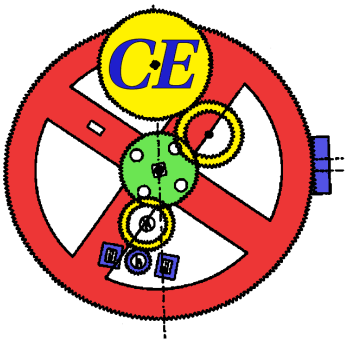

**Computer Engineering**
Mekelweg 4,
2628 CD Delft
The Netherlands
**http://ce.et.tudelft.nl/**

2010

# MSc THESIS

# Performance analysis of SCISM organization applied to the IA-32 architecture

**Eduard Gabdulkhakov**

**CE-MS-2010-03**

**Abstract**

There is a huge variety of processor microarchitectural techniques to decrease the program execution time, such as pipelining, branch prediction, and different methods to exploit the Instruction Level Parallelism (ILP). The Superscalar and VLIW machines are designed to exploit the ILP available in applications. These architectures improve performance by executing multiple independent instructions in parallel. However, this model faces some serious challenges, such as data hazards, and limited number of independent instructions that can be executed in parallel.

The Scalable Compound Instruction Set Machine (SCISM), also referred to as Superscalar Instruction Set Machine, proposes solutions for many of these challenges. The SCISM approach can be applied both to Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC) machines. SCISM performs a run-time analysis of decoded instructions to determine on the fly when they can be executed in parallel or not. This analysis is based on a predefined set of instruction compounding rules. Rules categorize instructions based on their hardware utilization. SCISM, furthermore, features interlock collapsing hardware, which can eliminate certain instruction interlocks, and often allows to execute in parallel instructions with real data hazards. Another important property of SCISM is that it is compatible with the existing instruction sets, thus, it does not require binary any code recompilation or translation.

In this work we have analyzed the IA-32 Instruction Set Architecture (ISA) with respect to compounding. The instruction categorization and compounding rules are applied to this ISA. The experiments are performed using Bochs x86 emulator implementing in-order execution. Very simple two-way instructions compounding is used, where the maximum of instructions that can be executed in parallel equals two. Experimental results, with SPEC CPU2006, demonstrate that such simple scenario the number of instructions executing in parallel varies from 13% to 24% for integer benchmarks and from 1% to 26% for floating-point benchmarks.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

# Performance analysis of SCISM organization applied to the IA-32 architecture

---

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

born in Birsk, USSR

January 26, 2010

# Performance analysis of SCISM organization applied to the IA-32 architecture

by Eduard Gabdulkhakov

**Abstract**

There is a huge variety of processor microarchitectural techniques to decrease the program execution time, such as pipelining, branch prediction, and different methods to exploit the Instruction Level Parallelism (ILP). The Superscalar and VLIW machines are designed to exploit the ILP available in applications. These architectures improve performance by executing multiple independent instructions in parallel. However, this model faces some serious challenges, such as data hazards, and limited number of independent instructions that can be executed in parallel.

The Scalable Compound Instruction Set Machine (SCISM), also referred to as Superscalar Instruction Set Machine, proposes solutions for many of these challenges. The SCISM approach can be applied both to Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC) machines. SCISM performs a run-time analysis of decoded instructions to determine on the fly when they can be executed in parallel or not. This analysis is based on a predefined set of instruction compounding rules. Rules categorize instructions based on their hardware utilization. SCISM, furthermore, features interlock collapsing hardware, which can eliminate certain instruction interlocks, and often allows to execute in parallel instructions with real data hazards. Another important property of SCISM is that it is compatible with the existing instruction sets, thus, it does not require binary any code recompilation or translation.

In this work we have analyzed the IA-32 Instruction Set Architecture (ISA) with respect to compounding. The instruction categorization and compounding rules are applied to this ISA. The experiments are performed using Bochs x86 emulator implementing in-order execution. Very simple two-way instructions compounding is used, where the maximum of instructions that can be executed in parallel equals two. Experimental results, with SPEC CPU2006, demonstrate that such simple scenario the number of instructions executing in parallel varies from 13% to 24% for integer benchmarks and from 1% to 26% for floating-point benchmarks.

**Laboratory** : Computer Engineering
**Codenumber** : CE-MS-2010-03

**Committee Members** :

| | |
|---|---|
| **Advisor:** | Georgi Gaydadjev, CE, TU Delft |
| **Chairperson:** | Koen Bertels, CE, TU Delft |
| **Member:** | Georgi Kuzmanov, CE, TU Delft |
| **Member:** | Wouter Serdijn, Electronics Research Laboratory, TU Delft |

# Contents

# List of Figures

# List of Tables

# Acknowledgements

The completion of this thesis would have been much difficult without the support of many people. First, I wish to thank my advisor, Assistant Prof. Georgi N. Gaydadjiev, for the opportunity to work on interesting topic and the successful completion. I am grateful to my family and my beloved girlfriend, for their encouragement and endless support during all my studies. I owe many thanks to my friends and colleagues: Demid Borodin, Juriaan Moolhuysen for support. I also want to thank the all other members and stuff of CE group for helping me to succeed with my thesis.

Eduard Gabdulkhakov
Delft, The Netherlands
January 26, 2010

# 1 Introduction

*This chapter describes the motivation behind this thesis in Section 1.1. Section 1.2 shows the objectives of this work. The organization of the thesis is represented in Section 1.3*

## 1.1 Motivation

The amount of data and information to be processed constantly increases. The increasing workload demands significantly more computational power. In the last decades, the integrated-circuit (IC) technology continued to improve, and these days we have order of magnitude more space on a chip to design more advanced and efficient Central Processing Units (CPU).

Modern processor designs use a big variety of microarchitectural techniques to improve the performance, e.g. pipelining, branch prediction, vector processing, exploitation of Instruction-Level Parallelism (ILP), and multithreading approach to exploit the Thread-Level Parallelism (TLP). The ILP exploitation is one of the promising performance boosters allowing to fetch and execute multiple instructions in a single machine cycle [15]. To exploit the ILP, for example, Very Long Instruction Word (VLIW) CPU microarchitecture executes instructions in parallel on a fixed schedule determined when programs are compiled. The compiler, analyses operations and combines them, if possible, in a bundle of multiple operations, which are executed in parallel. ILP exploitation technique is intensively studied [22][28] and widely applied in present days CPU's, e.g. IA-64 [18]. TLP exploits the available parallelism on a different level, than ILP. Due to the stalls, or dependencies between instructions the functional units are often idle[15]. Multithreading allows multiple threads to share the functional units in an overlapping fashion, in order to keep the processor busy during the stalls. In this thesis we focus on the ILP only.

*ILP* is a measure of the average number of operations or instructions in a program that a CPU might be able to execute simultaneously [29]. The *instruction parallelism of a CPU* is a measurement of the maximum number of instructions which a processor is able to execute at the same time. Due to the interlocks between the instructions, e.g. data dependencies, the instruction parallelism in the program is not enough to achieve the full performance potential of a processor. On the other hand some programs, e.g. a scientific application as *N-Body problem*, will not achieve their maximum performance potential because of the limited parallelism of a machine. In this work we address the general-purpose applications, which usually have a limited amount of available ILP.

The most popular machines are Very Long Instruction Word (VLIW) [11] and Superscalar [22]. The VLIW architecture is designed to exploit the ILP in the most straight-

forward way. The scheduling of the operations is performed by the compiler. Creating an efficient compiler, which schedules instructions for the parallel execution is the major challenge. The compiler combines scheduled instructions in an *issue packet* [15]. An issue packet is a way to decode the multiple operations or instructions as one instruction for the execution, and explained in Section 2.3.2. This implies that instruction scheduling is static, and during the execution the instruction sequence cannot be rescheduled or rearranged for better performance, in contrast to the Superscalar machines. The advantage of this organization is that the hardware is less complex as there is no need to check instructions for dependencies explicitly. Since the VLIW require a compiler for combining instruction in a specific way, a VLIW organization is not backward compatible, and to achieve the maximum performance gain programs need to be recompiled and retested.

In contrast to VLIW, Superscalar machines schedule instructions dynamically. The cost of dependency checking of instructions for parallel issuing is very high [7]. To estimate the cost of instruction preprocessing, consider the organization designed to process at most two instructions. The amount of instructions in the instruction set is $N$. The set of instructions which are allowed for parallel execution is $W \in N$. Furthermore, assume that preprocessing is based on the opcode description. For maximal exploitation of instruction parallelism, $W \times W$ rules need to be implemented.

The Superscalar organization does not require the compiler to schedule the instructions. One of the main advantages of the Superscalar organization is backward compatibility. For example, the IA-32 machines, the Superscalar Intel Pentium 4 processor is still able to execute programs compiled for the 386 Intel processor. However Superscalar machines might not exploit all their potential performance for a variety of reasons, e.g. structural, data and control hazards [19][12]. A structural hazard occurs from resource conflicts in hardware. Furthermore, not all combinations of instructions are supported for parallel execution, which has also its influence on performance. By the data hazards an instruction has to wait for the result of the previous instruction and it is not possible to execute these two instructions together in parallel, for example:

```
R1 := R2 + R3
R4 := R1 - R5
```

To calculate the value of the register $R4$, the second operation needs the value of the register $R1$, which has to be provided by the previous operation.

Control hazards occur by the conditional branches, that change the Instruction Pointer (IP) if branch is taken. Until the branch is resolved, the microprocessor can not execute the next instruction. To diminish this effect branch prediction techniques are used, where address on the next instruction can be predicted.

The Scalable Compound Instruction Set Machine (SCISM) [27] is an organization for the microprocessor architecture which exploits ILP without the need to design a new architecture from scratch and to recompile existing binary code. It also addresses important challenges of Superscalar and VLIW machines, e.g. true dependencies, the limited combinations of instructions to be executed in parallel, by analyzing instructions at execution time by use of specific *rules*, and by *compounding* them together for parallel execution. This organization can be applied to complex instruction set computer (CISC)

and to reduced instruction set computer (RISC) machines. Another property of SCISM organization is the use of *interlock collapsing units*, that allow the elimination of certain true dependencies.

SCISM compounds instructions for parallel execution according to a predefined categorization based on hardware utilization. For example, instructions *add, sub, logical or, logical and* are all executed by the Arithmetic Logic Unit (ALU) and instructions *shift arithmetic, rotate arithmetic* are executed by the shifter. The use of compounding rules, rather than rules based on the opcode description, reduces the complexity of instruction preprocessing hardware. Also the number of instructions which can be executed in parallel is higher, due to the fact that more instructions can be covered by the categorization, based on hardware utilization.

Existing interlocks between instructions are diminishing performance of Superscalar machines. Some hazards can be eliminated by use of *interlock collapsing units*, e.g. Interlock Collapsing ALU (ICALU) [31], multi-port data-cache [25], and multi-port address unit (AU) [27]. For example, Interlock Collapsing ALU (ICALU) is able to execute two arithmetic instructions, e.g. addition and subtraction, with data dependency as one compound instruction.

The IA-32 (also referred as x86, x86-32) is the instruction set architecture of the most popular general-purpose processors. IA-32 ISA is implemented by different vendors, e.g. Intel, AMD, and VIA. Due to the limitation of 32-bit architecture a 64-bit general-purpose architecture was emerging. Intel introduced the new IA-64 architecture with attempt to exploit ILP based the Explicitly Parallel Instruction Computing (EPIC)[28]. IA-64 machine is not able to run a IA-32 code. To run a legacy binary code a separate IA-32 engine with low performance was integrated on the Itanium IA-64 processors. AMD, in contrast to Intel, did not change the ISA, but just extended it with 64-bit architecture. One of the advantages of AMD x86-64 architecture is no performance sacrifice of legacy 32-bit code. The 64-bit (x86-64) extension, introduced by AMD, is also accepted by Intel. The x86-64 processors are Superscalar and have some disadvantages, which are describe earlier. The SCISM organization, which is has promising performance gain on the ILP exploitation, is fully compatible legacy binary code. In this thesis we want to determine how the SCISM compounding scheme can potentially improve the performance of Intel machine. Therefore we focus on compounding of the 32-bit Intel Architecture (IA-32 ) instructions.

## 1.2 Thesis Objectives and Methodology

The previous section briefly described the SCISM organization and mentioned that we focus on IA-32 ISA. The aim of this thesis is to estimate the possible performance gain of the SCISM compounding scheme when applied to the IA-32 machine. Therefore we use the emulator of IA-32 machine.

### 1.2.1 Objectives

Applying the SCISM organization to IA-32 will require the following stages:

- Define the hardware assumptions, e.g. functional units and their number, interlock collapsing units;
- Analise and categorize IA-32 instructions, based on the hardware assumption;
- Define instruction compounding rules.

First, we define the functional units which are available on the supposed IA-32 SCISM microarchitecture. Each functional unit executes a subset of the instructions. For example, the ALU performs all integer arithmetic and logical operations. The Floating Point Unit executes all operations on floating point numbers. Second, we categorize subsets of instructions according to the utilization of the hardware. For instance, addition, subtraction, logical OR, etc. utilize the ALU, thus these instructions are in the same category.

Last, instructions are categorized according to the hardware utilization to define the compounding rules. These rules depict if an instruction is compoundable with one of the next instructions, or not.

### 1.2.2 Methodology

To determine how many instruction can be compounded using the SCISM organization the following steps are required:

- Add the SCISM compounding functionality on an existing IA-32 simulator;
- Use widely excepted benchmarking software for performance evaluation.

The compounding facility should be able to analyze instructions for parallel execution at run-time. Bochs, an IA-32 emulator, will be used to implement the compounding facility. To analyze the performance, the number of compounded instructions (possible to executed in parallel), we use the benchmarking (SPEC CPU2006) suite.

## 1.3 Thesis Organization

The remainder of the thesis is organized as follows. The main properties of the SCISM organization, the IA-32 instruction set, and the comparison of SCISM tagging to 64 bit Intel Architecture (IA-64) is presented in Chapter 2. Chapter 3 explains the compounding scheme on the IA-32 instruction set, the hardware assumption, the IA-32 instruction categorization, and the resulting compounding rules. To implement the compounding facility the emulation software is used. The internal structure of Bochs x86 emulator and implemented compounding unit is described in the Chapter 4. Chapter 5 shows the benchmarking methodology and discusses the obtained results. The conclusion and future work is given in Chapter 6.

# 2 Background and Related Work

*This chapter introduces the background information about the Scalable Compound Instruction Set Machine (SCISM) organization and Intel Architecture. We describe the instruction categorization approach, and proposed cache preprocessor design in Section 2.1. The interlock collapsing units, which are able to gain more performance from ILP exploitation are explained in Section 2.2. Section 2.3 describes the Intel Architecture 32 instruction set (IA-32). Last Section compares the 64-bit processor architecture based on the IA-32.*

## 2.1 SCISM Main Properties

This section explains main properties of SCISM. Section 2.1.1 discusses how the instructions can be categorized. The SCISM organization is depicted in Section 2.1.2. Section 2.1.3 describes the cache preprocessor design, used for this thesis.

### 2.1.1 Instructions Categorization

Consider a two-way in-order issuing Superscalar IA-32 machine, which is able to execute maximum of two instructions in parallel. To determine whether two instructions are compoundable, we have to implement the rules for each possible combination. We marked 320 (section 3.3) of 707 [5] opcodes of IA-32, which we consider for parallel issuing. For example, an addition, and arithmetic *right* shift instructions are in the the subset of IA-32 instruction for pairing. The rule depicts, that these instructions can be executed in parallel *only if there is no dependency*. To determine if an addition instruction is compoundable with arithmetic *left* shift instruction we need another rule. If rules for the parallel execution are based on the opcode description the number of combinations of instructions is: $320 \times 320$. This is the case if we want to exploit the maximum instruction-level parallelism. Of course, this approach results in a complex instruction-analysis hardware that will impact the cycle time. Some processors use this approach [17], but just for a small combination of instructions. Instead of making rules for each pair of instructions, the Scalable Compound Instruction Set Machine (SCISM) [27] organization groups instructions according to their hardware utilization. For example, arithmetic logic unit (ALU), floating point unit (FPU), shifter, branch unit, etc. The rules, based on the hardware utilization, determine, for example, on what condition an instruction performed by the ALU with another instruction performed by the shifter, can be execute in parallel.

