMP(Z)
    | | |
    / \ \
npc jmp end
```

Example 15: Microinstruction set for *blt*

However, in practice, we use two active flag bits *jmp* for a condition branch in the Microinstruction Table. For example, there are two active flag bits *jmp* in the set of microinstructions for the instruction *blt.* We show this set of microinstructions in Example 16. The flag bit *jmp* in the first line is active. We adopt such a method because we hope the processor can know the state of microinstructions as early as possible. Earlier the processor knows the current instruction referring to a condition branch, earlier the processor begins to do the prediction for this condition branch and fetch the new instruction. It is useful for the SCIL processor to reduce the suspend cycles.

```
    010010000000101000011 0000        -- MAR=SS=SS-1
    000010000000101010011 0000        -- MAR=SS=SS-1; rd
    000000011000000001000010000        -- TPR=MDR; rd
    000100000000000000001100101        -- Z=TPR cmp TOS
    011000010100000000000010000        -- TOS=MDR; JMP(Z)
    | | |
    / \ \
npc jmp end
```

Example 16: Microinstruction set for *blt* in Microinstruction Table

The flag bit *npc* is active when this microinstruction asks the processor to update the value of PC with the data on BUS C. As same as the flag bit *jmp*, the active flag bit *npc* appears two times in one set of microinstructions. In Example 17, we show the set of microinstructions for the SCIL instruction *ret*. The flag bit *end* and the flag bit *npc* in the last line are active, and the flag bit *npc* of the first line is active too.

```
    100000001000000010001000000        -- MAR=LV
    000010000000101010101000000        -- MAR=SS=LV-1; rd
    000000010001000010001 0000        -- LV=MDR; rd
    000000011000000000000010000        -- TPR=MDR
```

```
  000010000000010100001 10000       -- MAR=SS=SS-1
  0000100100001010110000000         -- MAR=SS=PAR; rd
  0000000100100000000010000         -- PAR=MDR
  0000000100000101001010000         -- MDR=TOS; wr
  1010000100000000001100000         -- C=TPR
  |||
  /\\
npc jmp end
```

Example 17: Microinstruction set for *ret* in Microinstruction Table

By checking three flag bits, the MIU can acquire the state of microinstructions and then send three different commands, *fetch*, *jump* and *setPC*, to the IFU. When the MIU deals with a new set of microinstructions, firstly the MIU reads the two flag bit *jmp* and *npc* of the first line in the set. When both of them are not active, it means that there is no branch for this set of microinstructions. At this time, the MIU sends a command *fetch* to the IFU. Otherwise, when the flag bit *npc* of the first line is active, the MIU does not send any command to the IFU immediately. Instead the MIU sequentially executes the microinstructions one by one. When the MIU meets the second active flag bit *npc*, the MIU sends a command *setPC* to the IFU. When the flag bit *jmp* of first line is active, the MIU checks the prediction result made by the predictor and then sends commands. The MIU sends a command *fetch* to the IFU when the prediction is "Not Take". Otherwise, when the predictions result is "Take", the MIU sends a command *jump* to the IFU. After that, the IFU sequentially executes the microinstructions. When the MIU meets the second active flag bit *jmp*, the MIU reads the value of the condition result Z to check whether the previous prediction is correct. When the prediction is correct, it is all right and there is nothing happened. Otherwise, the MIU sends a new correct command to the IFU to replace the wrong one. At this time, the MIU sets the value of the signal *correct-prediction* to inform the IFU and the predictor whether the prediction is correct. (We will introduce how the predictor works in following sections.)

Tanenbaum's IJVM processor uses two flag bits while the SCIL processor uses three flag bits. Both of the two processors have the flag bit *end*. However, the SCIL processor uses two flag bits *jmp* and *npc* to represent branches which the IJVM processor uses one flag bit *jmp* to represent. For the IJVM processor, it obtains branch addresses only from the data bus.

However, there are two methods to obtain branch addresses for the SCIL processor. Firstly, the processor can obtain the branch addresses from the data bus. The bit flag *npc* is used to represent this situation. Secondly, the processor can use the value of register *Ins_opd* inside the IFU as the branch addresses. (In previous sections, we introduced that the branch addresses as instruction operands are conserved in this register.) The bit flag *jmp* is used to represent the second situation. As we can image, it takes less time for the processor to use the existing data in the register than to wait for the data on data bus. Therefore, we divided all branches into two different operations. In this way, although our SCIL processor uses one more flag bit than the IJVM processor uses, the SCIL processor can accelerate the execution of branch operations.

Data Dependency

After the MIU finding out the set of microinstructions in the Microinstruction Table based on the index-address, the processor begins to execute the microinstructions one by one. Theoretically, the processor can execute one microinstruction in one cycle because the SCIL processor uses single pipeline construction. However, it is quite easy to generate data dependency among microinstructions. The data dependency may result in one or more than one suspend cycles for some microinstructions. Although we can optimize the microinstructions to reduce data dependency, it is impossible to avoid all data dependency because the permutation of the different SCIL instructions is too complex to anticipate all possibilities. In fact, some data dependency is inevitable.

In Example 18, we show the set of microinstructions for the SCIL instruction *add*. The second microinstruction changes the value of the register TPR; and in the third microinstruction, the value of the TPR is used as one operand of the addition operation. Hence the third microinstruction is data dependency for the second microinstruction. The processor cannot execute the third microinstruction until the processor completes executing the second microinstruction.

```
MAR=SS=SS-1
TPR=TOS; rd
```

MDR=TOS=MDR+TPR; Wr

Example 18: Data dependency

Data dependency makes the processor generate suspending cycles, and too many suspend cycles affect the execution speed of the processor. Therefore, in order to minimize the influence of data dependency, we use the forwarding technique and a FIFO buffer in our processor. The forwarding technique can reduce the number of suspend cycles directly. The FIFO buffer can make the processor keep doing some works even when the processor is in suspend cycles.

Forwarding

The MIU uses forwarding technique when it checks data dependency for microinstructions. For example, the processing microinstruction needs the value of a certain register. However this register now is waiting for being update because one previous microinstruction referred to this register. At this time, without the forwarding, the processor cannot begin to execute this processing microinstruction until the processor finishes executing the previous microinstruction and updating that register. By contraries, with the forwarding, the processor can execute the processing microinstruction several cycles early.

In Example 19, we present an example. In (a), we show two sequential microinstructions. In (b), we present the execution of the two microinstructions without the forwarding technique. The MIU finishes executing the first microinstruction at the i+2nd cycle, and it begins to execute the second microinstruction at the i+3rd cycle. The whole execution time is 6 cycles. In (c), we present the execution of the two microinstructions with the forwarding technique. The MIU finishes executing the first microinstruction still at the i+2nd cycle. However, the MIU can execute the second microinstruction at i+2nd cycle as soon as the ALU finishes the addition operation. During the i+2nd cycle, the result of the ALU is sent to two objects at the same time. The first receiver is the register TOS. The register TOS uses the result to update its value. Another receiver is the ALU. The ALU uses the result as the operand of an addition operation, which is done by the second microinstruction. As a result, the whole execution time becomes 5 cycles, which is one cycle less than without forwarding.

(a) Two microinstructions

TOS=SS+TPR
MDR=TOS+MDR

(b) Without forwarding

Cycle	TOS=SS+TPR	MDR=TOS+MDR
i	load SS, TPR	
i+1	do addition operation	
i+2	update TOS	
i+3		load TOS, MDR
i+4		do addition operation
i+5		update MDR

(c) With forwarding

Cycle	instr: TOS=SS+TPR	instr: MDR=TOS+MDR
i	load SS, TPR	
i+1	do addition operation	
i+2	update TOS	load MDR, use result of ALU as the value of TOS
i+3		do addition operation
i+4		update MDR

Example 19: With forwarding and without forwarding

FIFO Buffer

The FIFO buffer is inside the MIU. Now the length of the FIFO buffer is 18, and each element of FIFO buffer is 25 bits, as same as the element of the Microinstruction Table. The FIFO buffer cannot reduce the number of suspending cycles directly as the forwarding technique does. Instead, this FIFO buffer helps the processor use these suspending cycles to continue part of works.

Without FIFO buffer, when there are suspending cycles for the current microinstruction, the MIU suspends completely. The MIU can neither read new microinstruction nor send a new command to the IFU. In next cycle, the MIU deal with the same microinstruction. With the FIFO buffer, when the MIU meet suspending cycles, the MIU copies the current microinstruction to the FIFO buffer. In next cycle, the MIU deal with the microinstruction existed in the FIFO buffer. Furthermore, now the MIU can pick out a new microinstruction from the Microinstruction Table. Therefore, even through the processor is during the suspending cycles, the MIU can continue sending commands to the IFU based on the new microinstruction. As a result, the IFU can receive the commands several cycles early, and correspondingly the IFU can finish its operations relatively early. It is very important for the IFU when there is a branch. Then after the MIU finishes treating the current set of microinstructions, the MIU can execute the new set of microinstructions immediately because the IFU completed the task of new commands and finished preparing the needed data. Therefore, the MIU can avoid new suspending cycles caused by waiting for the IFU completing its operations. (A FIFO buffer is also used in Tanenbaum's IJVM processor.)

Treating microinstructions

The MIU takes microinstructions from the Microinstruction Table, checks the data dependency, and then converts microinstructions to command signal sets. We call this process as *treating microinstructions*.

The MIU uses an internal pointer to indicate the position in the Microinstruction Table. When the MIU receives a new index-address from the IFU, the MIU move this pointer to the first line in the corresponding set of microinstructions. Each cycle, the MIU picks out the pointed microinstruction as the *processing microinstruction*, which is to be treated in the current cycle. Then the MIU moves the pointer to the succeeding line till the last line in this set of microinstructions.

When the MIU picks out the processing microinstruction from the Microinstruction Table, there are three possibilities for the MIU to deal with this processing microinstruction during the current cycle. Firstly, when the FIFO buffer is empty, the MIU check whether there is data dependency for the processing microinstruction. The MIU can convert this microinstruction to command signal sets when there is no data dependency. Otherwise, the MIU copies the processing microinstruction to the FIFO buffer. At this time, there is no microinstruction converted to command signals in the current cycle. Secondly, when the FIFO buffer is neither empty nor full, the MIU copies the processing microinstruction to the FIFO buffer. Then the MIU checks whether there is data dependency for the first microinstruction in the FIFO buffer. When there is no data dependency, the MIU uses the microinstruction in FIFO buffer as the new processing microinstruction, and converts it to command signal sets. Otherwise there is no microinstruction converted to command signals in the current cycle. Thirdly, when the FIFO buffer is full, the MIU check whether there is data dependency for the first microinstruction in the FIFO buffer. When there is no data dependency, the MIU picks out this microinstruction and copies the old processing microinstruction to the FIFO buffer. Otherwise, the MIU suspends working in the current cycle when there is data dependency for the first microinstruction in the FIFO buffer. In the next cycle, the MIU still use the same microinstruction as the processing microinstruction and do the same operation as what the MIU does in the current cycle.

When the MIU converts one microinstruction to command signal sets, the MIU splits this microinstruction into four parts and generate all command signal sets. There are four Microinstruction Registers (MIR1, MIR2, MIR3 and MIR4) inside the MIU. These registers store the command signals to be outputted. The MIU outputs the context of the register MIR1 as the command signals in the current cycle. Similarly, The MIU outputs the context of the MIR2, the MIR3 and the MIR4 in the second, third and fourth cycle respectively. Furthermore, after the MIU outputs the context of the MIR1, the MIU shifts contexts of the MIR registers with one position. The MIU copies the MIR2 to the MIR1, the MIR3 to the MIR2, as well as the MIR4 to the MIR3.

Each MIR register has 22 bits, as same as one microinstruction. When the MIU outputs the context of the register MIR1, the bits on the fixed places are outputted as the corresponding command signals. As we introduced, one microinstruction consists of five command signal sets. The MIU splits the microinstruction into four parts. (We make CMD_A and CMD_B as one part). The MIU puts these four parts to the corresponding fixed places of the four MIR registers. The MIU copies the part1 (CMD_A and CMD_B) to the MIR1, the part2 (CMD_ALU) to the MIR2, the part3 (CMD_REG) to the MIR3 as well as the part4 (CMD_MEM) to the MIR4. In this way, the MIU outputs the context of one microinstruction in four cycles, and only one part is outputted in one cycle. The MIU can output command signal sets in form of pipeline.

In Example 20, we present how the MIU converts one microinstruction to command signals. It supposes that there is no data dependency for this microinstruction.

(a) One microinstruction

```
001010100010100001011O        -- MDR=TOS=MDR+TPR; Wr
```

(b) Spilt the (a) into four parts:

```
00101   0100010   10      00010110
Part2   Part3     Part4   Part1
```

(c) Copy the (B) to the four MIR registers

```
MIR1: 0000000000000000010110
MIR2: 0010100000000000000000
MIR3: 0000001000100000000000
MIR4: 0000000000001000000000
```

(d) Output command signals

	Cycle 1	Cycle 2	Cycle 3	Cycle 4
CMD_A&B:	00010110	00000000	00000000	00000000
CMD_ALU:	00000	00101	00000	00000
CMD_REG:	0000000	0000000	0100010	0000000
CMD_MEM:	00	00	00	10

Example 20: Convert one microinstruction to command signals

In (a), this is one microinstruction that the MIU picked out from the Microinstruction Table. In (b), the MIU splits this microinstruction into four parts. In (c), the MIU copies these four parts to the four MIR registers. In (d), we present the command signals that the MIU outputs in successive four cycles.

ALU

The independent execution ALU can do the basic calculation of 32 bits integer. It does not support float-point operations. The ALU receives the values on BUS A and BUS B as two operands, and then does one operation according to the 5 bits command signal set CMD_ALU. (See Table 6)

CMD_ALU	Operation of ALU	Comment
00000	0	Output 0
00001	A*	Output A
00010	B **	Output B
00011	Not A	Not operation
00100	-A	Negative operation
00101	A + B	Addition operation
00110	A + B + 1	Add A, B and 1
00111	A + 1	Add 1
01000	A – 1	Substruct 1
01001	A – B	Substruct
01010	A and B	And operation
01011	A or B	Or operation

01100	A=B?	Z=1 when A=B;
		Z=0 when A/=B;

01101	A/=B?	Z=1 when A/=B;
		Z=0 when A=B;

01110	A>B?	Z=1 when A>B;
		Z=0 when A<=B;

01111	A>=B?	Z=1 when A>=B;
		Z=0 when A<B;

10000	A<B?	Z=1 when A<B;
		Z=0 when A>=B;

10001	A<=B?	Z=1 when A<=B;
		Z=0 when A>B;

10010	1	Output 1

10011	-1	Output -1
10100	Z='1'	Set signal Z '1'
10101	Z='0'	Set signal Z '0'
10110	SHL16	Shift left 16 bits
10111	SHR8	Shift right 8 bits
11000	SHL1	Shift left 1 bit

*A: data from BUS A;
**B: data from BUS B

Table 6: Command signal CMD_ALU

Besides arithmetic calculations, the ALU is responsible to determine whether the processor takes condition branches or not. Every condition branch instruction, such as *beg*, *blt* and *bge*, contains one comparison. (We use the notation "cmp" to express the comparison in microinstructions) Depending on the result of the comparison, the processor decides to take or not take the branch. When the ALU does a comparison, according to the command signal set *CMD_ALU*, the ALU chooses one kind of comparison operator to compare two operands. Then the ALU use the result of the comparison to update the value of the signal Z. After that, the processor checks the value of the signal Z. When the signal equals to '1', the processor takes the branch; otherwise, the processor does not take the branch. Specially, for the instruction *br*, although it does not need to do the comparison, we still ask the ALU to do an "always true" comparison and update the signal Z as '1'. In this way, the processor uses the same method to deal with all condition branches and avoid adding the new circuitry for the instruction *br*. In Example 21, we present the set of microinstructions for the SCIL instruction *blt*. The fourth microinstruction requires the ALU to do a comparison. After that, in the fifth microinstruction, "JMP(Z)" represents that the processor decides whether it takes this branch or not based on the value of signal Z.

```
MAR=SS=SS-1
MAR=SS=SS-1; Rd
TPR=MDR; Rd
Z=TPR cmp TOS
TOS=MDR; JMP(Z)
```

Example 21: Microinstructions for the SCIL instruction *blt*

Local Memory

Currently the local memory comprises of nine 32-bit registers. These nine registers are named as MAR, MDR, OPD, SS, LV, PAR, NPC, TOS and TPR. Each register has its particular purpose.

The register MAR (Memory Address Register) records the address of data memory while the register MDR (Memory Data Register) stores the data of data memory. These two registers are used to communicate with the data memory. When the processor reads data from the memory, it puts the address into the MAR firstly. Then the MAR sends this address to the data memory, and the return value of the memory is conversed in the MDR. When the processor writes data to the memory, it puts the address into the MAR and puts the data into the MDR. Then the processor sends a signal to make the memory writable, and writes the data to the address in the memory.

The register OPD (OPeranD) is used to conserve the instruction operand of the current SCIL instruction. As we introduced, the IFU is responsible to fetch instruction operands from the instruction memory, and stored them in the register *Ins_opd.* Every cycle, the processor updates the value of the register OPD with the value of the *Ins_opd.*

The register NPC (Next Program Counter) stores the address of the instruction that is next to the current PC. This register corresponds to the register *Next_PC* in the IFU. There are two reasons why we use a register to conserve the Next PC instead of the PC. Firstly, the processor does not need to know the value of the current PC because only the IFU accesses the instruction memory and fetches SCIL instructions. Secondly, the value of the Next PC is necessary for the processor. The processor has to put this value on BUS C when asking the IFU to do a *setPC* operation.

The register SS (Summit of Stack) and the register TOS (Top element Of Stack) are used to describe the memory stack. The SS always points to the summit of the stack in the data memory. The TOS always keeps the value of the top element of the stack. Because there are many operations using the value of the top element, the processor can accelerate these operations when the TOS can keep the correct value of the top element. However, now the processor has to add some microinstructions to do some additional operations in order to keep the correct value in the TOS. It spends a lot of time and hardware resources to sustain the correctness of the TOS. Therefore it is difficult for us to calculate how much time we can win by using the TOS on earth.

The register LV (Local Variable) conserves the address of the first local variable of the current procedure in the stack, while the register PAR (PARameter) stores the address of the first parameter of the procedure in the stack. These two registers help the processor load, store and modify local variables and parameters.

The last register TPR (TemPorary Register) is a temporary register. Usually the processor uses the TPR to conserve some interval value during a series of operations.