This basis of the hardware utilization is formed by the following characteristics:

1. Instruction in an instruction set are partitioned into several categories;
2. An instruction may be in a particular category if and only if it utilizes the same hardware unit(s) as the other instructions;
3. All instructions in a category are considered as "unique";
4. The difference between instructions in a category is considered as "trivial" and are resolved by the hardware.

Instructions which do not meet these definitions can either be assigned to specific categories or be assigned together in a single category. The latter minimizes the number of categories and simplifies the implementation.

Categorization of instructions by the hardware utilization reduces the number of rules significantly, since the number of functional units is limited. There is a multiplicity of instructions operating on each functional unit, or on fixed set of units. Instructions add, subtract, compare, logical AND, logical OR, and etc. work on the arithmetic logic unit (ALU). Such instructions as conditional jump require the ALU and the branch unit.

Although instructions in one category differ from each other, the difference is "trivial". Instructions performed by the same functions unit can be distinguished by the existing control signals. For example, in two's-compliment arithmetic instruction addition and a subtraction have difference only in a inversion and a "hot 1" carry [23].

Instruction categorization with limited amount of rules reduces the complexity of the preprocessing hardware. The SCISM preprocessor example is described in Section 2.1.3. Compounded instructions produced by the preprocessing mechanism are executed as a "single" instruction.

### 2.1.2 Organization of SCISM

The SCISM is a Superscalar [33] organization, which compounds instructions for parallel execution according to categorization based on hardware utilization. Instructions categorization by functional units utilization reduces the complexity of rules, which analyze whether a pair of instructions to be *compounded* or not. If instructions are compounded, they are executed as one instruction.

The organization of the SCISM machine is not dependent on the architecture of a processor. This makes SCISM suitable for both RISC and CISC architectures [27].

In the SCISM organization, instructions from the same category cannot be executed simultaneously since they use the same hardware resource, and therefore are considered as "unique". However, it is also dependent on the microarchitecture of the processor, where, for instance, may be two ALU's, and there is a possibility to execute two instructions from the same category. Instructions with interlocks, e.g. data interlocks, memory interlock, and execution interlocks are diminishing the amount of instruction to be executed in parallel. In certain cases, interlock can be even collapsed (Section 2.2).

The distinction among category members is regarded as "trivial" and can be resolved by the hardware. Instructions can be examined either in the compiler or on execution stage. In the content of this paper we assume that instructions are being analyzed during the execution, as it is described in [27].

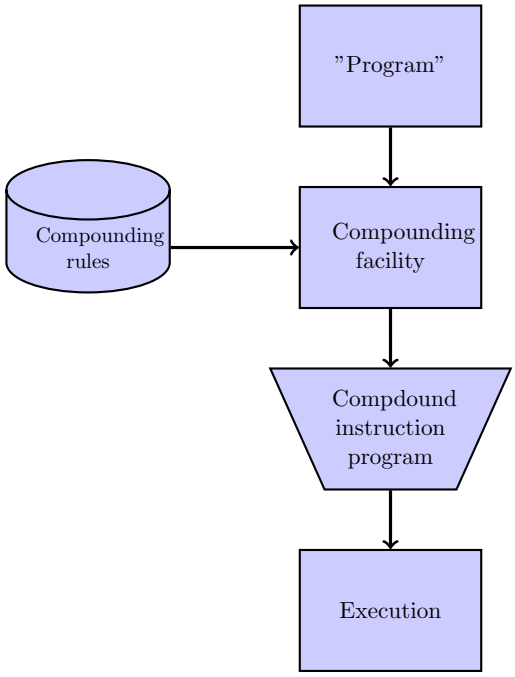


Figure 2.1: Abstract SCISM representation

Figure 2.1 shows the high level of abstraction of the SCISM organization. The *"Program"* is an input of the *Compounding facility* (preprocessor). At this step the instruction stream (the binary code stream) is analyzed, and *Compound instruction program* is produced. If instructions are compounded, they are executed by the execution engine, as a single instruction.

How the instructions are compounded is determined by the *"Compounding rules"*. These rules reflect the system microarchitecture, hardware organization, and achievable parallel execution among the group of instructions.

The compound instruction program is executed by the *Execution Unit*, and every compound instruction is handled as a single instruction. The compound instruction holds information related to parallel issuing of instructions. This information can be incorporated by tagging or decoding. Tagging is preferred since it allows the variable length of composed instruction and ability to intermingle instructions with data. The tag shows the boundaries between single instruction and the ones composed together.

SCISM machine allows out-of-order execution [27]. In order to compound instructions out-of-order, they should be rescheduled in the Compounding Facility and tagged for parallel execution.

The instructions $I_1$, $I_2$, and $I_3$ depicted in Figure 2.2 are in their original form, and all $T$'s are the tags bits. The tag contains information about compound boundaries. For sake of simplicity, we assume that tag is only one bit long, and there only two instructions can be compounded. If the tag contains value is "1" then the instruction is compounded with the next instruction. If tag is 0, instructions should be executed sequentially. In

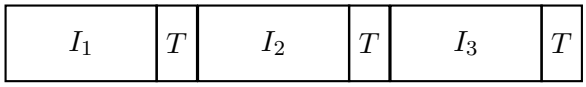


Figure 2.2: Compound instruction format using tagging

this example if the tag of instruction $I_1$ is 1, instruction $I_2$ is issued together with the first one.

The tag can contain as much information as needed and is not necessarily one bit long. For example, to determine the compounding of three instructions, two bits are required, denoted as $t_0$ and $t_1$. To delimit compounding of three instructions, we use 00 to represent a single instruction, 01 for compounded two-instructions and 10 for three-instructions [27].

The preprocessing in the SCISM is detached form the decode/issue stage of the processor. This is a fundamental property of the SCISM machine. It implies that instruction compound is "permanent". The compound facility may be in software in the form of a post compiler or implemented as hardware facility. The preprocessing of instruction by the post compiler [27] implies less hardware complexity. If the preprocessor is implemented in the hardware, it can be located before the cache of the processor. The hardware preprocessor fetches instructions from memory, tags instructions, and after that instructions can be loaded in the cache. The compound instruction remains intact as long as needed. If, for instance, the line in the cache is not valid any longer, tags become invalid also. It remains that at this point tags are relatively "permanent".

The preprocessing can be done, for example, during the cache miss or servicing times. It is possible, due the fact that the main memory speed is much slower than processor speed, processor has to wait for the to arrive from the memory. The latency can grow up to 500 - 1000 cycles [8].

One of the important challenges of a Superscalar processors is the branch handling. If branch is taken, the next instruction to be executed can be either at place of memory which is not preprocessed by the compounding facility yet, or to already preprocessed one. When the next instruction is not preprocessed, the compounding facility should fetch instructions and compound them. In cases, when the next instruction is a compounded instruction, the execution unit should execute compounded instruction. But the branches can occur in the middle of a compound instruction also.

This situation is illustrated in Figure 2.3 where issue maximum three instructions and $T$ is the field associated with its instruction. For sake of simplicity, the the size of tag to 1 bit assumed. The $m$th compound instruction expressed in terms of $CI^m$. For example, the first compound instruction looks as: $I_1^1\ B_2^1\ I_3^1$. Within the compounded instruction $CI_r^m$ the $r = 1$ stays for the first instruction $I_1^m$ and so on. The second instruction of $CI^1$ (first compounded instruction) is a branch instruction $B_2^1$ and can take the path $a$ or $b$. In this example the $a$ path branches to middle of instruction $CI^j$, while path $b$ branches to the beginning of same instruction. If $a$ path is taken instructions $I_2^j$ and $I_3^j$ as a compound instruction $CI^j$. The following compound instruction $I_1^k$ is fetched for sequential execution, with the next $I_2^k$ and $I_3^k$ instructions. By branch path $b$ the

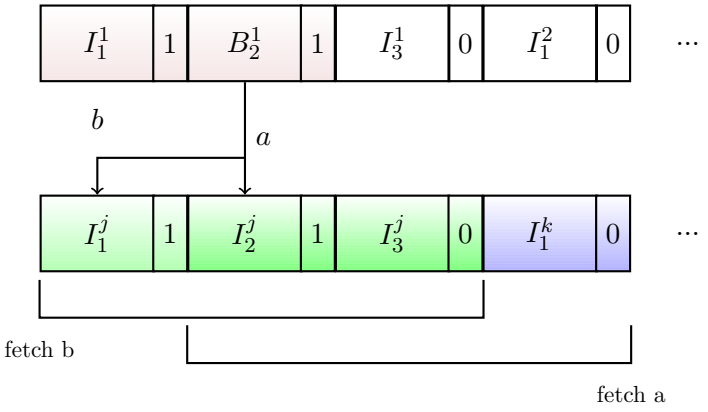


Figure 2.3: Branching to the "middle" if a compound instruction, adapted from [27]

complete $CI^j$ instruction is to be executed by the hardware where all instructions $I_1^j$, $I_2^j$ and $I_3^j$ are included.

### 2.1.3 Cache Preprocessor Design

The Complementary Metal-Oxide-Semiconductor (CMOS) technology is in continuous improvement, and there are already a lot of available resources. Since the beginning of 90's the feature size decreased about hundred times. The complex techniques, that were feasible for mainframes on the 80's and 90's can be implemented on-chip. In [27] it is assumed that the compounding facility is placed in the cache subsystem of System/370. We assume that the current CMOS technology has sufficient resources integrate SCISM into an existing processor.

In the previous section we mention that the preprocessing and compounding facility can be in software(postcompiler), main memory [9], or in the cache. The following properties of the SCISM in-cache design have to be considered:

1. During the cache miss it is possible to preprocess instructions;
2. The tag storage overhead is added only to the cache memory and to the fetch and issue units;
3. The cost of hardware for specific problems such as writing to the instruction stream, data intermingled with instructions, and variable-length instruction are considered as acceptable.

The Figure 2.4 gives the overview of an instruction compounding unit (ICU) with units as instruction compound unit (ICU), compound instruction cache (CIC), and miscellaneous functional units. Instructions are fetched from the main memory subsystem and analyzed by the ICU in combination with the compound instruction fetch controller (CIFC) sequential machine. The CIFC is a unit for controlling the entire process of supplying compounded instructions to the functional units, requesting line fetches from the

memory, and other tasks. The ICU consists of buffer, decode/analysis unit, rules base, branch and pipeline analyzer, and compounder. The buffer contains the instruction to be preprocessed by the ICU and delivers the compounded instructions, including tags, to the CIC. Certainly, the bigger buffer the wider the scope of preprocessing.

The decode/analysis unit decodes the instructions in the buffer, finds the interlock between instructions and passes the result to the rule base and branch and pipeline analyzer. The compounder uses the output of the buffer to produce the compound instruction to the CIC. The rule base described in Section 2.1.1 may contain rules on the complete instruction set or on a subset of instruction set. Furthermore the rules may also be in a form of fast accessible programmable storage, giving the possibility to reconfigure the rule base for a specific computing environment. The CIC design is similar to traditional cache, with the addition of the tag bits. That implies that the techniques applied to cache can be also used in the CIC design. The preprocessing design for IBM System/370 [27] suggests that the delay through the CIC will be less than one machine cycle.

## 2.2 Execution Interlocks Resolution Units

The true data dependencies between instructions have an adverse effect on the performance of Superscalar machines by forcing serial execution. We can distinguish two kinds of interlocks: (1) execution interlocks, and (2) memory interlocks. Memory interlocks, e.g. the load-use, force instructions to wait until a value is loaded from the memory subsystem. In case of the execution interlocks, e.g. register read-after-write hazards, an instruction waits for the result of the preceding instruction. In certain cases it is possible to execute instructions in parallel even with data dependency.

Example of data dependencies is shown in Figure 2.5, where each instruction is interlocked, because instructions (b) and (c) use the value of preceding instructions. Due to the data hazard each instruction has to be scheduled serially.

First pair of instructions (a) and (b) have a memory interlock, and cannot be removed. The interlock between instruction (b) and (c), however, can be eliminated such that the add operation (b) can be performed in parallel with effective address operation (c). To execute a group of instructions in parallel it is necessary that implementation incorporate both:

- multiple execution units;
- multi-operand execution units.

Multiple execution units allow to execute multiple instructions, which utilize the same functional unit. In addition, the multi-operand execution units allow to execute certain instructions in parallel even when interlock is present.

### 2.2.1 Interlock Collapsing ALU

Consider the next instruction sequence:

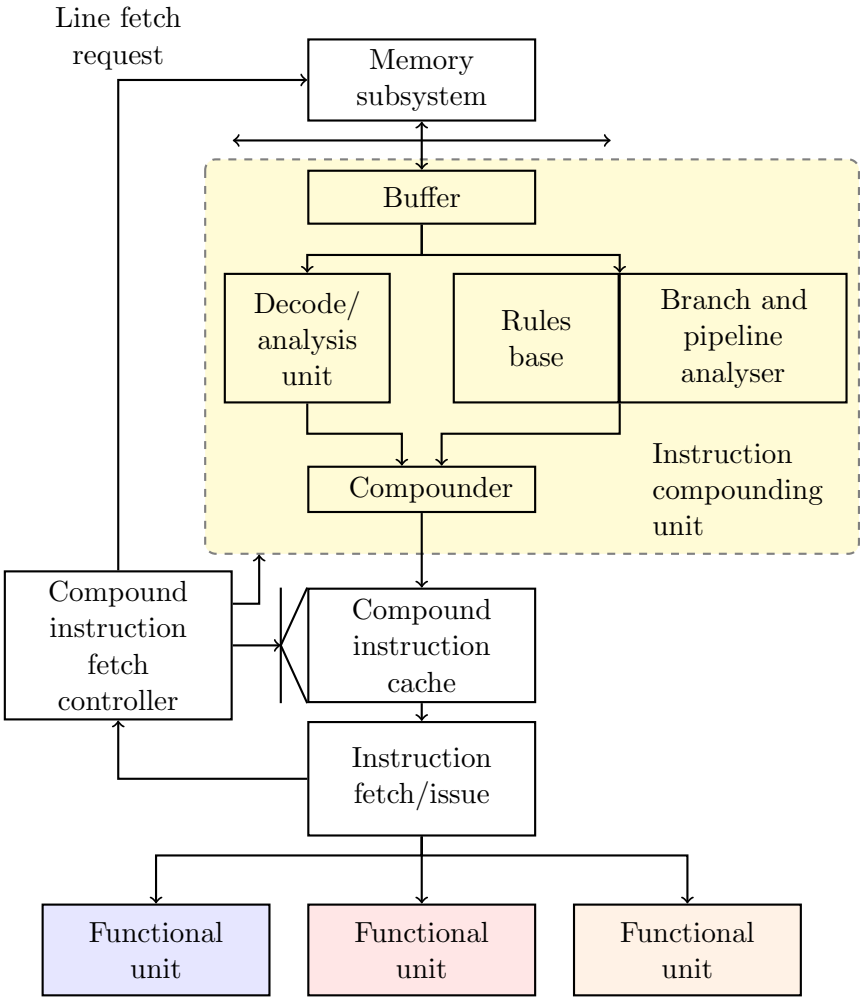


Figure 2.4: SCISM with cache processor, adapted from [27]

```
a. mov  edx,   [esi-4] ; EAX    := -4[ESI]
b. add  eax,   edx     ; EAX    := EAX + EDX
c. mov  [eax], ebx     ; 4[EAX] := EBX
```

Figure 2.5: IA-32 instruction sequence with interlocks

```
add  eax,   edx     ; EAX   := EAX + EDX
sub  esi,   eax     ; ESI   := ESI - EAX
```

The first instruction saves the result of addition to register EAX. The second instruction performs subtraction from register value of EAX from the ESI register. Two 2-to-1 ALU's are not able to perform these operations in parallel, hence one ALU has to use the result of second ALU.