Read and Write Registers

The processor cannot read and write the registers at the same time. Reading and writing registers are in two different pipeline stages. Figure 17 shows the connection among registers, three data buses and the data memory.

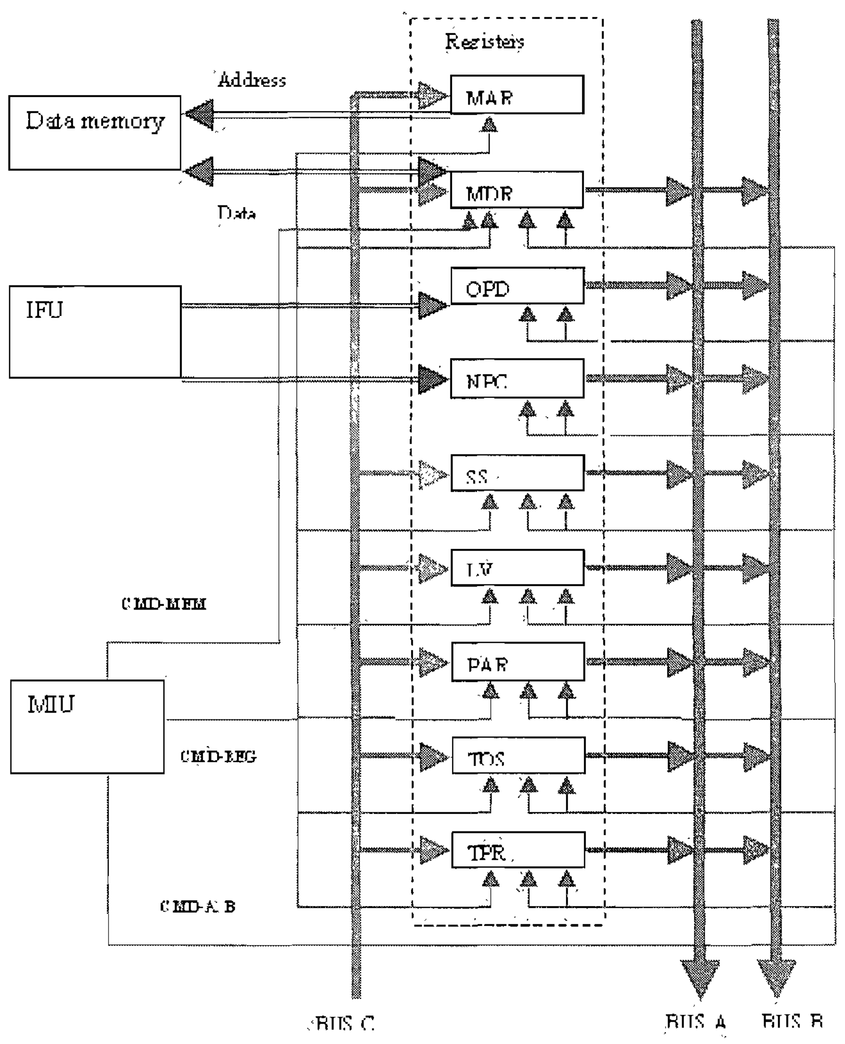


Figure 17: Registers, Data buses and Data Memory

Except for the MAR, the processor can put the value of all registers on BUS A or BUS B, and make them as operands for the ALU. The processor chooses two registers and put their value on BUS A and BUS B respectively according to the 4 bits command signal sets CMD_A (See Table 7) and the 4 bits command signal set CMD_B (See Table 8).

CMD_A	Register	Comment
0000		Clear BUS A
0001	MDR	MDR => BUS A
0010	OPD	OPD => BUS A
0011	SS	SS => BUS A

0100	LV	LV => BUS A
0101	TOS	TOS => BUS A
0110	TPR	TPR => BUS A
0111	NPC	NPC => BUS A
1000	PAR	PAR => BUS A

Table 7: Command signal CMD_A

CMD_B	Register	Comment
0000		Clear BUS B
0001	MDR	MDR => BUS B
0010	OPD	OPD => BUS B
0011	SS	SS => BUS B
0100	LV	LV => BUS B
0101	TOS	TOS => BUS B
0110	TPR	TPR => BUS B
0111	NPC	NPC => BUS B
1000	PAR	PAR => BUS B

Table 8: Command signal CMD_B

The SCIL processor uses three data sources to update registers. The first data source is the IFU. Every cycle, the processor updates the OPD with the value of the register *Ins_opd* and updates the NPC with the value of the register *Next_PC*. The second source is the data on BUS C. The SCIL processor can use the data on BUS C to update one or several registers at one time. The processor can update the MAR, MDR, SS, LV, PAR, TOS and TPR with the data on BUS C according to the 7 bits command signal set CMD_REG (See the Table 9). The third data source is the data memory. The processor takes the data from memory to the register MDR according to the 2 bits command signal CMD_MEM (See Table 10).

The No.i of CMD_REG	Register	Comment
0	MAR	BUS C => MAR
1	MDR	BUS C => MDR
2	SS	BUS C => SS
3	LV	BUS C => LV
4	PAR	BUS C => PAR

5	TOS	BUS C => TOS
6	TPR	BUS C => TPR

Table 9: Command signal CMD_REG

The No.i of CMD_MEM	Action	Comment
0	Read	Read data memory with the address in MAR to MDR
1	Write	Write the data of MDR to data memory with the address in MAR

Table 10: Command signal CMD_MEM

Predictor

The predictor is a one-bit predictor with 128 different addresses. We show the architecture of the predictor in Figure 18.

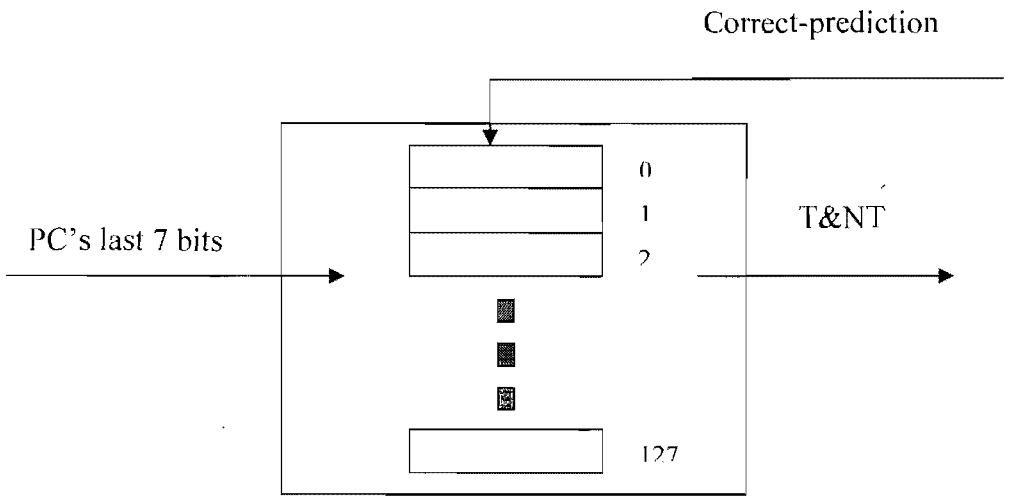


Figure 18: Architecture of one-bit predictor

The predictor receives the last 7 bits of PC as the prediction address. Therefore the prediction address may be same for the different branches when the PCs of these branches have the same last 7 bits. For example, one branch's PC is 129 and another branch's PC is

385. The predictor uses the same branch address 1 for these two branches because the last 7 bits of their PC are "0000001". Inside the predictor, there is an array with 128 elements, and each element is one bit. The predictor uses the inputted 7-bit prediction address as the index of the array, and then it outputs the value of the corresponding element as the prediction result. When the bit is '1', it represents the prediction is "Take"; otherwise, it represents the predictions is "Not Take". The predictor outputs the prediction result via the signal *T&NT*. Moreover the signal *correct-prediction* generated by the MIU informs the predictor whether the prediction is correct. When the prediction is incorrect, the predictor negatives the value of one bit for this prediction address. For example, when the prediction result "Take" for prediction address 4 is incorrect, the predictor will update this prediction element with "Not Take".

When the value of PC changes, the predictor will give a prediction based on the last 7 bits of this PC value whatever the current instruction is a branch instruction or not. When the current instruction is not a branch instruction, the MIU does not respond for this prediction. Therefore, the predictor thinks this prediction is correct and does not change the value of this prediction bit. In this way, we avoid adding the additional circuitry to check whether the instruction is a branch instruction or not.

There are four possible situations for one prediction. T/T: the prediction is "Take" and the processor needs to take the branch; NT/NT: the prediction is "Not Take" and the processor needs not to take the branch; T/NT: the prediction is "Take" and the processor needs not to take the branch; NT/T: the prediction is "Not Take" and the processor needs to take the branch. The T/T and NT/NT are correct predictions. The T/NT and NT/T are incorrect predictions. In our design, there are no penalty cycles for the correct predictions. However, when the predictions are not correct, there are three penalty cycles for the SCIL processor. In Table 11, we show the penalty cycles for these four prediction results. Moreover, we also show the penalty cycles for the processor without a predictor. Without the predictor, the processor has three penalty cycles when it takes the branch, and two penalty cycles when it does not take the branch.