The interlock collapsing ALU (ICALU) proposed in [31] eliminates dependency between two instructions, which require a binary or two's complement operation, without increasing the base cycle time. The described ICALU is able to perform 3-to-1 ALU operation. The 2-to-1 ALU can perform the first instruction: EAX = EAX + EDX. The 3-to-1 ALU is able to perform in parallel the second instruction: ESI = ESI - EAX', where is a result of preceding instruction: EAX' = EAX + EDX. The operation performed by the ICALU will be:

```
ESI = ESI - EAX - EDX
```

For this thesis we assume that only one 3-to-1 ALU is available. The design of 3-to-1 ALU must produce architecturally correct result and not stretch the machine cycle time. The following statements have been proven in the [31] for a set of 3-to-1 ALU operations:

- It is possible to design a 3-to-1 ALU, which is able to produce the correct results [31];
- A 3-to-1 design in comparison to 2-to-1 adder requires only one extra logic level , and one extra logic level will not stretch the machine clock cycle [31][32].

The implementation of 3-to-1 ALU can perform full two's complement ALU operations, and its detailed design is explained in [31].

Consider the next instruction sequence:

```
sub  eax,   edx     ; EAX   := EAX - EDX
add  eax,   eax     ; EAX   := EAX + EAX
```

The first instruction to be executed with 2-to-1 ALU : EAX = EAX - EDX. To execute both instructions in parallel, the second instruction require the following operation EAX = EAX' + EAX', where EAX' is the result of first instruction. The required operation will result in:

```
EAX = (EAX - EDX) + (EAX - EDX)
```

Since we assume that only 3-to-1 ALU is available, this instruction sequence will force a serial execution. In order to execute this sequence of instructions in parallel, a more complex 4-to-1 ALU is required.

### 2.2.2 Multiple Input Address Unit

Memory addressing of IA-32 in some cases can be very complicated. To calculate the effective address, depending on the addressing mode (Section 2.3.1.2), we need ModR/M and SIB byte. In most complex case, to resolve addresses there is one shift operation (scale) needed, for the index register, after an addition to the base register, and finally an addition to the immediate operand. For example:

```
mov eax, [2*esi+edi+ED28]
```

Figure 2.7 shows the instruction format of IA-32. Suppose there is a following instruction sequence:

```
add edi, 100
mov eax, [2*esi+edi+ED28]
```

In this example, the first instruction calculates the value of the register *edi*. The new value of register *edi* is used in the instruction *mov*, which is performed by the address unit (AU). AU performs calculation to resolve the address in the memory and loads the value from this memory location, given in the square brackets.

This sequence has an interlock because the *mov* instruction require the *edi* register. These two instructions can be compounded by the use of the five-port AU. The address unit with five inputs is able to perform the following calculation:

$$ADDRESS := ESI << 2 + EDI + 100 + ED28$$

To calculate this address a 4-input Arithmetic Unit is required and one shifter for the scale operand. Because shift (scale) operand is limited to only two bits, the scale operation can be done by a multiplexer, which has very low latency. The 4-to-1 Arithmetic Unit has delay of only two Carry Save Adders (CSA), and delay of CSA is equal to delay of an adder [23].

## 2.3 Intel Architecture

The Intel Architecture 32 (IA-32) is generally called x86, x86-32 or i386. IA-32 is a 32 bit extension of earlier 16-bit 8086, 80186 and 80286 processors. The first processor (Intel 80386) with 32 bit Intel Instruction Set Architecture (ISA) was introduced by Intel in 1985 [35]. Intel 80386 could correctly execute most of 16-bit Intel processor code. Due to the backward compatibility the x86 family microprocessors, IA-32 is the most widely used architecture in general-purpose computing systems.

Section 2.3.1 gives an overview of IA-32 ISA and architecture in respect to this thesis. Section 2.3.2 briefly describes the organization of the Itanium processor, the first Explicitly Parallel Instruction Computing (EPIC) processor implemented by Intel and HP. The EPIC has basis of research in VLIW [28]. Comparison of tagging between SCISM and IA-64 is given in Section 2.3.2.1.

### 2.3.1 IA-32

x86 architecture has eight 32-bit general-purpose integer registers. Figure 2.6 depicts these registers with description. It is noticeable that not all registers are entirely general-purpose. Special-purpose registers are the instruction pointer (EIP) and flag registers (EFLAGS).

IA-32 is a register-poor processor architecture. Hence, the chance that instructions have data hazards is very high and it is "ILP-poor" by its nature. The small amount of registers also had impact on the instruction decoding. Addressing of the register can be done by three bits and explained in the next section. It is most probably, that designers of this processor architecture were trying to save the memory on the opcodes.

Not only the ILP exploitation is hard by IA-32 due to the register-poor architecture. Memory addressing is also complex, which is explained in Section 2.3.1.2

| | |
|---|---|
| %eax | accumulator (for adding, multiplying, etc.) |
| %ebx | base (address of array in memory) |
| %ecx | count (of loop iterations) |
| %edx | data (e.g., second operand for binary operations) |
| %esi | source index (for string copy or array access) |
| %edi | destination index (for string copy or array access) |
| %ebp | base pointer (base of current stack frame) |
| %esp | stack pointer (top of stack) |
| | |
| %cs | code segment |
| %ds | data segment |
| %ss | stack segment |
| %es | extra segment (freely-useable) |
| %fs | extra segment (freely-useable) |
| %gs | extra segment (freely-useable) |
| | |
| %eip | instruction pointer (program counter) |
| %eflags | flags (condition codes and other things) |

| | | 8-bit | |
|---|---|---|---|
| **32-bit** | **16-bit** | **high** | **low** |
| EAX | AX | AH | AL |
| EBX | BX | BH | BL |
| ECX | CX | CH | CL |
| EDX | DX | DH | DL |
| ESI | DI | | |
| EDI | DI | | |
| EBP | BP | | |
| ESP | SP | | |

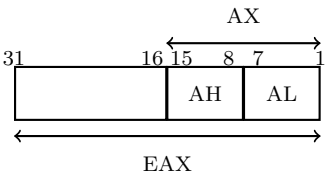


Figure 2.6: Intel Architecture 32 Registers

The last two general-purpose registers %ebx and %esp are used only for memory addressing. Other two registers %esi and %edi purpose is to address the arrays. Finally, there are only four real general-purpose registers.

#### 2.3.1.1 Instruction Format

Figure 2.7 shows the instruction format of IA-32. It is the longest typical instruction. The minimum instruction length is 1 byte long and contains the opcode only. The opcode field length can be only 8 or 16 bits long. x86 is a two-address machine, in other words an IA-32 instruction has two operands and both of the are sources and one is a destination, for instance: *add %eax, %edx*, where register *eax* is the source and the destination. After the *opcode* filed follows *mod r/m* field to specify the memory addressing. Some instruction use *mod r/m* field to select condition flags. The operation can have register, a memory location or immediate operand:

- register-register: *add %ebx, %edx*;
- register-memory: *add %eax, [%esp + 4]*;
- register-immediate: *cmp %edx, 10*;
- memory-register: *mov [%esp + 12], %eax*;
- memory-immediate: *mov [%esp + 12], 0* ;

The *s-i-b* field contains information about the memory-addressing mode by complex addressing modes. The last fields are optional displacement field, for address computation and/or immediate data. The length of last fields is dependent on the data-width mode (of a memory segment) and the size can be respectively 8-, 16 or 32-bits. The data-width mode can be overridden by a bit instruction (*w-bit*) or by a *prefix byte*. The prefix byte changes the instruction interpretation as address or operands size, default

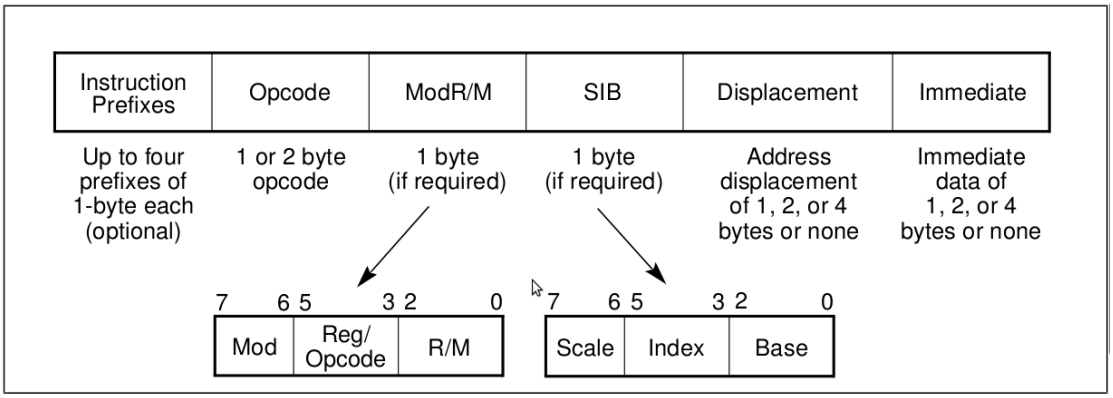


Figure 2.7: Intel Architecture Instruction Format

memory segment used by instruction, or indicate that instruction has to be executed with external bus locked.

Instruction length can vary from 8 to 104 bits (or longer in rare cases). The instruction length is a complex function of the opcode and related instruction fields. Some instruction fields specify the presents of other instruction fields, for example, the *mod r/m* and *s-i-b* both fields indicate the presents and length of the displacement field, moreover the prefix byte can change the address size and influence the displacement field. The length of the immediate-date fields can also be effected by other fields.

#### 2.3.1.2 Memory Accesses

In the IA-32 architecture require many operations to access the memory location because of the shortage of registers. To detect the dependencies in the memory access there are lot of calculations required. The x86 provides the extensive set of addressing modes to compute the effective address, after application of two levels of address translation to the effective address, by accessing segments and paging mechanism. For example, the double-indexed addressing mode uses three address operands and performs two additions to form the *effective address* . The effective address has to be added to the segment *base address* in a segment register. The base address is possible to override by the *prefix*. This mechanism allows to access the different segments. The addressing modes are shown as follows:

- Immediate
  MOV EAX,10 ; EAX = 10
- Direct
  MOV EAX,I ; EAX = Mem[&i]
- Register
  MOV EAX,EBX ; EAX = EBX
- Register indirect
  MOV EAX,[EBX] ; EAX = Memory[EBX]

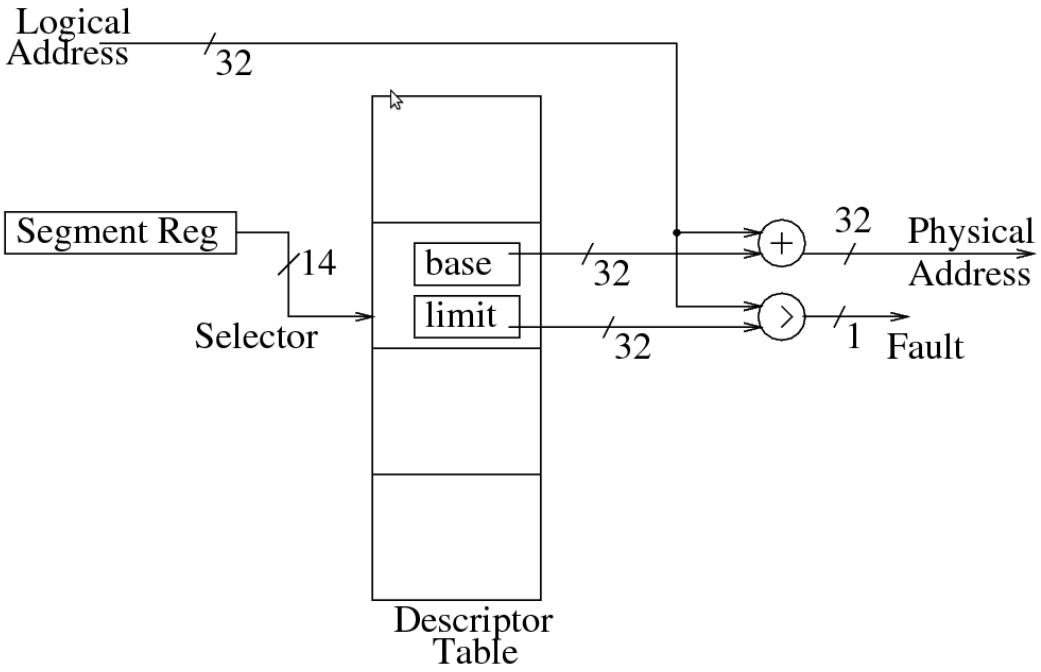


Figure 2.8: Memory protection with segmentation

- Based with 8- or 32-bit displacement
  MOV EAX,[EBX+8] ; EAX = Mem[EBX+8]

- Based with scaled index
  MOV EAX,ECX[EBX] ; EAX = Mem[EBX+$2^{scale}$ × ECX]

- Based plus scaled index with 8- or 32-bit displacement

The calculated (logical) address is has to be checked for the boundaries of the selected segment as shown in Figure 2.8. The segment selector points to the segment in the Global Descriptor Table (GDT), where the base and the limit are saved. If the calculated address within limits of the segment it is has to be translated by page translator to access the physical address. All this operations are required before a memory dependency can be checked.

The IA-32 allows the unaligned access to the memory of data either 8, 16, or 32 bits wide. An unaligned 32-bit value may span two memory locations. Depending on the crossing boundaries the addresses of these two memory locations can be different.

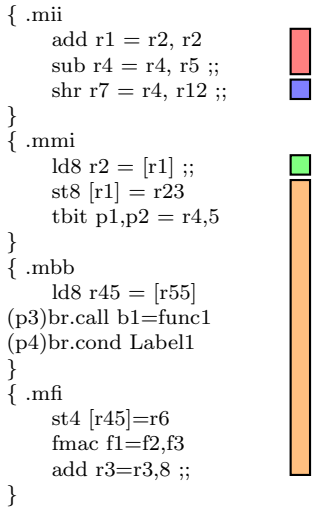


Figure 2.9: Example instruction groups

### 2.3.2 Itanium VLIW Properties

The Intel Itanium [18] architecture is designed to exploit the instruction-level parallelism (ILP) by means of Explicitly Parallel Instruction Computing (EPIC) [28]. The EPIC organization is derived from the VLIW machine architecture [11]. It is originated by Hewlett-Packard (HP) and later jointly developed by HP and Intel [34]. The Intel Itanium architecture is also referred as the IA-64 architecture.

To achieve high performance of IA-64 the developing team provided the "sufficient architectural capacity" [18]:

- full 64-bit address space;
- large register files;
- enough instruction bits to communicate information from compiler to the hardware;
- the ability to express arbitrarily large amount of ILP.

The main idea behind the design of Itanium processor that resolving the dependencies, reordering, and making compound instruction is on the compiler side. The hardware is not responsible for this task. IA-64 executes instructions in parallel, by grouping the instructions in *bundles*. Example of code is given in Figure 2.9. The shaded bars indicate bundles. Each compound instruction is terminated by double semicolon (;;).