Prediction/Fact	Stalls with predictor	Stalls without predictor
T/T	0	3
NT/NT	0	2
T/NT	3	2
NT/T	3	3

Table 11: Number of stalls caused by branch

Instruction Memory and Data memory

Currently, we use 8 * 1024 bits BRAM as the instruction memory and 32 * 512 bits BRAM as the data memory. Seemingly, the instruction memory is not very big because averagely one SCIL need occupy 16 bits. Therefore this instruction memory can support a SCIL program with about 500 SCIL instructions, and clearly the program with 500 instructions is not a big program. However, because the SCIL branch instructions use 16 bits to represent target branch addresses, we can enlarge the instruction memory to 8 * 65536 bits without change the definition of the SCIL. Moreover, in the current design, the SCIL processor already uses 16 bits data to represent the PC. As a result, we can modify the size of instruction memory with changing the width of signals working as the memory address between the instruction memory and the processor. For example, when the size of the instruction memory is 8*1024 bits, the processor connects the last 10 bits of the 16 bits as the memory address to the instruction memory; and when the size of the instruction memory becomes 8*65536 bits, the processor connects all 16 bits as the address to the instruction memory. When the size of the instruction memory is 8 * 65536 bits, it can contain a SCIL program with about 30000 instructions, and it is enough for most of embedded system designs. Furthermore, because the SCIL processor uses 32 bits data to conserve the memory address for the data memory, the size of the data memory can also be changed in some range.

The SCIL processor needs to modify the width of signals connecting to these two memories when the sizes of the instruction memory or the data memory change. As a result, the SCIL processor, as a softcore processor, can change the hardware resource usages with different memory configurations. At present, we do these changes by modifying the generics in VHDL source code directly. It is not very convenient for the users who do not know well VHDL to implement memory configurations. Therefore it is possible for us to develop a GUI (Graphical User Interface) Wizard to facilitate memory configurations in the future.

Chapter 5 Experiments

For the purpose of prototyping, we target our SCIL processor for a Xilinx's Virtex II PRO FPGA. Moreover in order to compare the performance of the SCIL processor with other existing softcore processors, we create a MicroBlaze system whose construction is similar to the SCIL processor system. Both of two systems consist of one processor, one instruction memory and one data memory. We respectively run four benchmarks on two systems, and compare the number of cycles to execute programs on the different processors.

Design Flow

We use VHDL as programming language to code the processor entry, and use Xilinx ISE 8.2i as the development environment. The FPGA we used is Xilinx's Virtex II PRO on the platform AP1000. The functional simulation tool is ModelSim 6.2g. In addition, in order to observe internal signals and BRAM results on FPGA, we use Xilinx ChipScope pro 8.2 [25] to implement monitor signals on FPGA. Xilinx ChipScope Pro Core Inserter can insert logic analyzer, bus analyzer and virtual I/O low-profile cores directly into the design, and these captured signals can be analyzed through Xilinx ChipScop Pro Analyzer.

In Figure 19, we present the basic design flow of the SCIL processor. This design flow refers to the Xilinx ISE 8.2 design flow [26]. First of all, under Xilinx editor, we use VHDL to create the entities of the SCIL processor. At this time, we do the functional simulation with ModelSim tool to verify the correctness of our design. After that, we use the Xilinx Synthesis Technology (XST) GUI to synthesize the VHDL files into NGC files. Then we use ChipScope Pro Core Inserter to add monitor signals into the processor design. In design implementation step, we convert the logical design file format in order to fit the design with AP1000 platform. The physical information about Virtex II PRO FPGA is contained in the native circuit description (NCD) file and the information about CPLDs is in VM6 file. Then we generate a bitstream file for our device depending on these files. Finally, we use iMPACT to load the bitstream file to FPGA on AP1000 via Xilinx download cable. After that, we check the result of program and the values of monitor signals by using ChipScop Pro Analyzer.

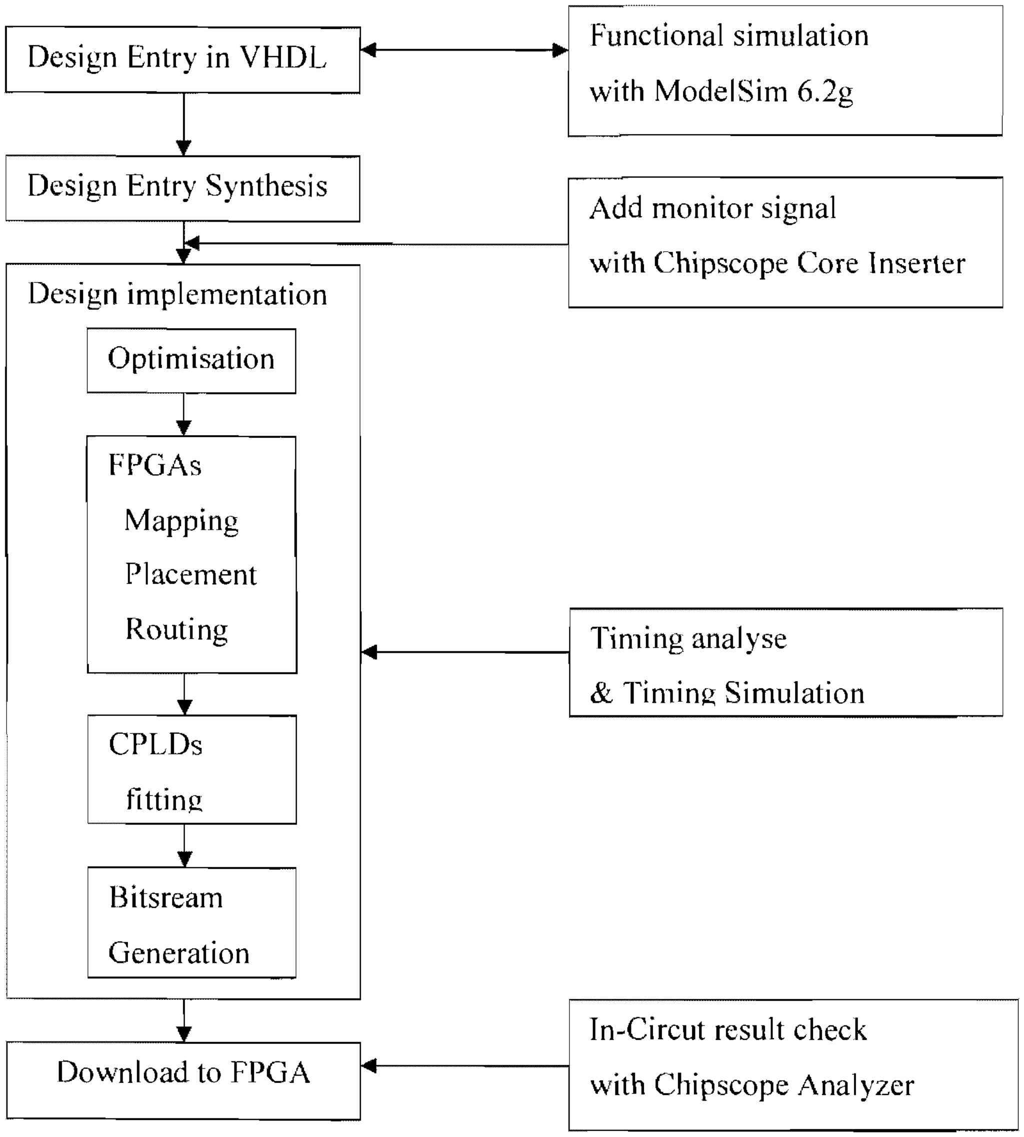
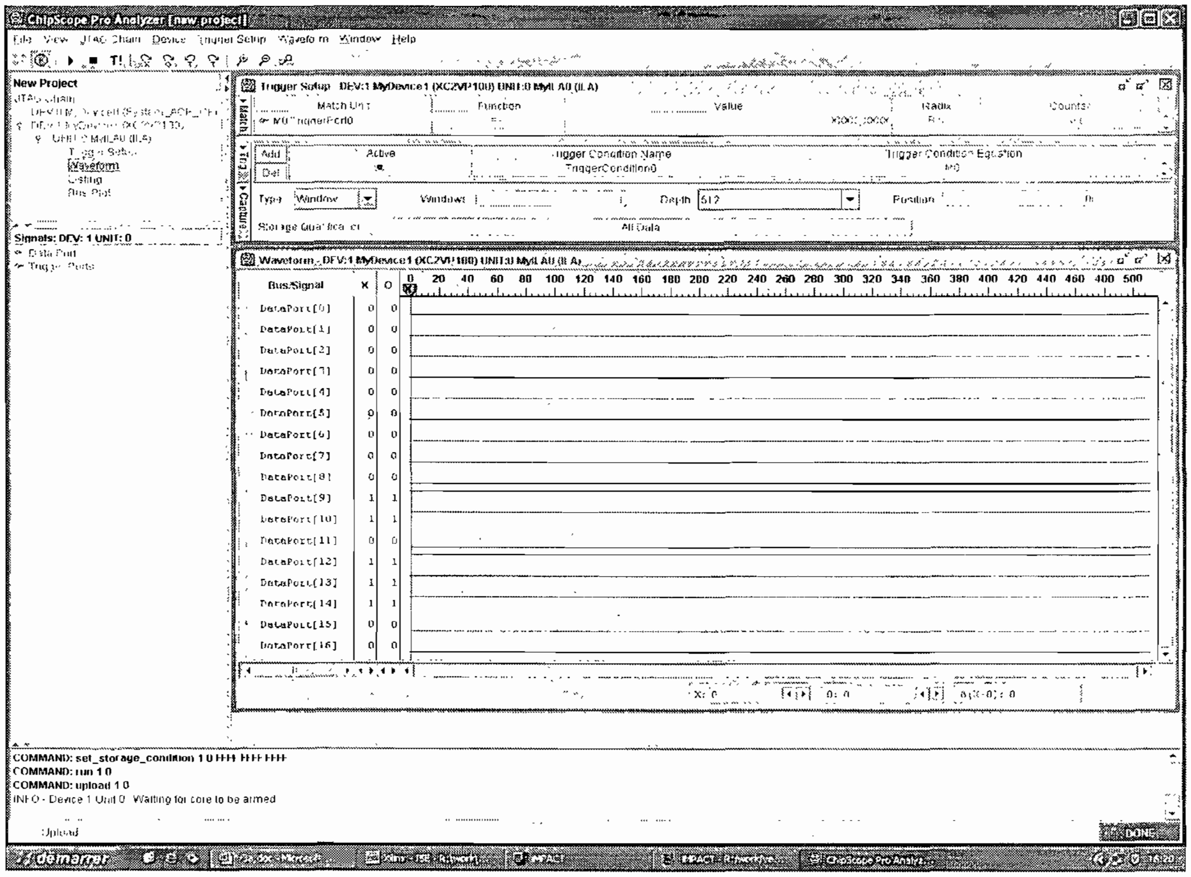

Design Entry in VHDL
Functional simulation with ModelSim 6.2g
Design Entry Synthesis
Add monitor signal with Chipscope Core Inserter
Design implementation
Optimisation
FPGAs
Mapping
Placement
Routing
Timing analyse & Timing Simulation
CPLDs fitting
Bitsream Generation
Download to FPGA
In-Circut result check with Chipscope Analyzer

Figure 19: Design flow

In the part of software application design, we use C# to program software applications for the SCIL processor system. Then we generate the SCIL file from the CIL file by using the SCIL compiler. Finally the SCIL file is used as the initial file to initialize the BRAM for instruction memory. In the SCIL processor system, in order to record the number of cycles to complete programs, we add one signal to the unit MIU. Before the SCIL processor runs the program, the value of this signal is 0. Then the value of this signal continues increasing till the processor completes executing the program. We can use Xilinx ChipScope Pro 8.2i to view the value of this signal. In Figure 20, we show the interface of ChipScope Pro Analyzer.