IA-64 organization has 128 65-bit wide general-purpose registers. For encoding the registers instruction opcode contains 7-bit fields. Most of them have predicate register argument that requires 6-bit as shown in Figure 2.10(b). The grouped instruction, called bundle, is 128 bit long. The maximum number of instruction in each group is three. Each instruction is 41-bit with remaining 5 bits for template. These template bits

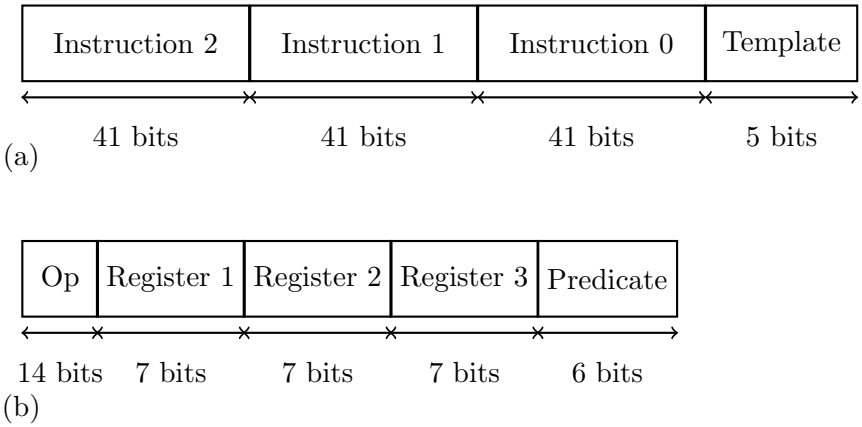


Figure 2.10: IA-64 bundle (a) and instruction (b) format

mark instruction for parallel execution and help decoding and routing of instructions (Figure 2.10(a)).

To avoid the performance lost on the branch the compiler technique branch *predication* is used. The predication is a way to execute the multiple conditional paths and drop the incorrect result when the branch is resolved. For example the following code:

```
if (r1 == r2)                cmp.eq p1, p2 = r1, r2
  r9 = r10 - r11;            (p1) sub   r9 = r10, r11
else
  r5 = r6 + r7;              (p2) add   r5 = r6, r7
```

The $p1$ and $p2$ predications guard the execution. If $p1$ is true instruction the result of instruction *sub* is hold, and the result of *add* instruction is dropped. In this way the compiler can help the processor in exploiting of the available ILP.

#### 2.3.2.1 The IA64 and SCISM Tagging

The IA64 instruction bundle (Figure 2.10) is reviewed in [14] in respect to SCISM compounding mechanism. Instruction categorization in SCISM is explained in Section 2.1.1 and the tagging of instruction in Section 2.1.2.

IA-64 machine executes bundles, as explained previous section. As shown in Figure 2.10, the template is a 5-bit wide field. The template is used for decoding, routing, and ILP. IA-64 instructions can be routed to specific *slots*. IA-64 has five instruction slot types - Memory (M), Integer (I), Floating-point (F), Branch (B), and Long Extended (L+X), corresponding to the functional execution units. The instructions are partitioned in six groups - ALU (A), Memory (M), Integer (I), Floating-point (F), Branch (B), and Long (L+X). Instruction in the bundle has 12 possible combinations (due the limited number of bits) of three instructions: MII, MI_I, MLX, MMI, M_MI, MFI, MMF, MIB, MBB, BBB, MMB and MFB. Underscore "_" indicates a stop in the

bundle of instructions, to tell to the processor which instruction to be executed sequentially in the budle. Instructions of type A, for example integer addition, can be executed to either I or M execution unit. Instruction type L+X require two instruction slots and executes on I-unit or on B-unit.

SCISM instruction partitioning on the other hand is done according to the hardware utilization (Section 2.1.1). For instance the category *one* of sixteen instruction categories, in evaluation of IBM System/370 [27], contains fourteen instructions and executed by the functional unit (ALU). Instruction decoding/tagging in the SCISM organization has cost of $log_2(n)$, where $n$ is a number of all executed instructions. Another important property of the SCISM tagging mechanism is that the tagging implies the full binary compatibility with legacy binary code. The SCISM machine is also able to branch in the middle of compound instruction allowing code compaction, shown in Figure 2.3, without a need to use *nops* and removing of branch alignment.

To investigate the effect of tagging on the code size the IA-64 instruction can be mapped onto SCISM [14]. For the demonstration it is chosen for straight-forward mapping. For fair comparison the three-way compounding scheme is assumed. The following main properties of the SCISM in this particular case are important:

- The SCISM organization is not restricted to *5-bits template* with only 24 possible instruction combinations;
- The three-way compounding require three tag bits for stop indication, saving on two of the five Itanium template bits;
- For routing to the functional units additional bits are required in the tag.

IA-64 instructions are executed by four execution units types: M, I, F or B. The last two IA-64 instruction types (A-type and X-type) are executed also by one of these execution units and this requires additional bits for instructions routing. Figure 2.11 shows the instruction format. The $T$ is a tag for parallel execution and two additional bits ($rr$) for routing. The total length of compound SCISM instruction becomes 132 bits: three 41 bits IA-64 instructions and three bits per instruction for SCISM tagging. The template bits are not required for the routing with this tagging approach.

By compiling the CPU2000 with default makefiles (level of optimization) it is observed that approximately one third of the IA-64 instruction are *nops* [14]. The reason behind it that compiler inserts the nop-instruction into the instruction sequence, when instructions have interlocks [21]. SCISM machine is not using nop-operations, hence tagging is preferred. The comparison between SCISM tagging by use of three tags it is found that the static code size can be reduced by 21%-27%. The original IA-64 size of compound instruction is 128 bits and of the SCISM straight-way compounding scheme is 132 bits.

## 2.4 SCISM on Intel Architecture

Over the years Intel and AMD added extensions to the IA-32 instructions set, e.g. MMX, SSE, 3DNow!, etc, which are intended to boost the performance of multimedia applica-

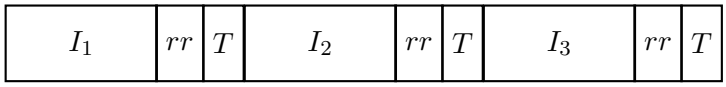


Figure 2.11: SCISM instruction format for IA-64

tions. These extensions are based on Single Instruction, Multiple Data (SIMD) paradigm, used to exploit Data-Level parallelism.

The limitations of 32-bit microprocessor, e.g. memory addressing limitations, and the need of 64-bit general-purpose registers, motivated to step to 64-bit CPU's. Two largest vendors of the Intel Architecture, Intel and AMD, have introduced their new 64-bit solutions. Table 2.1 shows the comparison between 64-bit microprocessor of Intel and AMD.

Table 2.1: Comparison of 64-bit microprocessors of Intel and AMD

| **AMD x86-64** | **Intel IA-64** |
|---|---|
| Fully x86 compatible. | Instruction sets are NOT x86 compatible. |
| Superscalar processor | ILP exploitation based on EPIC |
| No performance sacrifice for 32-bit | Separate x86 engine with low performance for 32-bit legacy code. |
| No recompilation | Code has to be recompiled for performance |

To achieve very high performance from ILP exploitation, Intel introduced the IA-64. IA-64 is derived from EPIC [28] and explained in Section 2.3.2. One of the disadvantages of IA-64 is that the instruction set is not compatible with IA-32. To compensate the incompatibility, the IA-64 Itanium processor is armed with a small x86 engine causes speed penalty on legacy code [3]. AMD has introduced AMD64, which extends the IA-32 architecture to 64 bit [3]. AMD64 stays fully compatible with the original IA-32, while IA-64 offers no native IA-32 compatibility. The advantages are, that 64-bit calculations are performed natively and this architecture provides bigger register file. Due the fact that AMD64 microprocessors are Superscalar, they still have challenges as high complexity of the instruction analysis, and limitation of ILP exploitation.

When considering designs of different 64-bit processors, a question arises, whether it is possible to exploit the maximum of ILP, keep microprocessor compatible with legacy code, and have 64-bit architecture. The SCISM microarchitecture achieves high performance [27] in such way, that designing a new architecture and recompilation of the existing applications can be avoided as we intend to prove on this thesis.

# SCISM Organization on IA-32 microarchitecture

# 3

*This chapter describes the approached SCISM organization on the IA-32 instruction set. First Section shows the compounding scheme from high level of abstraction. The assumption of the processor microarchitecture in terms of the functional units is described in the Section 3.2. Section 3.3 explains how IA-32 instructions are categorized for the compounding unit. The Compounding Rules, based on the categorization and hardware assumption, are described in Section 3.4.*

## 3.1 Compounding Scheme

The compound facility can be placed in the post-compiler, in main memory, or in the cache. Using the post-compiler it is possible to compound instructions in the VLIW form with tagging [27]. If instructions are analyzed on run-time, the preprocessing of instructions is performed by the Instruction Compounding Unit (ICU). In this thesis we assumed that the preprocessor is placed between the processor and the memory subsystem. With this approach we want to evaluate the performance gain on the level of the ILP exploitation, the number of instructions that are compounded. The behavior of the cache is not in the scope of this work.

To exploit as much ILP as possible, we assume that multiple functional units are available, for example, two ALU's, two shifter etc. We use a simple two-way compounding scheme, hence the number of functional units is limited to two. For instance, to execute two shift instructions in parallel, two shifters are needed. These instructions must be independent. We can execute two instructions, which perform two's arithmetic complement operations, using two ALU's. But when there is data dependency between instructions, instructions cannot be executed in parallel. If an interlock between instructions occurs, in certain cases it can be resolved by the interlock collapsing units. In Section 2.2 we explained how the Interlock Collapsing ALU (ICALU) is able to execute two "ALU-instructions" in parallel, even with the dependency.

The SCISM preprocessor is shown in the Figure 3.1. The prefetching unit decodes the instructions from the memory subsystem to the instruction buffer. The compounder analyses the instructions from the instruction buffer and places the tag for each instruction based on the compounding rules. The Fetch controller makes the request to the memory. After the instructions are analyzed by the compounder, instructions can be executed.

To achieve the maximum performance we assume that there is enough functional units and interlock collapsing units. We choose to implement the two-way compounding scheme, as we execute two instruction in parallel. The IA-32 SCISM machine contains

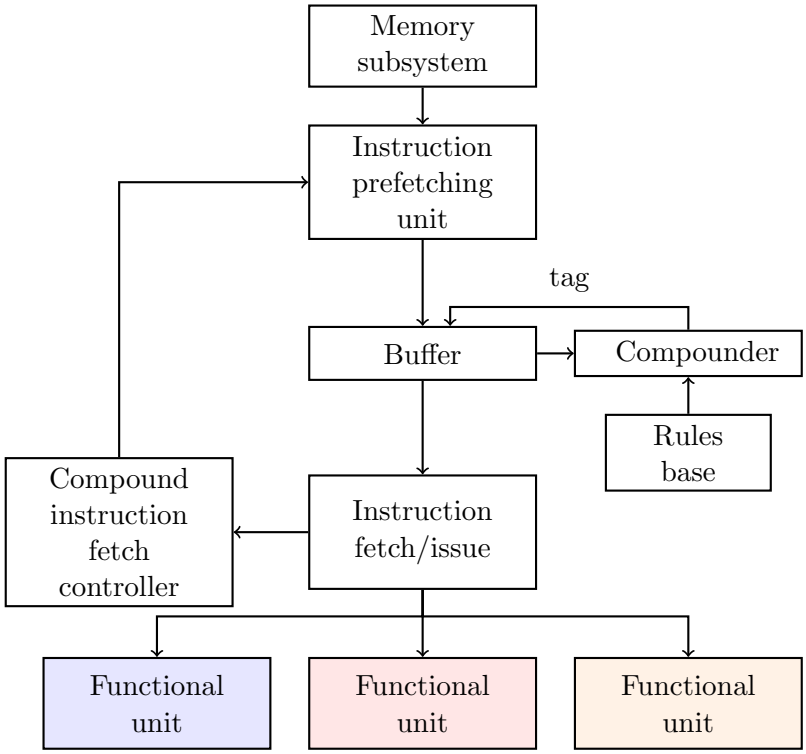


Figure 3.1: IA-32 SCISM design

the following functional units:

- One 2-to-1 ALU;
- One 3-to-1 ALU for dependency collapsing (Section 2.2.1);
- Two shifters;
- One three-input address unit (AU);
- One five-input address unit for dependency collapsing (AU) (Section 2.2.2);
- One two-port data cache (DC);
- One floating-point unit (FPU);
- One branch unit (BU);
- One multiplier;
- One divider.

The FP registers in the IA-32 microarchitecture are organized as a stack. Due to the stack nature , which is last in/first out (LIFO), it is not possible to execute two

FP instructions in parallel. An additional FPU would not provide any performance improvement. However, even in applications performing floating-point calculations, all kinds of instructions are intermingled. It implies that instructions of different classes or categories can be executed in parallel. Instructions, performing integer operations, can be executed in parallel with the FP instructions. This is because FPU instructions use the FP register file while integer instructions use the integer register file.

## 3.2 Functional Units

In this section we will survey the functional units. As all Superscalar processors the SCISM organization requires multiple execution units. Further, to eliminate certain interlocks the multi-operand and interlock collapsing units are used.

In this thesis assume a two-way compounding scheme, where the maximum amount of instructions to execute in parallel is two.

### 3.2.1 ALU's

Assume the following sequence of the IA-32 instruction:

```
sub  edx,   ebx     ; EDX'  := EDX - EBX
add  eax,   edx     ; EAX   := EAX - EDX'
```

The first instruction calculates the value of EDX by performing a subtraction value of EBX register from EDX register, and second one uses this value for the substraction. Using the 2-to-1 ALU's instructions have to be executed sequentially. Using the interlock collapsing 3-to-1 ALU (Section 2.2) together with 2-to-ALU the processor are able to execute these two instructions in parallel:

```
EDX'  := EDX - EBX
EAX   := EAX + EDX' = EAX + EDX - EBX
```

The first instruction is to be performed by the conventional ALU, and the second instruction is to be performed by 3-to-1 ALU at the same time.

However, in the one of examples given in Section 2.2.1 of instructions sequence, if we want to compound them, the 4-to-1 ALU is required. Because only 3-to-1 interlock collapsing ALU is assumed, instructions, like in this example, are not compounded.

### 3.2.2 Shifters

To compound as many instructions as possible we also assume that there are two shift units. Two shift instructions will be compound only if these are not dependent.

### 3.2.3 Address Units

To compound two memory access instructions, we also assume that second AU is available. Consider the following instruction sequence:

```
mov ds:[1024], ebx
mov eax, [esi+edi)
```

where the first instruction store the *ebx* register to the memory location at $ds + 1024$, and second instruction loads a value to *eax* register from $eax + edi$ location.

### 3.2.4 Two-port Data Cache

Approximately one third of instructions are memory operations. To exploit the ILP from instructions with a memory access it is required to use multi-port data cache. Due the demand to access multiple memory location in the Superscalar machines, there are some implementations of such cache as proposed in [25]. The assumption is, that the x86 SCISM organization issues two instructions in maximum per machine cycle, and there is a requirement of dual-port data cache.

### 3.2.5 Floating-Point Unit and its limitations

The architecture of floating point operations of IA-32 machine was not defined with a Superscalar implementation in mind. The Pentium and Pentium Pro processor can issue both, floating-point, and an integer instructions in very restricted circumstances [30]. Unlike many RISC ISA's with a *flat* floating-point register file, the IA-32 contains eight 80-bit registers as a *stack*. Stack organization will not allow to execute two FP-instructions in parallel, because of the LILO structure. However, the stack organization is suitable for the certain algorithms, e.g. speech recognition [30].

To write data to floating-point stack a *push* instruction is used. The *push* operation reads data from the stack. The items are not accessible until all other items above are removed, also called *FILO* (first in, last out) data structure. However there is a microarchitectural hack allowing to swap a top element with any other element in the stack to access the item programmer need. An instruction looks as:

```
fxch ST(2)
```

In this example instruction swaps the top element (0) with an element (2). The *ST* stands for *stack top* and points to the top element of the stack. This hack allow to perform an operation on the value, which is not an the top of the stack. Unfortunately it is not giving as any possibility to perform two FP operations in parallel.

### 3.2.6 Branch Unit

Branches occur roughly each four-five instructions in general-purpose programs. Until the branch is taken, the processor can not resolve the next instruction pointer. In order to diminish the negative effect on exploitation of ILP, the branch prediction schemes are introduced. There is a big variety of mechanism to predict a branch. The simplest branch-prediction mechanism is a branch prediction buffer or branch history table.