ChipScope Pro Analyzer [new project]
File View JTAG Chain Device Trigger Setup Waveform Window Help
New Project
Waveform
Listing
Bus Plot
Signals: DEV: 1 UNIT: 0
Data Port
Trigger Ports
Trigger Setup DEV:1 MyDevice1 (XC2VP100) UNIT:0 MyILA0 (ILA)
Match Unit
Function
Value
Radix
Counter
Add
Del
Active
Trigger Condition Name
Trigger Condition Equation
TriggerCondition0
M0
Type Window
Windows
Depth 512
Position
Storage Qualification
All Data
Waveform - DEV:1 MyDevice1 (XC2VP100) UNIT:0 MyILA0 (ILA)
Upload
DONE

Bus/Signal	X	O
DataPort[0]	0	0
DataPort[1]	0	0
DataPort[2]	0	0
DataPort[3]	0	0
DataPort[4]	0	0
DataPort[5]	0	0
DataPort[6]	0	0
DataPort[7]	0	0
DataPort[8]	0	0
DataPort[9]	1	1
DataPort[10]	1	1
DataPort[11]	0	0
DataPort[12]	1	1
DataPort[13]	1	1
DataPort[14]	1	1
DataPort[15]	0	0
DataPort[16]	0	0

```
COMMAND: set_storage_condition 1 0 HHHH HHHHHHH
COMMAND: run 1 0
COMMAND: upload 1 0
INFO - Device 1 Unit 0 Waiting for core to be armed
```

Figure 20: ChipScope Pro Analyzer

A MicroBlaze System

We create the MicroBlaze system by using Xilinx Platform Studio (XPS) 8.2i [27]. The device board is Virtex-II Multimedia FF896 Development Board which is presented in Figure 21. We use this platform board because we find that the platform board AP1000, which the SCIL processor system uses, does not support the MicroBlaze processor. (We failed in constructing a MicroBlaze system on AP1000 to the end.) The detail introduction of the board FF896 could be found in the reference [28].

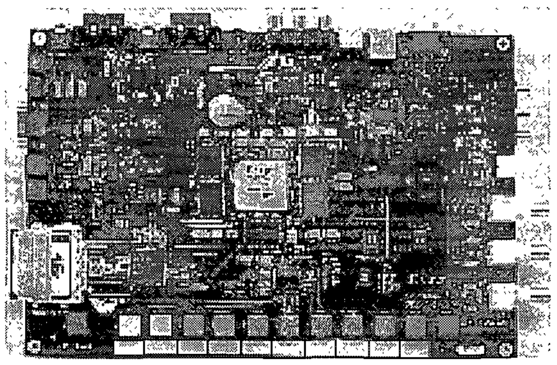


Figure 21: Virtex-II Multimedia FF896 Development Board

With Base System Builder wizard (BSB), we construct a simple MicroBlaze system. In Figure 22, we show the block-diagram generated by Xilinx Platform Studio 8.2i's Block-Diagram view for the MicroBlaze system. In this MicroBlaze system, there is one MicroBlaze processor as the embedded processor. There is only one BRAM (16k) with two ports because the instruction memory and data memory for MicroBlaze are combined into one single memory. The two ports of this BRAM connect to the MicroBlaze processor via two data buses.

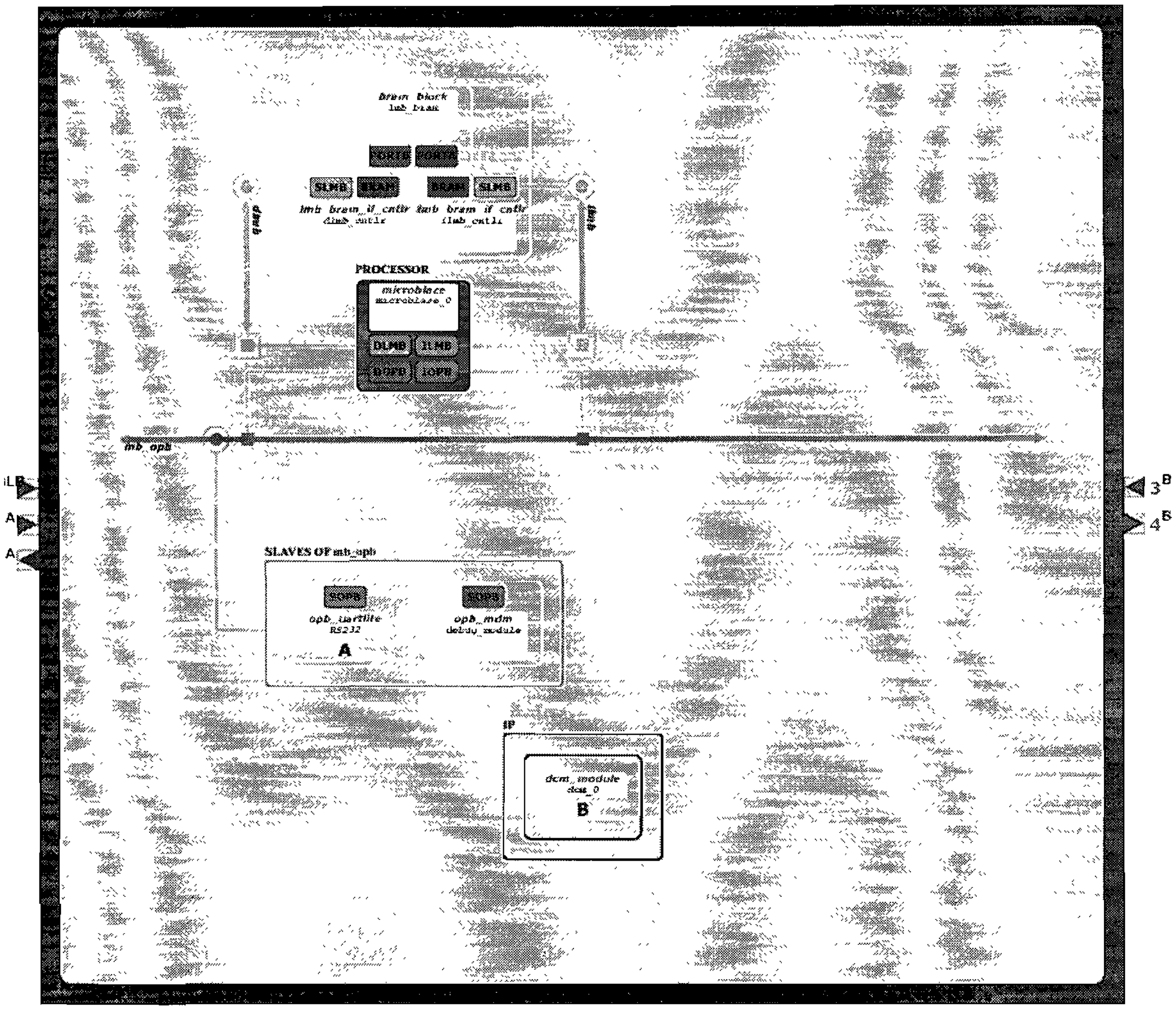
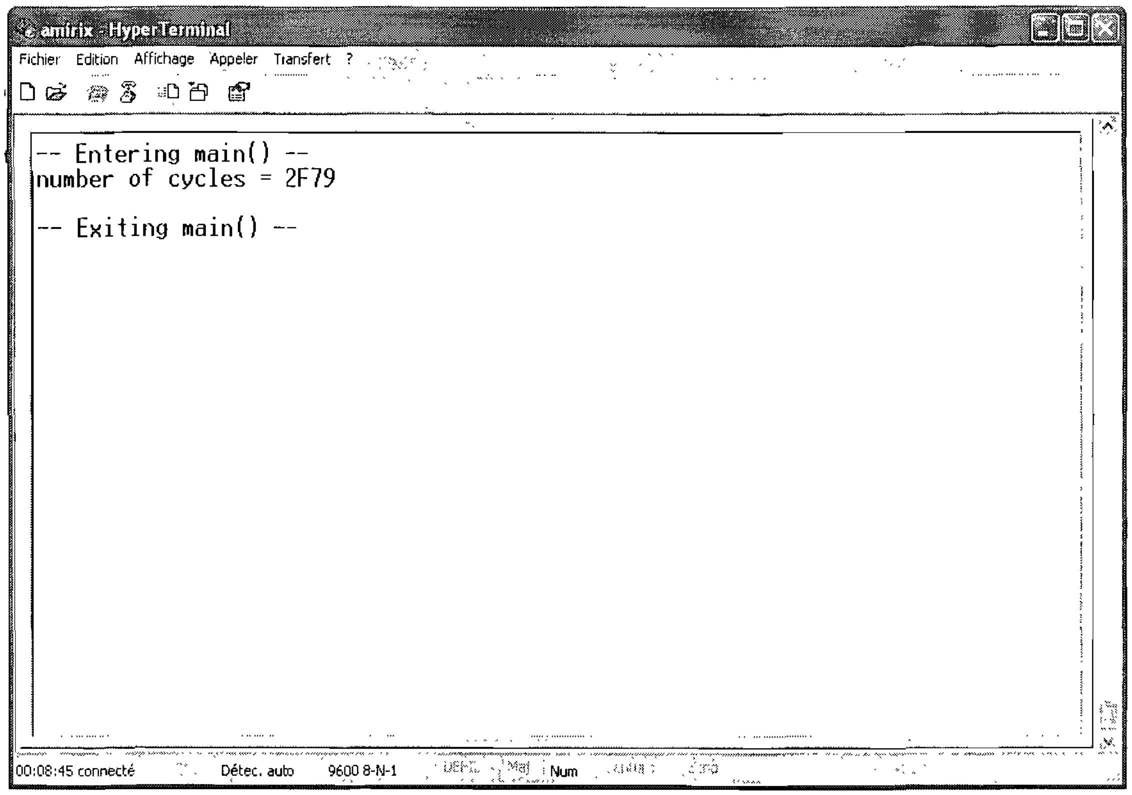


Figure 22: Block-diagram of MicroBlaze System

There are two peripheral controllers and one IP in this MicroBlaze system. We do not use the debug_module which is generated by the wizard. We use the peripheral controller RS232 port [29] as the output device. We connect this RS232 port to our computer's COM port. Moreover the IP we developed is responsible to count the number of cycles to complete programs. This number can be outputted via the RS232 port and finally shown on the compute screen by using the tool HyperTerminal. (See Figure 23)



```
-- Entering main() --
number of cycles = 2F79

-- Exiting main() --
```

Figure 23: HyperTerminal

In the software part, we use C to develop the software applications for the MicroBlaze system [30]. After compiling the software application, the XPS generates a bitstream file, which includes the system design and memory initial data. Then we download the bitstream file to the FPGA on develop broad.

Clock Rate and LUTs usages

Through timing analysis, the maximum clock rate achieved of the SCIL processor is 50 MHz. Moreover, the LUTs (Looking-Up Table) usage for our SCIL processor is 3308 on Virterx II RPO. We compare the LUTs usage with the MicroBlaze in Table 12. The LUTs used by MicroBlaze varies depending on the configuration used. We can see clearly that the cost of our processor and the cost of MicroBlaze are quite in the same range.

	LUTs Used	**Available**	**Utilization (%)**

MicroBlaze	800 – 2,600 [31]	88,192	0.9 - 2.9
CIL Processor	3,308	88,192	3.7

Table 12: LUTs utilization on the Virtex II PRO

Benchmarks

We respectively do four benchmarks:

- *Fibo* – computes the Fibonacci number
- *QSort* – sorts an array of integers using the quick sort algorithm with recursive procedure calls
- *BSort* – sorts an array of integers using the bubble sort algorithm
- *CRC32* – Cyclic Redundancy Checksum with digital signature is a 32-bit number

Benchmarks *QSort, Bubble Sort, CRC32* are modified versions of the benchmarks from MiBench [32]. We compare the cycle utilization for Microblaze and our SCIL processor.

Fibo

The Fibo does calculate the Fibonacci number. This benchmark is simple and only includes basic operations. In Table 13, we show the number cycles that both the SCIL processor and the MicroBlaze processor need to execute the Fibo benchmark. The N is the sequence length of Fibonacci number. The maximal value of N is 46 because the SCIL processor only supports 32 bits integer.

N	**10**	**20**	**46**
MicroBlaze	40	70	148
SCIL processor	930	1880	4350

Table 13: Cycle utilization for the benchmark Fibo

CRC32

The CRC32 does cyclic redundancy checksum for the list of 32-bit words. This benchmark includes many bit operations such as XOR and shift. In Table 14, we show the number cycles that both the SCIL processor and the MicroBlaze processor need to execute the CRC32 benchmark. The N represents the number of words to do CRC32 operation.

N	100	500	1000	2000	4000
MicroBlaze	453	1,653	3,153	6,153	12,153
SCIL Processor	4,933	10,373	17,173	31,000	77,973

Table 14: Cycle utilization for the benchmark CRC32

BSort

The BSort does sort an integer array by using the algorithm bubble sort. This benchmark includes lots of comparison operations and array operations such as loading element and storing element. In Table 15, we show the number cycles that both the SCIL processor and the MicroBlaze processor need to execute the BSort benchmark. The N is the size of the array.

N	10	50	100	200
MicroBlaze	831	21,613	75,756	313,087
SCIL Processor	13,260	297,247	1,271,967	4,963,570

Table 15: Cycle utilization for the benchmark BSort

QSort

The QSort does sort an integer array by using the algorithm quick sort. Different from the BSort, this algorithm use recursive procedure calls. Hence in this benchmark, there are lots of call instructions. In Table 16, we show the number cycles that both the SCIL processor and the MicroBlaze processor need to execute the QSort benchmark. The N is the size of the array.

N	10	50	70	80	100
Microblaze	1,444	21,584	40,054	51,389	unable
SCIL processor	955	14,007	25,933	33,246	50,572

Table 16: Cycle utilization for the benchmark QSort

Discussion

For the result of the benchmarks, we can see the performance of Microblaze processor is better than that of the SCIL processor. The MicroBlaze expresses good performance when it executes the first three benchmarks. These three benchmarks do not include many procedure calls. The number of cycles needed by the MicroBlaze is much less than the cycles needed by the SCIL processor. In fact, even when our processor could complete one microinstruction during each cycle, which is the limit for the single pipeline architecture, the SCIL processor should still use more cycles than that the MicroBlaze processor needs. From our viewpoint, the performance of the SCIL processor is not bad and acceptable. Taking the benchmark Fibo for example, the SCIL processor need to execute about 170 SCIL instructions when N=10. Because usually one SCIL instruction needs 4.5 microinstructions to complete its functionality, the SCIL processor has to execute about 700 microinstructions in sum. The number of cycles the SCIL processor needs to execute this program is 930. So the CPI (Cycles Per Instruction) for microinstructions is 1.32. We think it is acceptable because there should exist many suspend cycles among these microinstructions.

The result of benchmark Qsort inspires us very much. We can find that the SCIL processor uses less number of cycles than the Microblaze uses. The benchmark Qsort includes many recursive procedure calls and it can show the superiority of our processor. Because the SCIL simplifies the CIL by using the absolute address as instruction operands for branch address, the SCIL processor uses tiny time to invoke a procedure. The processor can use static branch jumps because the SCIL does not support the object-oriented concept. The SCIL compiler can calculate all branch addresses before we load SCIL instructions into the memory.

Chapter 6 Conclusion and Future works

Currently embedded processors are used widely in embedded system designs. The embedded processors can accelerate the development period of embedded systems, and let the embedded system designers start their works at high abstract level. We introduced a new embedded processor targeted for Microsoft's CIL. The SCIL Processor is a synthesisable softcore processor, and it implements a subset of the CIL. Since the CIL is the intermediate language for the all .NET languages, it is possible for designers to use all languages of .NET framework as the programming language to develop software applications for embedded systems. However, because the CIL has many characteristics of high level languages, it is difficult to implement the CIL directly on hardware. We adopted the approach of simplifying CIL instructions, and converted them to SCIL instructions via a small complier. The SCIL, as the machine language for the SCIL processor, improved the performance of the processor and reduced the amount of needed hardware resources. The SCIL processor modified the architecture of Tanenbaum's IJVM processor to adapt to the SCIL instruction set. Moreover the SCIL processor used a predictor and the forwarding technique to reduce the number of suspending cycles. We illustrated the performance of our processor and compared benchmark results of the SCIL processor system with a MicroBlaze processor system.