A branch history table is simplest sort of buffer, and constitute a self of a small memory, containing a bit(s), that say whether the branch was taken last time or not. For addressing of the table the lower portion of the branch instruction address is used.

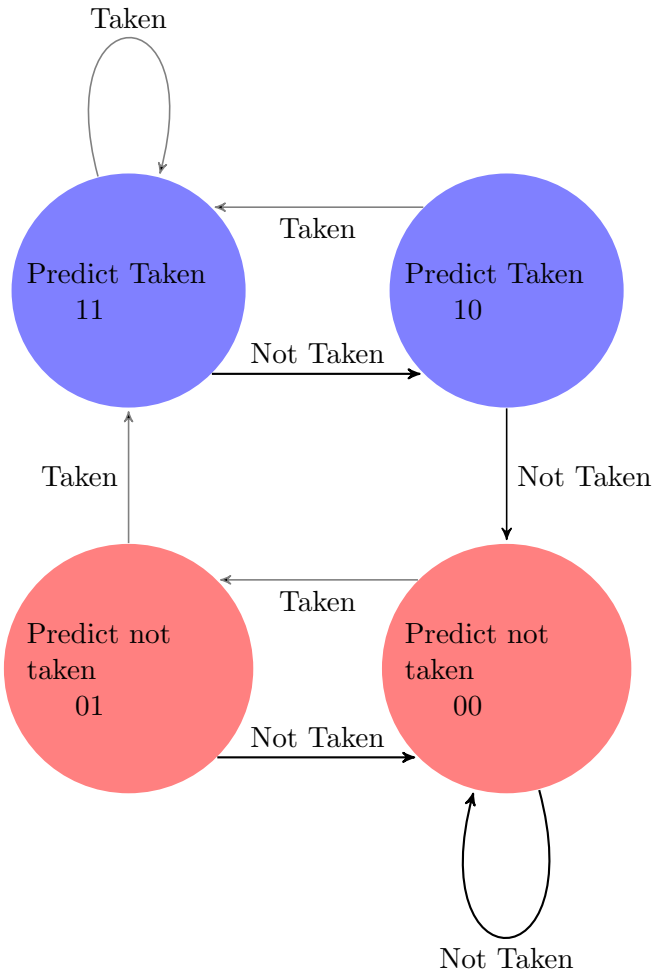


Figure 3.2: The state machine of two-bit branch prediction scheme

Because the lower bits of branch instruction addresses are used, it is not possible to say if the next prediction is correct, and this prediction refers to the same branch instruction form last time.

Figure 3.2 illustrates the state machine of two-bit prediction scheme. In this case branch that strongly taken or not-taken will be misspredicted less often that with only one bit. In general all entries of the table are initialized as taken.

## 3.3 IA-32 Instructions Categorization

The IA-32 is a Register-Memory (R/M) architecture. It is implies that this architecture can execute both: register-register operations, and memory-register operations. For example instruction *add* in the x86 organization can have both operands as a register, or one as a register and second in the memory. The memory operand is addressed by

registers, and the value has to be retrieved from memory subsystem by the addressing unit (AU).

The main property of the SCISM organization is that the instructions categorization is based on the hardware utilization. For instance, instructions add, sub, logical or, logical and, etc. utilize the ALU. Instruction *JMP* utilizes the branch unit to resolve the next address for the instruction pointer. Arithmetic shift instructions (rol, ror, sar, shl, shr) are executed by the shifter.

Instructions with R/M addressing utilize more that one functional unit. Also, all instructions with same combination of functional units need to be partitioned in one category. For example, instruction *LOOP* perform a loop operation using the ECX or CX register as counter. This instruction (1) decrements the count register, (2) checks the register if it is not 0, (3) and performs a jump if needed. These three operation are performed on the ALU (1), (2) and BU (3).

IA-32 instructions are divided into 18 categories (we will also refer as groups) according to the hardware utilization. All categories are listed in the Table 3.1. The IA-32 instruction set is explained in more details in [5].

Instructions of the first Category are register-register (RR) operations and they all utilize the ALU's. A RR-instruction, performs operation using the values of two registers, and save the result to the destination register.

Category 2 contains the shift-instructions of RR and register-memory (RM) format. Depending on the opcode the *count* operand may be an immediate or a value from the CL register. The source operand is either memory location, or register.

Category 3 contains conditional branches. These instructions branch if a count register is 0, with exception that instruction *LOOP* that also decrements the count register. Instruction from Category 4 and 5 utilize the BU. Branch on condition Category contain 62 opcodes [5]. This instruction checks the status flags from EFLAGS register and performs the jump if the condition is met. For instance instruction *JA* performs the jump if carry flag (CF) and zero flag (ZF) are 0, and instruction *JC* if the CF is 1. Instruction *JMP* is performed unconditionally by the branch unit. The offset of jump, dependently on the opcode, can be an immediate value or a register, or memory location. Category 4 contains only jumps with an immediate offset.

The store(Category 6) and load(Category 6) operation in the IA-32 are performed by instructions *mov, push, and pop*. To write a value to a memory location instruction *mov* is used, while to store value to the stack, operation *push* is performed. Instruction *mov* has the following addressing modes:

- Direct addressing mode: *mov ds:[disp],cx*.
- Register indirect addressing mode: *mov [di], bx*.
- Base addressing mode: *mov [di + disp], bx*.
- Indexed addressing mode: *mov cs:disp[si], al*.

In examples of addressing mode the following registers are used *ds, cx, di, bx, si, al* and displacement *disp*. Even if the addressing modes in the IA-32 look complex,

we consider that it is "trivial" for hardware to resolve right registers. For the index addressing mode, for example, we have to resolve two registers, for base and index, displacement and the scaling from the opcode.

Instruction *push* uses the stack base register (SP) to find location on top of the stack and decrements the stack pointer according to the size of the stored to stack. To pop the value from the stack instruction *pop* is used.

Instructions from Category 8 perform the multiplication and from Category 9 the division. We assume that our SCISM IA-32 machine has a divider and a multiplier. The action of this instruction and the the registers are depended on the opcode and the size, as follows:

| **Operands size** | **Source 1** | **Source 2** | **Destination** |
|---|---|---|---|
| Byte | AL | r/m8 | AX |
| Word | AX | r/m16 | DX:AX |
| Double word | EAX | r/m32 | EDX:EAX |

Where the AL, AX, EAX are general-purpose registers and *r/m8-16-32* is a general-purpose register or a memory location. The memory location is specified by the addressing mode, discussed earlier. This category of instruction use the multiplier and AU.

Instruction of the Category 9 use the divider and the AU. The division instruction use the following operands:

| **Operands size** | **Dividend** | **Divisor** | **Quotient** | **Remainder** |
|---|---|---|---|---|
| Word/Byte | AL | r/m8 | AL | AH |
| Double Word/word | DX:AX | r/m16 | AX | DX |
| Quad Word/ Double word | EDX:EAX | r/m32 | EAX | EDX |

Next Category is 10 and utilizes the ALU and the AU. Instructions of this Category require source from the memory, the second source and the destination is a register. After the operation is performed, the result saved in the destination register. This instruction performs a load and an ALU operation. Category 11 contains instructions, which utilize the same functional units and the EFLAG register. The reasoning to place instruction will be discussed in the following section. Category 12 instructions use the shifter and carry flag from the EFLAG register.

The instruction *LEA* of Category 13 computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address, specified with one of the processors addressing modes. The destination operand is a general-purpose register.

Instruction for manipulating the flags as carry flag, interrupt flag, and direction flag and not requiring a special functional unit are placed in the Category 14.

Instruction for the floating-point operations are placed in the Category 15 and 16. In the Category 15 instructions are only in the register-register format. Category 16 are instruction of a register-memory format.

Instruction is read-modify-write (RMW) format are in the Category 17. The RMW instruction loads the value from a memory, makes the operation, and stores the result

Table 3.1: IA-32 instruction set partitioned in categories

| **Cat.** | **Description** | **Instructions** | **Recourses** |
|---|---|---|---|
| 1 | RR-format load, logicals, arithmetics, compares | ADD, AND, CMP, DEC, INC, NEG, NOT, OR, SUB, TEST, XOR, MOV | ALU |
| 2 | RM-format shifts | ROL, ROR, SAR, SHL, SHR | SHIFTER, AU |
| 3 | Branch on count and index | LOOP, JCXZ | BU, ALU |
| 4 | Branch on condition FLAGS | Jxx | BU |
| 5 | Branch with no condition, offset is immediate | JMP imm | BU |
| 6 | Stores | MOV, PUSH | AU, DC |
| 7 | Loads | MOV, POP | AU, DC |
| 8 | RM-format multiplication | MUL, IMUL | Multiplier |
| 9 | RM-format divisions | DIV, IDIV | Divider |
| 10 | RM-format arithmetic, logics | ADD, AND, CMP, OR, SUB, TEST, XOR (XADD) | ALU, AU |
| 11 | RM-format ALU with CF | ADC, SBB | ALU, EFLAG |
| 12 | RM-format shifts with CF | RCL, RCR | SHIFTER, EFLAG |
| 13 | Load address | LEA | AU |
| 14 | Control | CLC, CLD, CLI, CMC, STC, STI, STD | |
| 15 | RR-format floating-point | FLD_STi, FST_STi, FLD1, FLDL2T, FLDL2E, FLDPI, FLDLG2, FLDLN2, FLDZ, FADD_ST0_STj, FADD_STi_ST0, FMUL_ST0_STj, FMUL_STi_ST0, FSUB_ST0_STj, FSUBR_ST0_STj, FSUB_STi_ST0, FSUBR_STi_ST0, FDIV_ST0_STj, FDIVR_ST0_STj, FDIV_STi_ST0, FDIVR_STi_ST0, FCOM_STi, FUCOM_STi, FCOMPP, FUCOMPP, FXCH_STi, FPLEGACY, FCHS, FABS, FTST, FXAM, FDECSTP, FINCSTP, FFREE_STi, FFREEP_STi, F2XM1, FYL2X, FSCALE, + All FP Trigonometric instr. | FP UNIT |
| 16 | RM-format floating-point | FLD, FILD, FBLD_PACKED_BCD, FST, FBSTP_PACKED_BCD, FISTTP16, FISTTP32, FISTTP64, FNINIT, FNCLEX, FRSTOR, FNSAVE, FLDENV, FNSTENV, FLDCW, FNSTCW,FNSTSW, FNSTSW_AX, FADD, FIADD, FMUL, FIMUL, FSUB, FISUB, FDIV, FIDIV, FCOM, FICOM, FCMOV_ST0_STj | FP UNIT, AU |
| 17 | RMW-format logicals, arithmetics, shifts | ADD, AND, CMP, DEC, INC, NEG, NOT, OR, SUB, TEST, XOR,ROL, ROR, SAR, SHL, SHR, ADC, SBB, RCL, RCR | ALU, SH, AU |
| 18 | All other instructions | Various | Various |

to the same memory location. This category has instructions utilizes the ALU, Shifter, and AU.

All remaining instruction are lumped to the last Category 18.

## 3.4 Compounding Rules

The instruction compounding unit (ICU) analyses the instructions on the base of the compounding rules. The summary of the compounding rules is given in Table 3.2. By finding the Category of the first instruction and across the Category of the second instruction, we indicate how the two instructions compound or not compound. For example, the rule **Y** indicates that instructions can always compound (they can not have dependency or dependency can be collapsed). Instruction which never compound are indicated with **N**. The case when instruction can compound only if no dependency is available, is noticed by **I**. When parallel execution can collapse an execution dependency but not an address dependency is noticed by **E**, or **A** if instructions can collapse the address dependency but not an execution dependency.

Table 3.2: IA-32 compound rules

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Y | A | Y | Y | Y | A | Y | A | A | Y | Y | N | Y | Y | Y | A | N | N |
| 2 | I | I | I | Y | Y | I | I | I | I | I | N | N | I | Y | Y | E | N | N |
| 3 | Y | I | N | N | N | I | Y | I | I | Y | Y | N | Y | Y | Y | Y | N | N |
| 4 | Y | Y | N | N | N | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | N |
| 5 | Y | Y | N | N | N | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | N |
| 6 | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | N | N |
| 7 | I | I | I | Y | Y | I | I | I | I | E | E | I | I | Y | Y | I | N | N |
| 8 | I | I | I | Y | Y | I | I | N | I | I | N | N | I | Y | Y | I | N | N |
| 9 | I | I | I | Y | Y | I | I | I | N | I | N | N | I | Y | Y | I | N | N |
| 10 | Y | I | E | Y | Y | I | I | I | I | E | E | N | I | Y | Y | I | N | N |
| 11 | Y | A | E | Y | Y | I | I | I | I | E | E | N | I | Y | Y | I | N | N |
| 12 | I | I | I | Y | Y | I | I | I | I | E | N | N | I | Y | Y | I | N | N |
| 13 | Y | I | I | Y | Y | E | I | I | I | E | E | I | I | Y | Y | I | N | N |
| 14 | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | N |
| 15 | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | N | N | N | N |
| 16 | I | I | I | Y | Y | I | I | I | I | I | I | I | I | Y | N | N | N | N |
| 17 | N | N | N | Y | Y | N | N | N | N | N | N | N | N | Y | N | N | N | N |
| 18 | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N |

To understand the Table 3.2, consider compounding two instructions of Category 1. The **Y** indicates that can always compound on the available hardware. However, in the following instructions sequence, instruction are not compoundable:

```
sub  eax,   edx ; EAX := EAX - EDX
add  eax,   eax ; EAX := EAX' + EAX' = (EAX-EDX) + (EAX-EDX)
```

The second instruction *add* requires four inputs, and the assumed ICALU design is able to perform only 3-to-1 operation.

Two instructions of Category 2 can compound only if they have no dependency, as two shifters are not able to collapse the dependency. Instructions of Category 2 and Category 12 cannot compound, because instruction of Category 12 require a carry-flag. Two instructions of Category 1 and Category 11 can compound because the carry flag can be handled by the interlock-collapsing ALU.

In case of **A** rule, like Category 1 and Category 2, it means that instructions are compounded if there is an address dependency. This is possible to collapse with the five-input AU. If these two instructions have no memory interlock, instruction must be independent. Category 2 and Category 1 can compound only if independent, due to the fact that there is not shift-add interlock collapsing unit. Category 1 and a floating point R-M operation from Category 16 are compounded if there is an address generation dependency or independent.

Instruction which can be compound only with execution dependency, like Category 7 and Category 10, require a dual-port cache. This situation is not conflicting with the our hardware assumption.

# 4 Implementation

*In this chapter first Section describes the Bocsh emulator used for the implementation of SCISM compounding unit. Section 4.1 shows the internal structure of the emulator and preparation. In the Section 4.2 we describe the compounding facility implemented on the emulator. In the Section 4.3 we discuss the limitation of Bochs The "magic instruction" required to perform experiments is described in Section 4.4.*

## 4.1 Bochs x86 Emulator

For the analysis of SCISM machine performance, applied to IA-32 organization, we need an *emulator* of an IA-32 processor. It emulates (duplicates) the behavior of a computer architecture. A software emulator should be able to execute programs compiled for the simulated ISA. For instance, we can execute programs compiled for x86 on the ARM processor. A simulator only attempts to reproduce the program behavior, while an emulator attempts to model of various degrees the state of the device being emulated. An emulator consists of three modules:

- a CPU emulator or CPU simulator (the two terms are mostly interchangeable in this case);
- a memory subsystem module;
- various I/O devices emulators;

We want to use the IA-32 emulator to implement a compounding facility, based on the compounding rules from Section 3.4. It is suitable to use the full-system simulation, which also model the Operation System (OS) overhead for completeness of results.