The future works can continue in three directions. First of all, the performance of SCIL processor might be improved. We can use an eight-stage pipeline to replace current six-stage pipeline. Now the unit IFU and the unit MIU have to do lots of works. The IFU is responsible to fetch data from memory and decode instruction code. The MIU is responsible to search microinstructions and generate command signals. Both of two units needs relatively long time to complete their tasks. As a result, it is hard for us to improve the clock frequency of the processor further. Therefore, we can split the works of the MIU into two parts. We use two different units to do search microinstructions and do generate command signals respectively. In this way, each unit needs less time than the MIU needs. Accordingly, we can split the tasks of the IFU to two units. Then we modify six pipeline stages to eight pipeline stages. We think the clock frequency of the processor might be improved a lot after the processor using the new pipeline. Another approach of improving the performance is to implement some often-used instructions in circuitry. The processor does not need many hardware resources on these circuitries. Furthermore when the processor can execute these often-used instructions in a very short time, we think the processor could improve the performance remarkably. Secondly, we intend to enlarge the semantic of the language SCIL, and make the processor support the object-oriented concepts. Currently the SCIL is like the simple C and does not support object-oriented programming. However, it is necessary to make the processor support it if we hope the SCIL processor could be used in practice. In our opinion, the processor would need some modifications in order to support object-oriented concepts. The processor can add one memory to recode the address of object instances and another memory to converse these object instances. Moreover, in order to implement the garbage collection function effectively, we make the memory for object instances a little special. The whole memory is divided into two parts. Both of two parts can do garbage collection independently. When the processor is accessing data in one part of memory, this part suspends doing garbage collection. However another part can still do garbage collection. In this way, the processor can use object instances and do garbage collection at the same time. We think this approach of garbage collection can give the processor a good performance. Finally, it is possible to fit multiple SCIL processors into a same FPGA, we think it would be interesting to realize a network-on-chip design and measure the overhead. We can use different network topologies, such as bus network, ring network and star network, to connect various SCIL processors in a multiprocessor design. Furthermore, we can construct the systems with different number of SCIL processors to test the speedup obtained due to the use of many processors instead of one being used.

Reference

[1] A. S. Tanenbaum, Structured Computer Organization, 5/E, Prentice Hall, 800 pp, 2006, ISBN13:9780131485211

[2] Standard ECMA-335 Common Language Infrastructure, 3rd Edition , June 2005

[3] Michael Barr, Embedded Systems Glossary, Netrino Technical Library, Retrieved on 2007-04-21

[4] W.Warner, Great moments in microprocessor history - The history of the micro from the vacuum tube to today's dual-core multithreaded madness

[5] http://www.arm.com/miscPDFs/3823

[6] Erin Farquhar and Philip Bunce, MIPS Programmer's Handbook, Morgan Kaufmann Publishers, ISBN 1-55860-297-6

[7] http://www.atmel.com/products/avr/

[8] http://www.zilog.com/products/

[9] http://www.renesasinteractive.com/

[10] Microchip Technology Delivers Five Billionth PIC® Microcontroller, Press release, Retrieved on 2006-02-13.

[11] Nios II Processor Reference Handbook

[12] J.Michael, O'Connor, and Marc Tremblay, picoJava-I: The Java Virtual Machine in Hardware. IEEE Micro, 17(2):45–53, 1997

[13] http://www.sun.com/software/communitysource/processors/download_picojava.xml

[14] S. Dey, P. Sanchez, D. Panigrahi, L. Chen, C. Taylor, and K. Sekar, Using a Soft Core in a SOC Design: Experiences with picoJava. IEEE Design and Test of Computers, 17(3):60–71, July 2000

[15] K. B. Kent, H. Ma, and M. Serra, Rapid Prototyping of a Co-Designed Java Virtual Machine, RSP2004, 2004

[16] Lightfoot 32-bit Java Processor Core, Digital Communication Technologies Ltd.

[17] K. E. Plambeck, W. Eckert, R. R. Rogers, and C. F, Webb Development and attributes of z/Architecture, http://www.research.ibm.com/journal/rd/464/plambeck .html

[18] Principles of Operation, IBM

[19] z/Architecture Reference Summary, IBM

[20] MicroBlaze Processor Reference Guide, Embedded Development Kit EDK9.1i

[21] Paul Rogers, Luiz Fadel, Fernando Ferreira, zSeries Application Assist Processor (zAAP) Implementation, IBM/Redbook

[22] ANS T1.523-2001, Telecom Glossary 2000

[23] Matt Pietrek, The original Portable Executable, MSDN Magazine, March 1994

[24] .NET Framework Developer's Guide, Compiling MSIL to Native Code

[25] ChipScope Pro Software and Cores User Guide, ChipScope Pro Software 8.2i

[26] ISE 8.2i Development System Reference Guide

[27] Embedded System Tools Reference Manual, Embedded Development Kit, EDK 8.2i

[28] MicroBlaze and ultimedia Development Board User Guide, Xilinx

[29] Xilinx's OPB UART Lite (V1.00b), Xilinx Reference Guide

[30] EDK 8.2 MicroBlaze Tutorial in Spartan

[31] Xilinx, MicroBlaze FAQ, http://www.xilinx .com

[32] B. Fort, Simulation Tool for Soft Core Processor Performance Analysis, University of Toronto

[33] Platform Studio User Guide, Xilinx Embedded Development Kit EDK 7.1i

[34] Introduction to theQuartus® IISoftware, Altera Corporation

[35] Kathy Walsh, zAAP – what it can do for you, Introduction to zSeries zAAP Presentation, IBM

[36] Mike Ebbers, Wayne O'Brien, Bill Ogden, Introduction to the New Mainframe: z/OS Basics, IBM/Redbooks

[37] Martin Schoeberl, JOP: A Java Optimized Processor for Embedded Real-Time Systems

[38] John Edwards, No room for Second Place - Xilinx and Altera slug it out for supremacy in the changing PLD market, Electronic Business, 6/1/2006

Appendix

List of Supportable CIL Instructions

CIL instruction	SCIL instruction	SCIL Instruction Code
nop	nop	00000000
Dup	dup	00000011
pop	pop	00000110
ret	ret	00000100
add	add	00000001
add.ovf	add	00000001
Add.ovf.un	add	00000001
sub	sub	00000010
sub.ovf	sub	00000010
Sub.ovf.un	sub	00000010
and	and	00001010
or	or	00001011
xor	xor	00001100
neg	neg	00001101
not	not	00001110
newarr	newarr	00000111
Ret_main	ret_main	00000101
ldelem.i1	ldelem	00001001
ldelem.u1	ldelem	00001001
ldelem.i2	ldelem	00001001
ldelem.u2	ldelem	00001001
ldelem.i4	ldelem	00001001
ldelem.u4	ldelem	00001001
ldelem.i8	ldelem	00001001
ldelem.i	ldelem	00001001
ldelem.r4	ldelem	00001001
ldelem.r8	ldelem	00001001
ldelem	ldelem	00001001
stelem.i	stelem	00001000
stelem.i1	stelem	00001000
stelem.i2	stelem	00001000

stelem.i4	stelem	00001000
stelem.i8	stelem	00001000
stelem.r4	stelem	00001000
stelem.r8	stelem	00001000
stelem	stelem	00001000
shl	shl	00010000
shr	shr	00001111
shr.un	shr	00001111
ldarg.0	ldarg	01000101
ldarg.1	ldarg	01000101
ldarg.2	ldarg	01000101
ldarg.3	ldarg	01000101
ldarg.s	ldarg	01000101
ldarg	ldarg	01000101
ldloc.0	ldloc	01000110
ldloc.1	ldloc	01000110
ldloc.2	ldloc	01000110
ldloc.3	ldloc	01000110
ldloc.s	ldloc	01000110
ldloc	ldloc	01000110
stloc.0	stloc	01000010
stloc.1	stloc	01000010
stloc.2	stloc	01000010
stloc.3	stloc	01000010
stloc.s	stloc	01000010
stloc	stloc	01000010
starg.s	starg	01000001