For this thesis we chose an open source emulator, due the fact that the source of a commercial simulation software is not publicly available. There are two candidates: Qemu [4] and Bochs [20]. Qemu supports a big variation of hardware platforms, for example x86, ARM, Alpha, MIPS etc. Bochs is a portable IA-32 and IA-64 PC emulator and debugger, written in C++. It is free software under GNUL Lesser General Public License [13] as Qemu. Bochs emulates different IA-32 CPU's, memory, disk, display. There is a big variety of guest operation systems. The host operating systems can be Linux, Windows or Mac OS X.

Qemu uses a dynamic binary translation to port the binary code of the emulated system. By translating the original code Qemu splits a target CPU instruction in its own *micro-operations* and uses the translated code to emulate the system. The main

advantage of the dynamic code translation is its high performance. To implement a compounder, it is necessary to buffer instructions, and after that analyze the instructions. Qemu fetches instruction and directly matches them to its own instructions. Because Qemu actually runs its own micro-code, instead of straightforward emulation, we are not interested in this emulator. Bochs, on the other hand, emulates each instruction. Consider, for example, the following IA-32 instruction:

```
add eax, 0x10
```

where an instruction adds the immediate value $0x10$ to register EAX and stores the result into register EAX. After instruction is decoded, the execution of instruction is performed by the following code written in C++:

```
void BX_CPP_AttrRegparmN(1) BX_CPU_C::ADD_EAXId(bxInstruction_c *i)
{
  Bit32u op1_32, op2_32 = i->Id(), sum_32;

  op1_32 = EAX;
  sum_32 = op1_32 + op2_32;
  RAX = sum_32;

  SET_FLAGS_OSZAPC_ADD_32(op1_32, op2_32, sum_32);
}
```

All instructions are written in a similar fashion.

### 4.1.1 Preparation

For this project we choose the Intel Pentium processor emulation by Bochs (2.3.7 release). Bochs is build with setting listed in the Table 4.1.

| Option | Value | Comments |
|---|---|---|
| CPU level | 5 | Choices are 3,4,5,6 which mean to target 386, 486, Pentium, or Pentium Pro and later emulation. |
| FPU enabled | yes | to make use of the FPU emulator. |
| Trace cache enabled | no | support instruction trace cache for faster execution. |
| Enable host-specific-asms | yes | support for running native x86 instructions on an x86 machine. |
| Enable MMX | yes | Add support for MMX instruction. |

Table 4.1: The emulated CPU on the Bochs

To run the benchmarks, emulators require an operating system or some kind of firmware, while the simulators does not. For this project we have selected on of the smallest Linux live-CD from Finnix [10] (Finnix 2.6.26-1-x86 release). The reason to choose this distribution because is that it supports the Executable and Linkable Format (ELF) [2].

Bochs is able to use the disk images. We can copy the benchmarks on the image to run them on the Bochs. A *flat* disk image is used for placing there the benchmarks programs and created using Bochs's *bximage* utility:

```
bximage -hd -mode=flat -size=200000 hd_cpu2006_X.img
```

## 4.2 Compounding Facility on Bochs

Each instruction is fetched from the memory system and executed in the infinite CPU-loop. Figure 4.1 (a) depicts the execution flow of this loop. At the first stage CPU loop checks for asynchronous external events, e.g. interrupts. During the prefetch stage instruction is checked whether it is in the segment boundary and the instruction pointer on the host system is resolved. In next stage, if an instruction contains memory reference, the effective address of an instruction is calculated. The instruction is executed by the instruction's execution method (indirect call dispatch).

By providing the compounding facility to the emulation software we can analyze the instruction-stream of the test-programs (benchmarks). The compounding facility has been described in Section 2.1.2 and it consists of the following items:

- Instruction buffer;
- Compounder;
- Compounding Rules;
- Branch Target Buffer (BTB).

For this work we modified the CPU loop by adding the instruction buffer and the compounding facility, as shown in Figure 4.1 (b). During the Fetch-to-buffer stage instructions are fetched to the instruction buffer. In the compounding stage instruction are analyzed according to the compounding rules (Section 3.4). Instruction in the buffer has a tag and other important elements, e.g. the instruction pointer (RIP), physical address:

```
struct compound_bxInstruction_c_t{
  bxInstruction_c  i;
  bx_address rip;
  unsigned address;
  unsigned tag;
}
```

For this project the size of instruction buffer is eight. Choosing smaller buffer, e.g. four instructions, can limit the number of compounded instructions. For example, if the first instruction and the second one are compounded, and third instruction cannot be compounded with fourth instruction, there is possibility that instruction four can be compounded with instruction five.???

The instruction buffer is made in form of a circular array. If the first instruction is executed, the next cycle emulator executes the second instruction, and first instruction

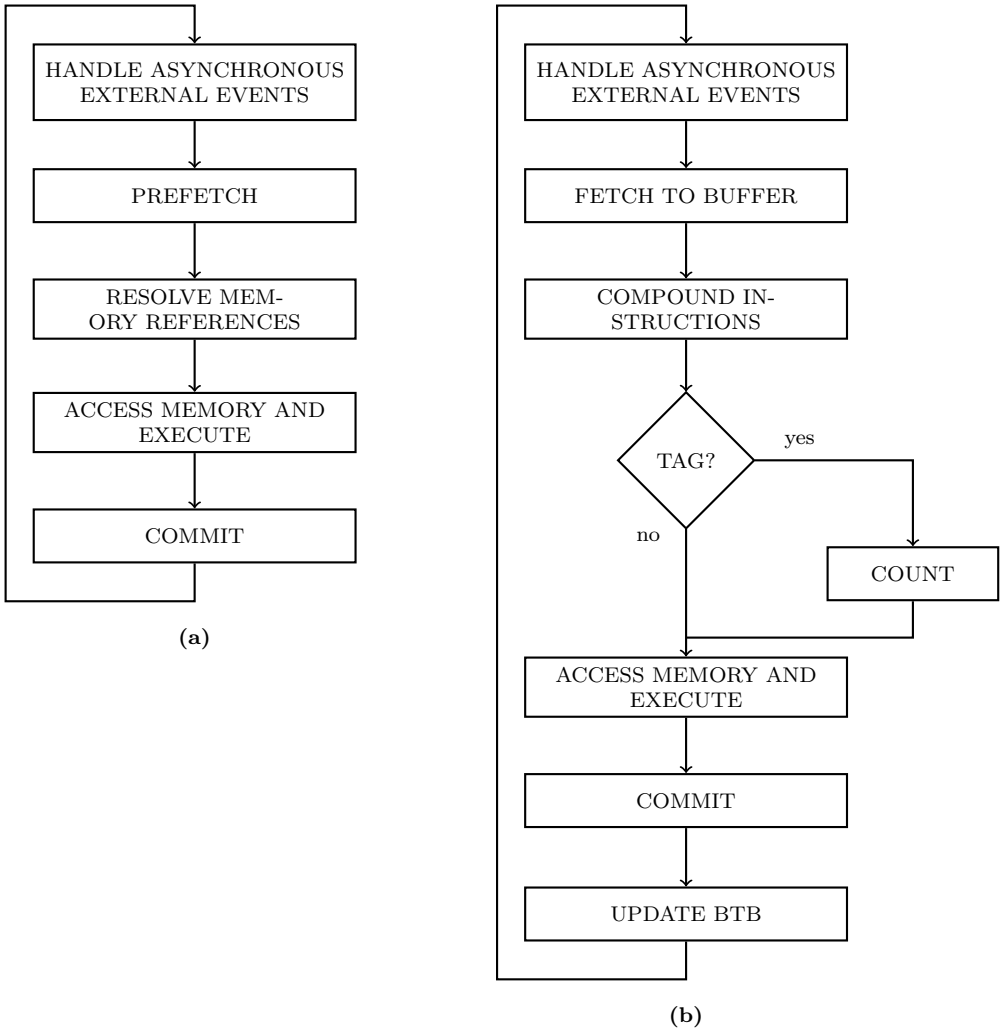


Figure 4.1: The CPU loop in Bochs (a) and (b) with compounding facility

($k$) can be replaced with upcoming instruction. For example the buffer with eight instructions, from $I_k$ to $I_{k+7}$ with eight prefetched instruction. Instruction $I_k$ is executed and next instruction to be executed is $I_{k+1}$. The place of $I_k$ can be used by $I_{k+8}$. In this way we try to keep the buffer as full as possible. If it is not possible to resolve the instruction pointer due to unresolved jump, we stop prefetching. The calculation of instruction pointer is done as follows:

$$IP_{k+8} = IP_{k+7} + i- > len()_{k+7}$$

where the IP is an instruction pointer and $i- > len()$ is the length of instruction in bytes.

The instruction compounding stage follows after the instruction buffer stage. During this stage each time two instructions are checked whether they can be executed in parallel,

based on the compounding rules. The compounding rules are implemented in the lookup array. For example, two instruction of Category 2 can be compound only if there is no dependency, as it is described in the Section 3.4.

```
static const unsigned Compound_Rules[18][18]
```

The IA-32 instructions decoding is very complex. To determine if there is dependency between the instructions we need to resolve the registers of both instructions by looking up in the instruction description table:

```
static const compound_instr_prop compoundInstructProperty[1086] = {
..
{ CMPD_GR1,  CMPD_REG_A_BYTE, CMPD_IMM,      CMPD_TWO_OP, "ADD_ALIb" },
{ CMPD_GR1,  CMPD_REG_A,      CMPD_IMM,      CMPD_TWO_OP, "ADD_AXIw" },
{ CMPD_GR1,  CMPD_REG_A,      CMPD_IMM,      CMPD_TWO_OP, "ADD_EAXId"},
{ CMPD_GR17, CMPD_MEM,        CMPD_NNN_BYTE, CMPD_TWO_OP, "ADD_EbGbM"},

..
}
```

The first member of this structure is the Category (or group) number. The second and third member describe the destination and the source, respectively. Fourth element is the number of operands of an instruction. The last element is the name of instruction in the Bochs environment. This information helps to retrieve the registers and the Category for the dependency resolving between a two instructions in the compounding facility.

The compounder modifies the tag to 1 for first instruction and 2 to second, when two instructions are compounded. After the next instruction pair can be analyzed. If a pair of instructions is not compounded, the next pair of instruction is checked. For example when $I_k$ and $I_{k+1}$ are compound and $I_{k+2}$ and $I_{k+3}$ not, we want to check if the combination $I_{k+3}$ and $I_{k+4}$ is compoundable.

After compounding stage, it necessary to detect whether instructions are compounded for the benchmarking. When the instruction tag is "1" we count this instruction as compound. After, instruction is executed, the CPU loop handle the asynchronous events only after the second instruction is executed.

## 4.3 Limitations of the Emulator

Although Bochs is suitable for this thesis, there are still some limitations. First, Bochs is not a cycle accurate because, it an instruction set emulator. The behavior of the simulator looks very like non-pipelined CPU like Motorolla 68000 or Intel 8086 - every instruction is fetched, decoded, executed, and committed in cycle. For example a load instruction require more clock cycles than an ALU instruction, because of the latency of the main memory, for example due to the cache-misses.

Second limitation is that the use of the benchmarks is complicated. In order to run the benchmarks, an operation system (OS) is required, which executes millions of

instruction, until the OS is booted. To indicate, how many instruction are compounded for a benchmark application, it is necessary to use the start and the stop signal, before and after application is executed. For this project we use a *magic instruction* described in the next section.

## 4.4 Magic Instruction

In order to evaluate the performance of a IA-32 microarchitecture with compounding facility we want to measure how many instruction are executed in parallel. Similar to other emulators Bochs only emulates a hardware architecture. It is not possible to run only the benchmark for the evaluation of its behavior without an operation system (OS). This makes it difficult to distinguish the running benchmark and the OS code. To evaluate a benchmark program we need to start the compounding facility, and when benchmark programs is finished we need to stop compounding and report the results.

To overcome this problem, we introduce a *magic instruction*, which is able to start and stop the compounding facility. This instruction is to be placed in the source program. When a magic instruction is executed Bochs detects it and starts counting compound instructions.

There are a number of opcodes in the IA-32 instruction set which are *undefined opcodes*. The undefined opcodes are several reserved opcodes, but never implemented. If the undefined opcode is detected by the processor it gives the *Illegal Instruction* error. One of the undefined opcodes is *0F 04*. After adding this instruction to the instruction list of the Bochs by the chosen opcode we are able to use it as an start and stop condition:

```
static const BxOpcodeInfo_t BxOpcodeInfo32R[512*2] = {
 ..
#if COMPOUND_UNIT == 1
  /* 0F 04 /dr */ { 0, BX_IA_ICU},
#else
  /* 0F 04 /dr */ { 0, BX_IA_ERROR },
#endif
 ..
 }
```

The **BxOpcodeInfo32R** is one of lookup tables in Bochs environment for resolving of the correct execution method for a fetched opcode. Bochs has instruction fetch procedure, which is quite complex, and with use of several lookup tables it is able to obtain the instruction and it's properties from the opcode. The *0F 04* opcode is changed from an error (**BX_IA_ERROR** ) to the *BX_IA_ICU* method,

Now, when Bochs is able to detect the condition to start and stop the compounding facility, we need to add the magic instruction to the source of benchmarks. The benchmark programs need to be compiled after the magic instruction is added. Due to the fact, that instruction does not exists in the ISA, compiler will not recognize it. To solve this problem, we added instruction *"ICU"* to the GAS (GNU Assembler) [1] and compiled it. Now we have the object file:

```
void magic_instruction(void) {
        asm("icu");
}
```

The function *magic_instruction* can be added to any source code by linking the *magic_instruction.o* file to the source. For example in the *401.bzip2* file:

```
..
#include "magic_instruction.h"
..

void spec_compress(int in, int out, int lev) {
    blockSize100k           = lev;
  magic_instruction();
    compressStream ( in, out );
  magic_instruction();
}
```

However, it is difficult to find the suitable place in all of benchmark the program, where to put magic instruction. Thus, we compiled an *icu* executable, which executes only this instruction. For some programs we run this executable before and after the program, to start and to stop the compounder facility.

# 5 Experimental Results

*The previous two chapters describe the approach and the implementation of the compounding scheme. This chapter describes the experimental setup and presents the obtained results. Section 5.1 gives an overview on the benchmark suite used for the evaluation. The obtained results from benchmarking are presented and analyzed in Section 5.2.*

## 5.1 Benchmarking Methodology

To evaluate the CPU performance it is common to run tests using standard benchmarks. By running a set of programs multiple times we can determine the performance of a CPU. To evaluate the SCISM organization on the IA-32 processor we use Standard Performance Evaluation Corporation (SPEC) CPU2006 [6] benchmarks. SPEC is a non-profit organization with the goal to establish and maintain a standardized set of benchmark programs. Their benchmarks are widely used today for performance evaluation.

The source code of the SPEC CPU2006 suite is based on real user applications. It includes, for example, file compression, video decoding, artificial intelligence, scientific application etc. The CPU2006 suite consists of integer and floating point benchmarks, listed in Tables 5.1 and 5.2. In the considered microarchitecture, SCISM cannot provide any benefit for FP instructions due to the stack nature of the FP register file. However, FP instructions are often intermingled with integer instructions, which can benefit from SCISM. Hence we compound both integer and FP instructions.

Table 5.1: Integer Component of SPEC CPU2006

| **Benchmark** | **Language** | **Application Area** |
|---|---|---|
| 400.perlbench | C | Programming Language |
| 401.bzip2 | C | Data Compression |
| 403.gcc | C | C Compiler |
| 429.mcf | C | Combinatorial Optimization |
| 445.gobmk | C | Artificial Intelligence: Go |
| 456.hmmer | C | Search Gene Sequence |
| 458.sjeng | C | Artificial Intelligence: chess |
| 462.libquantum | C | Physics / Quantum Computing |
| 464.h264ref | C | Video Compression |
| 471.omnetpp | C++ | Discrete Event Simulation |
| 473.astar | C++ | Path-finding Algorithms |

For the evaluation of the SCISM organization we do not use a tools set, provided by SPEC CPU2006, which compiles, runs, validates and reports on the benchmark mea-

Table 5.2: Floating Point Component of SPEC CPU2006

| **Benchmark** | **Language** | **Application Area** |
|---|---|---|
| 410.bwaves | Fortran | Fluid Dynamics |
| 416.gamess | Fortran | Quantum Chemistry. |
| 433.milc | C | Physics / Quantum Chromodynamics |
| 434.zeusmp | Fortran | Physics / CFD |
| 435.gromacs | C,Fortran | Biochemistry / Molecular Dynamics |
| 436.cactusADM | C,Fortran | Physics / General Relativity |
| 437.leslie3d | Fortran | Fluid Dynamics |
| 444.namd | C++ | Biology / Molecular Dynamics |
| 447.dealII | C++ | Finite Element Analysis |
| 450.soplex | C++ | Linear Programming, Optimization |
| 453.povray | C++ | Image Ray-tracing |
| 454.calculix | C, Fortran | Structural Mechanics |
| 459.GemsFDTD | Fortran | Computational Electromagnetics |
| 465.tonto | Fortran | Quantum Chemistry |
| 470.lbm | C | Fluid Dynamics |
| 481.wrf | C, Fortran | Weather & Weather modeling |
| 482.sphinx3 | C | Speech recognition |

sure. We used only the source code of the benchmark programs and the provided input files [24].

We compiled the benchmarks using the *gcc-3.4* and *gfortran-4.3*. To run the benchmark programs on the Bochs emulator we use the Linux live-CD from Finnix distribution [10]. A Linux distribution is a software distribution build on Linux kernel and consists of software application. The Finnix distribution has limited number of dynamically linked shared object libraries, and therefore the benchmarks are compiled to static binaries.

## 5.2 Experimental Results

The SCICM compounding facility implemented on the Bochs emulator is used to run the SPEC CPU2006 benchmarks. Figure 5.1 shows the obtained results for the integer benchmarks. For every benchmark it shows the number of original instructions and the number of instruction with compounding facility. The amount of compounded instructions varies between 13% and 25%. In other words, there are up to 25% of instructions which could be executed in parallel. The results also include the overhead of the operation system, described in previous section. Hence his test is very closed to the realistic case.

The application 471.omnet has the lowest ratio of compounded instructions. This application simulates a discrete event. It is very likely that there are dependencies between instructions, due to the fact that each state of a discrete event uses the values of the previous state. The highest result is 25%, observed in 401.bzip2 (file compression) and 471.h264 applications (video and audio decoding). These applications are computation intensive by nature, what can explain the highest amount of compounded instructions in comparison to others.

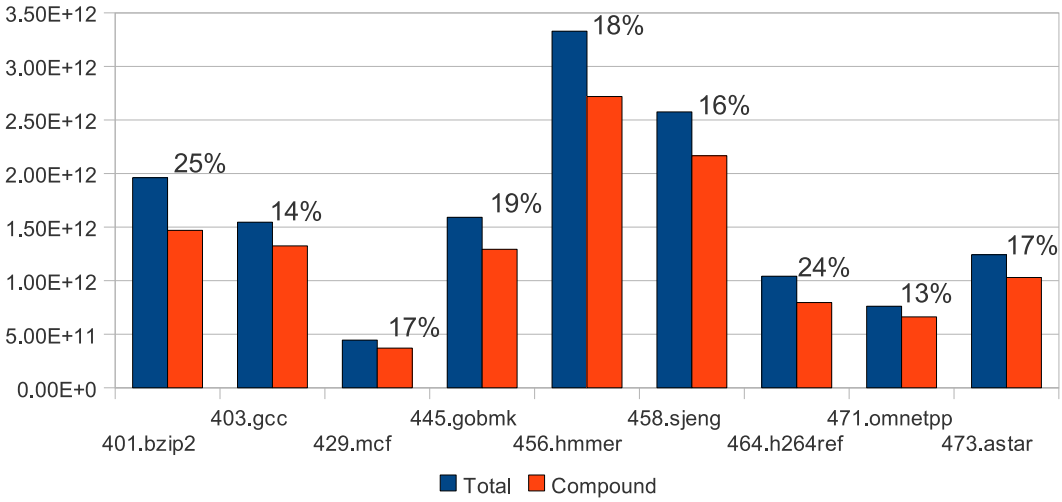


Figure 5.1: Number of original instructions and compounded instructions

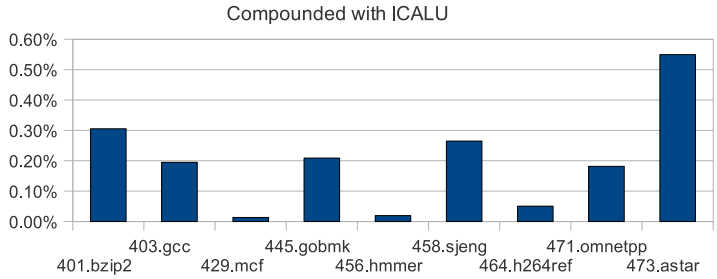


Figure 5.2: Number of compounded instructions with ICALU

The number of compounded instructions, which use the ICALU is shown in Figure 5.2. It varies between 0.01% and 0.55%. The lowest amount of "ICALU-instructions" is observed in 429.mcf application (Combinatorial optimization / Single-depot vehicle scheduling). The 473.astar application has the highest amount of instructions, which are using the ICALU (Computer games. Artificial Intelligence. Path finding.). This chart indicates only how many instructions are compounded, if the ICALU is used. It is hard to define which combination (ICALU with ALU or a pair of ALU's) is more efficient to choose in terms of performance and area.

Figure 5.3 shows the number of compounded instructions among the different categories. According to this chart the majority of compounded instructions are from Category 1, Category 4, Category 6, and Category 7. Instructions which are never compounded or very rare are from the following categories:

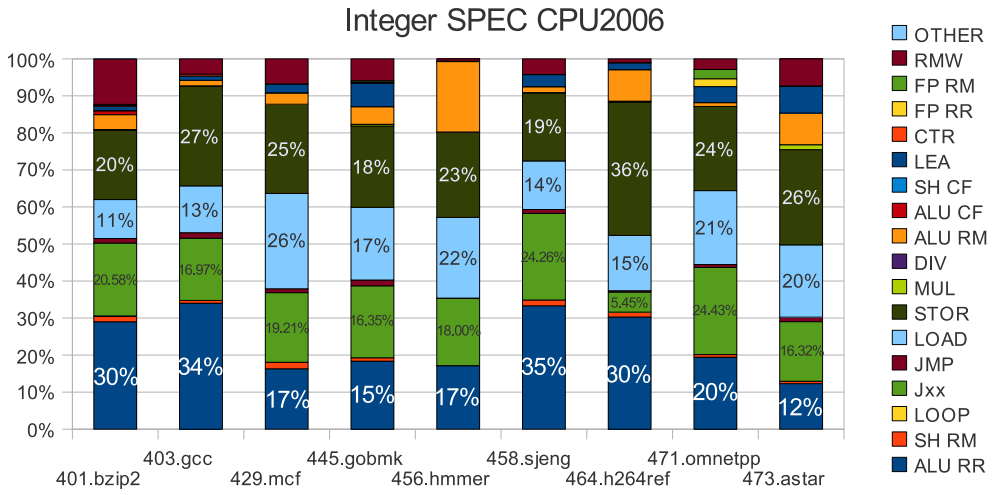


Figure 5.3: Categories of compounded instructions for integer benchmarks

- Category 3 (Instruction LOOP);
- Category 5 (Unconditional branch instruction);
- Category 8 (Multiplication instructions);
- Category 9 (Division instructions);
- Category 12 (Shift with carry instructions);
- Category 14 (Control instructions);
- Category 15 (Floating point RR-format instructions);
- Category 16 (Floating point RM-format instructions).

The fact that FP instructions are rare is not surprising, because this benchmark part performs integer operations. Instruction *LOOP* is not used often by the compilers, usually the conditional jumps are used. Integer Division and Multiplication instructions use three to four registers. It is most probable that these instructions have execution interlocks, because these ones require more general-purpose registers than other instructions, for example, integer operations. Control instructions, which manipulate the processor flags are rare, as it is shown in Figure 5.4.

The average number of executed instructions with and without compounding facility for each category is shown in Figure 5.4 for all benchmark applications. According to this chart instructions from the Category 7, Category 1, Category 4 and Category 6 are executed in most. These instruction utilize the ALU, AU and branch unit (BU). It is remarkable that there are a big amount of conditional jumps. The amount of executed instructions of the Categories 8, 9, 11, 12, and 14 is very low. We consider, that instructions can be lumped to the Category 18. Category 18, containing all other instructions,

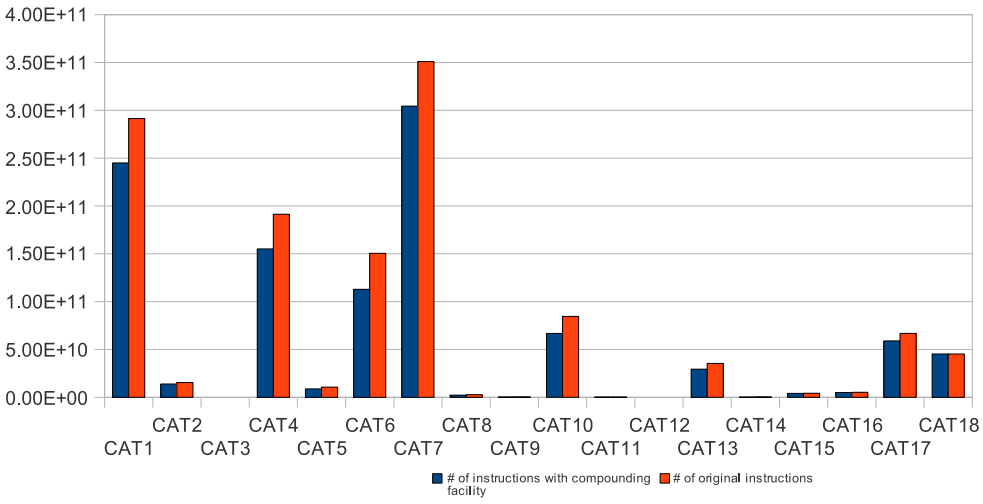


Figure 5.4: Average number of compounded instructions for each category

is approximately 3% of all executed instructions. This will make compounding facility even less complex.

IA-32 architecture is register-poor. Hence, the instructions have a lot of data hazards. Also, there is lack of registers to keep the variables. In order to solve this problem, compilers use the memory subsystem to save the some temporary values, as it is depicted in Figure 5.4. Category 7 and Category 6 contain instructions which load and store values to the memory. These instructions are executed in most, according to the results.

Table 5.3 gives an overview of compounded instructions by the categories in 401.bzip2. The meaning of this table is explained in the Section 3.4, which determines if two instructions can be compounded, and on what condition. About 24% of compounded instructions are of Category 4 and Category 17. Category 17 are Read-Modify-Write (RMW) instructions, utilizing the ALU. Instructions from Category 4 are only compoundable with RMW. Almost 18% of compounded instructions are of Category 1 and Category 1. Category 6 and Category 7 in all four combinations (6-6, 6-7, 7-6, 7-7) in total have approximately 17% of compounded instructions. The multi-port AU units and multi-port cache make possible to execute this combination of instructions.

Figure 5.5 depicts the performance gain of the floating-point benchmarks. The amount of compounded instructions varies between 1% and 26%. The largest number of compounded instructions is in the 410.bwaves application, which numerically simulates blast waves. The lowest ratio of compounded instructions is in the 470.lbm application, which implements the so-called "Lattice Boltzmann Method" (LBM) to simulate incompressible fluids in 3D. The binary code of applications, which perform floating point calculations, contain not only FP-instructions. Instructions of all classes are intermingled. The compounding rules, depicted in Section 3.4, allow to execute FP-instructions together with other instructions.

Figure 5.6 shows that, except the instructions which are utilizing the ALU and AU,

Table 5.3: Compounding among the categories in 401.bzip application

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | **17.6%** | 0.0% | 0.0% | **1.7%** | **0.1%** | **6.1%** | **2.0%** | 0.0% | 0.0% | **0.1%** | 0.0% | 0.0% | **0.3%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **2** | **0.9%** | **0.4%** | 0.0% | 0.0% | 0.0% | **0.3%** | **0.6%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **3** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **4** | **8.0%** | 0.0% | 0.0% | 0.0% | 0.0% | **1.1%** | 0.0% | 0.0% | 0.0% | **1.0%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | **24.1%** | 0.0% |
| **5** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | **0.2%** | **0.7%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **6** | 0.0% | 0.0% | 0.0% | **0.2%** | 0.0% | **1.8%** | **3.0%** | 0.0% | 0.0% | **0.6%** | 0.0% | 0.0% | **0.5%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **7** | **6.0%** | **0.5%** | 0.0% | **4.7%** | **0.2%** | **5.3%** | **6.9%** | 0.0% | 0.0% | **1.5%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **8** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **9** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **10** | **0.3%** | 0.0% | 0.0% | **0.3%** | 0.0% | **0.5%** | **0.5%** | 0.0% | 0.0% | **1.1%** | 0.0% | 0.0% | **0.3%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **11** | 0.0% | 0.0% | 0.0% | 0.0% | **0.8%** | 0.0% | 0.0% | **0.1%** | 0.0% | **0.3%** | **0.1%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **12** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **13** | 0.0% | 0.0% | 0.0% | **0.1%** | 0.0% | **0.5%** | **0.2%** | 0.0% | 0.0% | **0.4%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **14** | 0.0% | 0.0% | 0.0% | 0.0% | **0.3%** | 0.0% | 0.0% | 0.0% | 0.0% | **0.1%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **15** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **16** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | **0.1%** | **0.2%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **17** | 0.0% | 0.0% | 0.0% | 0.0% | **0.2%** | 0.0% | 0.0% | **0.3%** | 0.0% | **0.4%** | **0.5%** | 0.0% | 0.0% | 0.1% | 0.0% | 0.0% | 0.0% | 0.0% |
| **18** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |

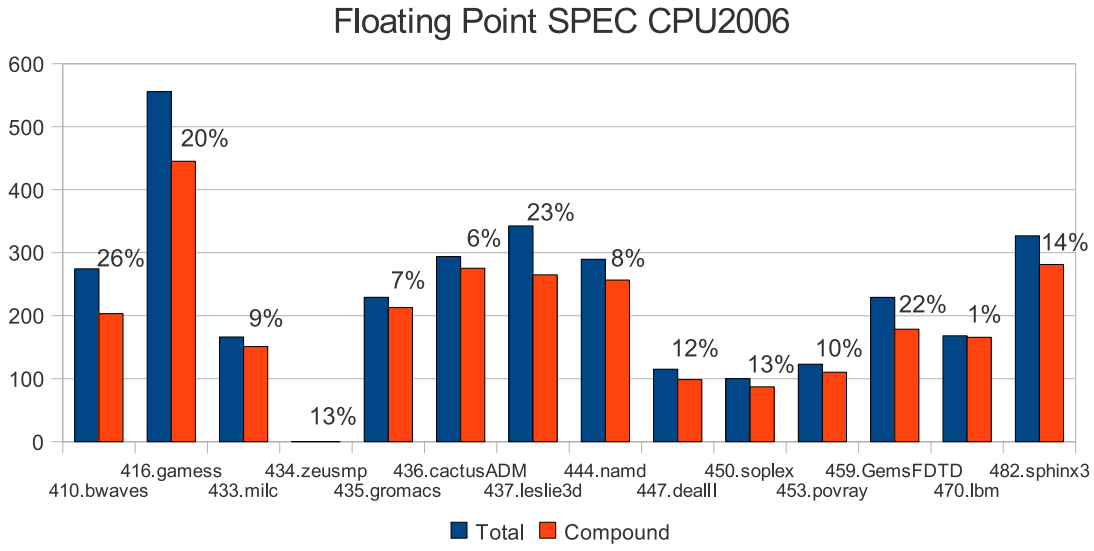


Figure 5.5: Number of original instructions and compounded instructions$\times 10^{12}$

there are many instructions of Category 15 (FP register-register) and Category 16 (FP register-memory). Instructions which never compounded or very rare are:

- Category 3 (Instruction LOOP)
- Category 5 (Unconditional branch instruction)
- Category 8 (Multiplication instructions)
- Category 9 (Division instructions)
- Category 11 (ALU-RM instruction with carry-flag)
- Category 12 (Shift with carry instructions)

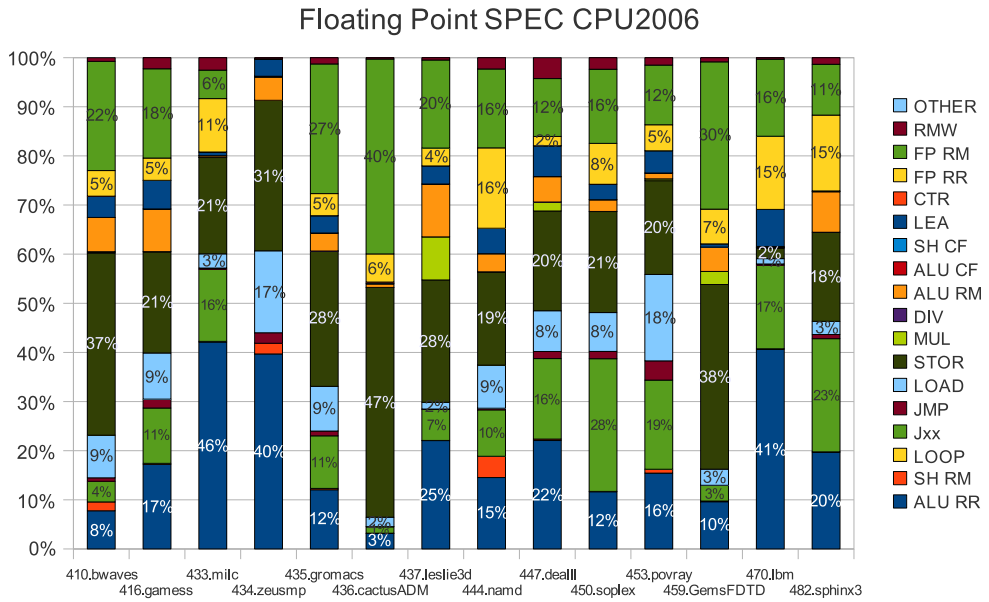


Figure 5.6: Categories of compounded instructions (FP)

- Category 14 (Control instructions)

According to the results obtained from the integer and floating-point benchmarks certain categories, which are executed very rare, are almost the same.

Table 5.4: Compounding among the categories in 410.bwaves application

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | **1.0%** | 0.0% | 0.0% | **0.4%** | 0.0% | **0.9%** | **0.1%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | **0.2%** | 0.0% | **0.2%** | **10.1%** | 0.0% | 0.0% |
| **2** | **0.1%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.8% | **2.7%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **3** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **4** | **0.1%** | 0.0% | 0.0% | 0.0% | 0.0% | **1.0%** | 0.0% | 0.0% | 0.0% | **4.2%** | 0.0% | 0.0% | 0.0% | 0.0% | **0.6%** | 0.0% | **0.5%** | 0.0% |
| **5** | **0.1%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | **1.0%** | 0.0% |
| **6** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | **0.9%** | **4.8%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **7** | **0.2%** | 0.0% | 0.0% | **1.5%** | **0.1%** | **7.4%** | **3.7%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | **5.9%** | 0.0% | **0.3%** | **32.9%** | 0.0% | 0.0% |
| **8** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | **0.2%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **9** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **10** | **0.4%** | 0.0% | 0.0% | **0.1%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | **4.5%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **11** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **12** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **13** | **0.2%** | 0.0% | 0.0% | 0.0% | **0.2%** | **0.5%** | **1.0%** | **0.2%** | 0.0% | 0.0% | 0.0% | 0.0% | **0.2%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **14** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **15** | **0.5%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | **8.7%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **16** | **0.1%** | 0.0% | 0.0% | **0.1%** | 0.0% | 0.0% | **0.9%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | **0.1%** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **17** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **18** | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |

Table 5.4 shows the number of instructions compounded for combinations among the categories in 410.bwaves application. As with the integer part of SPEC benchmarks, we see that instructions of Category 1, Category 6, and Category 7 are compounded in total about 19%. Also the instructions of Categories 15 and 16 which utilize floating-point unit and combination of FP unit with AU, are compounded significantly. To compound instructions of Category 1 and Category 16, the instructions must have the address dependency or no dependency, because the general-purpose registers are used

for addressing in a floating point R/M instructions. It implies that the multi-port AU unit is needed to eliminate the dependency and execute instructions in parallel, if such interlock occurs.

# 6 Conclusions and Future Work

*In this chapter we describe the results achieved by the SCISM instruction compounding scheme and draw the conclusions (Section 6.1). In Section 6.2 we discuss the future work.*

## 6.1 Conclusions

The Superscalar and VLIW machines are CPU microarchitectures designed to exploit ILP. Intel has developed the Itanium processor based on IA-64 architecture. This architecture is based on EPIC computer paradigm, which has its roots in the VLIW[28]. The performance of this processor is considered to be adequate for the floating-point computations. However, to execute widely used legacy IA-32 code, the Itanium processor is equipped with a small x86 (IA-32) hardware engine, that has low performance. Simultaneously AMD designed its 64-bit architecture as an extension of the IA-32 architecture. It is fully IA-32 compatible Superscalar machine without any performance sacrifice (even significant improvements in some cases). However, there is a number of serious challenges that still remains:

- in both approaches the number of instructions for parallel execution is limited;
- very often the true dependencies between instructions force sequential execution;
- stepping to the new VLIW instruction set requires recompilation of existing applications (sometimes the source code might not be available).

To address the above problems of VLIW and Superscalar architectures and provide compatibility with the IA-32 ISA, the SCISM [27] organization can be used. In this thesis we evaluate the SCISM compounding facility applied to the IA-32 architecture. The SCISM organization analyzes instructions for parallel execution according to a categorization based on the hardware utilization. Categorizing instructions in a such way simplifies the analyzing hardware, in contrast to the categorization based purely on opcode descriptions. Because SCISM organization is not dependent on the targeted ISA, it is possible to apply it to any existing processors. This fact also implies that SCISM machine stays compatible with the legacy binary code.

The use of interlock collapsing hardware eliminates certain interlocks. In this thesis we assumed that ICALU, multi-port cache, and multi-port AU are available to resolve interlocks between the instructions in certain cases. Correctness of such ALU has been proven in [31].

In this thesis we have analyzed the IA-32 instruction set and divided the instructions into categories. Based on the categorization we have defined and implemented the compounding rules. The Bochs x86 emulator was extended with a compounding facility. In order to estimate the performance gain we used the widely accepted SPEC CPU2006 benchmarking suite. Experimental results show that 13% to 25% of instructions can be executed in parallel for the integer workloads of SPEC CPU2006. For the floating-point programs the number of compounded instructions varies between 1% and 26%. Most instructions which are compounded utilize the two ALU's, two address units (AU) or a combination of the different functional units, for example FPU and AU, FPU and ALU etc.

Some instruction combinations are never or rarely compounded, for instance LOOP, unconditional branches, multiplication, division, shift-with-carry, and flag-instruction. This implies that these categories are excessive. Using less categories will result in lower complexity of the compounding facility.

In this thesis we have analyzed very simple two-way in-order compounding scheme. Due to the lack of registers on the IA-32 microarchitecture, a lot of instructions have true data hazards. However, even in this simple scenario the SCISM organization shows reasonable performance gain.

## 6.2 Future Work

In this thesis we used the SCISM two-way compounding scheme with in-order execution for IA-32 ISA. Experimental results show that up to one quarter of instructions can be executed in parallel in widely used SPEC CPU2006 integer and floating point workloads. It will be interesting to implement a three- or four-way compounding scheme and analyze whether the SCISM organization can deliver additional performance gain on IA-32. It is also interesting analyze the out-of-order compounding and execution.

The emulator used for this work is not cycle accurate and it is not possible to report the performance gain in real machine cycles. Cycle accurate emulator is envisioned as helpful to analyze how the pipeline and the cache will behave if the compounding facility is used.

As it is has been noticed in the previous section, some instruction categories are never compounded. This concludes that not all instruction categories are equally important for the targeted workloads. Additional code analysis can help to find out which instructions are almost never used in the general-purpose programs, which can simplify the compounding rules.

Superscalar machines use the wider dispatch/issue windows for the parallels execution. The trace cache accelerates execution [26] by saving decoded instructions to the small memory on the processor. For example, this is implemented on Pentium 4 [16] processor. We are also interested if the tagging can be applied with the trace cache for the instruction preprocessing on the SCISM machine.

Furthermore, it could be interesting to compare the 64-bit extension of IA-32 with SCISM compounding functionality and the IA-64 microarchitecture in terms of the Power/Area/Performance metrics. First, it is required to estimate the growth of additional functional units used in the SCISM machine. It is possible to use the a known

and simple processor, for example, Intel 486. Secondly, we can use the AMD64 processor as reference point to predict the power and area costs of an envisioned 64-bit Intel SCISM machine.

# Bibliography


[1] *Gnu binutils*, `http://www.gnu.org/software/binutils`.

[2] *Tool interface standard (tis) executable and linking format (elf) specification*, http://x86.ddj.com/ftp/manuals/tools/elf.pdf, May 1995.

[3] AMD Corporation, *x86-64$^{tm}$ technology white paper*, Tech. report, AMD Corporation, One AMD Place, Sunnyvale, CA 94088, USA, August 2000.

[4] Fabrice Bellard, *Qemu, open source processor emulator (home page)*, `http://www.qemu.org`.

[5] Intel Corporation, *The ia-32 intel(r) architecture software developer's and volume 2: Instruction set reference*, 2003.

[6] Standard Performance Evaluation Corporation, *Spec cpu2006(home page)*, `http://www.spec.org/cpu2006/`.

[7] S. D. Cotofana and S. Vassiliadis, *On the design complexity of the issue logic of superscalar machines*, Proc. of 24th EUROMICRO Conf., August 1998, pp. 277–284.

[8] Adrian Cristal, Josep Llosa, Mateo Valero, and Daniel Ortega, *Future ilp processors*, Int. J. High Perform. Comput. Netw. **2** (2004), no. 1, 1–10.

[9] Vassiliadis S. Eickemeyer, R. J. and B. Blaner, *An in-memory preprocessor for scism instruction-level parallel processors*, (1992), 16.

[10] Ryan Finnie, *Finnix(home page)*, `http://www.finnix.org`.

[11] Joseph A. Fisher, *Very long instruction word architectures and the eli-512*, ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture (New York, NY, USA), ACM, 1983, pp. 140–150.

[12] Michael J. Flynn, *Computer architecture: Pipelined and parallel processor design*, Jones and Bartlett Publishers, Inc., USA, 1995.

[13] Free Software Foundation, *Gnu lesser general public*, 2009.

[14] G. N. Gaydadjiev and S. Vassiliadis, *Scism versus ia-64 tagging: Differences and code density effects*, Proceedings of 10th International Euro-Par Conference, August 2004, pp. 571–577.

[15] John Hennessy and David Patterson, *Computer architecture - a quantitative approach*, Morgan Kaufmann, 2003.

[16] G. Hinton, M. Upton, D.J. Sager, D. Boggs, D.M. Carmean, P. Roussel, T.I. Chappell, T.D. Fletcher, M.S. Milshtein, M. Sprague, S. Samaan, and R. Murray, *A 0.18-m cmos ia-32 processor with a 4-ghz integer execution unit*, Solid-State Circuits, IEEE Journal of **36** (2001), no. 11, 1617–1627.

[17] Robert W. Horst, Richard L. Harris, and Robert L. Jardine, *Multiple instruction issue in the nonstop cyclone processor*, SIGARCH Comput. Archit. News **18** (1990), no. 3a, 216–226.

[18] Jerry Huck, Dale Morris, Jonathan Ross, Allan Knies, Hans Mulder, and Rumi Zahir, *Introducing the ia-64 architecture*, IEEE Micro **20** (2000), no. 5, 12–23.

[19] Mike Johnson, *Superscalar microprocessor design*, Prentice Hall series in innovative technology, Prentice Hall, 1991.

[20] Kevin Lawton, *The open source ia-32 emulation project (home page)*, `http://bochs.sourceforge.net`.

[21] J. Liu, B. Bell, and T. Truon, *Analysis and characterization of intel itanium instruction bundles for improving vliw processor performance*, Computer and Computational Sciences, 2006. IMSCCS '06. First International Multi-Symposiums on, vol. 1, June 2006, pp. 389–396.

[22] David W. Wall Norman P. Jouppi, *Available instruction-level parallelism for superscalar and superpipelined machines*, ACM SIGARCH Computer Architecture News **17** (1989), no. 2, 272 – 282.

[23] Behrooz Parhami, *Computer arithmetic: Algorithms and hardware designs*, New York : Oxford University Press, 2000.

[24] Vijay Janapa Reddi, *Spec cpu2006 commands*, 2009, [Online; accessed 22-September-2009].

[25] Jude A. Rivers, Gary S. Tyson, Edward S. Davidson, and Todd M. Austin, *On high-bandwidth data cache design for multi-issue processors*, MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture (Washington, DC, USA), IEEE Computer Society, 1997, pp. 46–56.

[26] Eric Rotenberg, Jim Smith, and Steve Bennett, *Trace cache: a low latency approach to high bandwidth instruction fetching*, Microarchitecture, IEEE/ACM International Symposium on **0** (1996), 24.

[27] R J Eickemeyer S Vassiliadis, B Blaner, *Scism: A scalable compound instruction set machine*, IBM Journal of Research and Development **38** (1994), no. 1, 59 – 78.

[28] M.S. Schlansker and B.R. Rau, *Epic: Explicitly parallel instruction computing*, Computer **33** (2000), no. 2, 37–45.

[29] James E. Smith and Gurindar S. Sohi, *The microarchitecture of superscalar processors*, 1995.

[30] Jon Stokes, *Inside the machine: An illustrated introduction to microprocessors and computer architecture*, No Starch Press, San Francisco, CA, USA, 2006.

[31] S. Vassiliadis, J. Phillips, and B. Blaner, *Interlock collapsing alu's*, IEEE Trans. Comput. **42** (1993), no. 7, 825–839.

[32] S. Vassiliadis and J. E. Phillips, *High performance interlock collapsing scism alu apparatus*, (1994).

[33] Stamatis Vassiliadis, Bart Blaner, and Richard J. Eickemeyer, *On the attributes of the scism organization*, SIGARCH Comput. Archit. News **20** (1992), no. 4, 44–53.

[34] Wikipedia, *Explicitly parallel instruction computing — wikipedia, the free encyclopedia*, 2009, [Online; accessed 24-October-2009].

[35] _____, *Intel 80386 — wikipedia, the free encyclopedia*, 2009, [Online; accessed 27-October-2009].

# Curriculum Vitae

**Eduard Gabdulkhakov** was born in Birsk, USSR, on the 24th of May in 1981. From 1993 to 1998 he did his secondary education at the Lyceum in Neftekamsk.

From 1998 to 1999 he studied in the Moscow State University of Food Industry.

In the year 2006 he graduated at Hogeschool van Arnhem en Nijmegen (HAN) in Arnhem as bachelor of ICT in informatics (Technische Informatica). In the same year he enrolled as master student into the MSc program of the Computer Engineering department of the Delft University of Technology. In the November 2008 he started the thesis at the Computer Engineering group with Georgi N. Gaydadjiev as his advisor.