starg	starg	01000001
param	param	01000100
Ldc.i4.m1	loads	11000001
ldc.i4.0	loads	11000001
ldc.i4.1	loads	11000001
ldc.i4.2	loads	11000001
ldc.i4.3	loads	11000001
ldc.i4.4	loads	11000001
ldc.i4.5	loads	11000001
ldc.i4.6	loads	11000001

ldc.i4.7	loads	11000001
ldc.i4.8	loads	11000001
ldc.i4.s	loads	11000001
ldc.i4	loads	11000001
ldc.i8	loads	11000001
ldc.r4	loads	11000001
ldc.r8	loads	11000001
call	call	10000010
jmp	br	10000001
br.s	br	10000001
br	br	10000001
beq.s	beq	10001000
beq	beq	10001000
bge.s	bge	10001011
bge.un.s	bge	10001011
bge	bge	10001011
bge.un	bge	10001011
bgt.s	bgt	10001010
bgt.un.s	bgt	10001010
bgt	bgt	10001010
bgt.un	bgt	10001010
ble.s	ble	10001100
ble.un.s	ble	10001100
ble	ble	10001100
ble.un	ble	10001100
blt.s	blt	10000111
blt.un.s	blt	10000111
blt	blt	10000111
blt.un	blt	10000111
bne.un.s	bne	10001001
bne.un	bne	10001001
local	local	10000011

Table of Microinstructions

SCIL code	Microinstructions	Comment
loads1	MDR=TOS=OPD	Copy OPD to TOS and MDR
loads2	MAR=SS=SS+1;wr	Increase SS and set MAR; write memory
add1	MAR=SS=SS-1	Read and store the word following the stack top
add2	TPR=TOS; rd	TPR = the stack top; read memory
add3	MDR=TOS=MDR+TPR; wr	Add two element; write memory
stloc1	MAR=LV+OPD	MAR=first variable address + displacement
stloc2	MDR=TOS; wr	MDR=the stack top; write memory
stloc3	MAR=SS= SS-1	Read and store the word following the stack top
stloc4	rd	Read memory
stloc5	TOS=MDR	Write the new stack top
starg1	MAR=PAR+OPD	MAR=first parameter address + displacement
starg2	MDR=TOS; wr	MDR=the stack top; write memory
starg3	MAR=SS=SS-1	Read and store the word following the stack top
starg4	rd	Read memory
starg5	TOS=MDR	Write the new stack top
call1	TPR=OPD	Copy OPD to TPR
call2	MAR=SS=SS+1	Increase SS; Copy new SS to MAR
call3	MDR=NPC; wr	Copy NPC to MDR; write memory
call4	MAR=SS=SS+1	Increase SS; Copy new SS to MAR
call5	MDR=TOS=LV; wr	Copy LV to MDR and TOS; write memory
call6	LV=SS	Copy new SS to LV
call7	BUS C=TPR	Output TPR via BUS C
local1	SS=SS+OPD	Increase SS with number of variables
local2	TOS=0	Reset TOS
Param1	TPR=OPD	Copy OPD to TPR
Param2	MAR=SS=SS+1	Increase SS; Copy new SS to MAR
Param3	MAD=PAR; wr	Cope PAR to MAD; write memory
Param4	PAR=SS-TPR	PAR=position of stack top – number of parameters
ret1	MAR=LV	Copy LV to MAR
ret2	MAR=SS=LV-1; rd	Set SS; Copy new SS to MAR; read memory
ret3	LV=MDR; rd	Copy MDR to LV; read memory
ret4	TPR=MDR	Copy MDR to TPR
ret5	MAR=SS=SS-1	Read and store the word following the stack top
ret6	MAR=SS=PAR; rd	Set SS; Copy new SS to MAR; read memory

ret7	PAR=MDR	Copy MDR to PAR
ret8	MDR=TOS; wr	Copy the stack top to MDR; write memory
ret9	BUS C=TPR	Output TPR via BUS C
ret_main1	MAR=SS=0	Reset MAR and SS
ret_main2	MDR=TOS; wr	Write the result to the first element
ret_main3	Output all '0'	End of program
Idarg1	MAR=PAR+OPD	MAR=first parameter address + displacement
Idarg2	MAR=SS=SS+1; rd	Increase SS; Copy new SS to MAR; read memory
Idarg3	TOS=MDR; wr	Set the stack top = MDR; write memory
Idloc1	MAR=LV+OPD+1	MAR=first variable address + displacement
Idloc2	MAR=SS=SS+1; rd	Increase SS; Copy new SS to MAR; read memory
Idloc3	TOS=MDR; wr	Set the stack top = MDR; write memory
br1	Z='1'	Set Z='1'
br2	JMP(Z)	Branch if Z
blt1	MAR=SS=SS-1	Read and store the word following the stack top
blt2	MAR=SS=SS-1; rd	Read and store the word following the stack top; Read memory
blt3	TPR=MDR; rd	Copy MDR to TPR; read memory
blt4	Z=TPR cmp TOS	If TPR<TOS then Z='1' else Z='0'
blt5	TOS=MDR; JMP(Z)	Set the stack top = MDR; Branch if Z
Beq1	MAR=SS=SS-1	Read and store the word following the stack top
Beq2	MAR=SS=SS-1; rd	Read and store the word following the stack top; Read memory
Beq3	TPR=MDR; rd	Copy MDR to TPR; read memory
Beq4	Z=TPR cmp TOS	If TPR=TOS then Z='1' else Z='0'
Beq5	TOS=MDR; JMP(Z)	Set the stack top = MDR; Branch if Z
Bne1	MAR=SS=SS-1	Read and store the word following the stack top
Bne2	MAR=SS=SS-1; rd	Read and store the word following the stack top; Read memory
Bne3	TPR=MDR; rd	Copy MDR to TPR; read memory
Bne4	Z=TPR cmp TOS	If TPR<>TOS then Z='1' else Z='0'
Bne5	TOS=MDR; JMP(Z)	Set the stack top = MDR; Branch if Z
Bgt1	MAR=SS=SS-1	Read and store the word following the stack top
Bgt2	MAR=SS=SS-1; rd	Read and store the word following the stack top; Read memory
Bgt3	TPR=MDR; rd	Copy MDR to TPR; read memory
Bgt4	Z=TPR cmp TOS	If TPR>TOS then Z='1' else Z='0'

Bgt5	TOS=MDR; JMP(Z)	Set the stack top = MDR; Branch if Z
Bge1	MAR=SS=SS-1	Read and store the word following the stack top
Bge2	MAR=SS=SS-1; rd	Read and store the word following the stack top; Read memory
Bge3	TPR=MDR; rd	Copy MDR to TPR; read memory
Bge4	Z=TPR cmp TOS	If TPR>=TOS then Z='1' else Z='0'
Bge5	TOS=MDR; JMP(Z)	Set the stack top = MDR; Branch if Z
ble1	MAR=SS=SS-1	Read and store the word following the stack top
ble2	MAR=SS=SS-1; rd	Read and store the word following the stack top; Read memory
ble3	TPR=MDR; rd	Copy MDR to TPR; read memory
ble4	Z=TPR cmp TOS	If TPR<=TOS then Z='1' else Z='0'
ble5	TOS=MDR; JMP(Z)	Set the stack top = MDR; Branch if Z
Sub1	MAR=SS=SS-1	Read and store the word following the stack top
Sub2	TPR=TOS; rd	TPR = the stack top; read memory
Sub3	MDR=TOS=MDR-TPR; wr	Subtract two element; write memory
And1	MAR=SS=SS-1	Read and store the word following the stack top
And2	TPR=TOS; rd	TPR = the stack top; read memory
And3	MDR=TOS=MDR and TPR; wr	AND two element; write memory
or1	MAR=SS=SS-1	Read and store the word following the stack top
or2	TPR=TOS; rd	TPR = the stack top; read memory
or3	MDR=TOS=MDR or TPR; wr	OR two element; write memory
not1	MAR=SS	Copy SS to MAR
not2	MDR=TOS=not TOS; wr	NOT the stack top; Copy new stack top to MDR; write memory
Neg1	MAR=SS	Copy SS to MAR
Neg2	MDR=TOS=-TOS; wr	Negative the stack top; Copy new stack top to MDR; write memory
Dup1	MAR=SS=SS+1	Increase SS; Copy new SS to MAR
Dup2	MDR=TOS; wr	Copy the stack top to MDR; write memory
Pop1	MAR=SS=SS-1	Read and store the word following the stack top
Pop2	rd	Read memory
Pop3	TOS=MDR	Copy MDR to TOS
xor1	MAR=SS=SS-1	Read and store the word following the stack top
xor2	rd	Read memory
xor3	TPR=not MDR	NOT MDR; Copy new MDR to TPR
xor4	TPR=TPR and TOS	AND two elements; Copy the result to TPR

xor5	TOS=not TOS	NOT TOS
xor6	TOS=MDR and TOS	AND two elements; Copy the result to TOS
xor7	MDR=TOS=TPR or TOS; wr	OR two elements; Copy the result to MDR and TOS; write memory
loadw1	TPR=shl16(OPD)	Copy high 16 bits to TPR
loadw2	MAR=SS	Copy SS to MAR
loadw3	MDR=TOS=TPR+TOS; wr	Combine high 16 bits and low 16 bits; write memory
newarr1	TPR=SS	Copy SS to TPR
newarr2	MAR=SS=SS+TOP	Increase SS with the size of array; Copy new SS to MAR
newarr3	MDR=TOS=TPR; wr	Copy the initial position of array to MDR and TOS; write memory
stelem1	MAR=SS=SS-1	Read and store the word following the stack top
stelem2	MAR=SS=SS-1; rd	Read and store the word following the stack top; Read memory
stelem3	TPR=MDR; rd	Copy initial position of array to TPR; read memory
stelem4	MAR=TPR+MDR;	MAR=initial position of array + index
stelem5	MDR=TOS; wr	Copy TOS to MDR; write memory
stelem6	MAR=SS=SS-1	Read and store the word following the stack top
stelem7	rd	Read memory
stelem8	TOS=MDR	Copy MDR to TOS
ldelem1	MAR=SS=SS-1	Read and store the word following the stack top
ldelem2	rd	Read memory
ldelem3	MAR= MDR+TOS	MAR=initial position of array + index
ldelem4	MAR=SS; rd	Copy SS to MAR; read memory
ldelem5	TOS=MDR; wr	Copy index element to TOS; write memory
shr1	MAR=SS	Copy SS to MAR
shr2	MDR=TOS=SHR8(TOS); wr	Shift TOS; Copy new TOS to MDR; write memory
shl1	MAR=SS	Copy SS to MAR
shl2	MDR=TOS=SHL1(TOS); wr	Shift TOP; Copy new TOS to MDR; write memory