

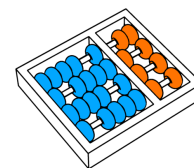


Bruno Cardoso Lopes

“Design and evaluation of compact ISAs”

“Estudo e avaliação de conjuntos de instruções compactos”

CAMPINAS
2014



University of Campinas
Institute of Computing

*Universidade Estadual de Campinas
Instituto de Computação*

Bruno Cardoso Lopes

“Design and evaluation of compact ISAs”

Supervisor: Prof. Dr. Rodolfo Jardim de Azevedo
Orientador(a):

“Estudo e avaliação de conjuntos de instruções compactos”

PhD Thesis presented to the Post Graduate Program of the Institute of Computing of the University of Campinas to obtain a Doutor degree in Computer Science.

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Doutor em Ciência da Computação.

THIS VOLUME CORRESPONDS TO THE FINAL VERSION OF THE THESIS DEFENDED BY BRUNO CARDOSO LOPES, UNDER THE SUPERVISION OF PROF. DR. RODOLFO JARDIM DE AZEVEDO.

ESTE EXEMPLAR CORRESPONDE À VERSÃO FINAL DA TESE DEFENDIDA POR BRUNO CARDOSO LOPES, SOB ORIENTAÇÃO DE PROF. DR. RODOLFO JARDIM DE AZEVEDO.

Supervisor's signature / *Assinatura do Orientador(a)*

CAMPINAS
2014

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Maria Fabiana Bezerra Muller - CRB 8/6162

L881d Lopes, Bruno Cardoso, 1985-
Design and evaluation of compact ISAs / Bruno Cardoso Lopes. – Campinas,
SP : [s.n.], 2014.

Orientador: Rodolfo Jardim de Azevedo.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de
Computação.

1. Arquitetura de computador. 2. Sistemas embutidos de computador. 3.
Compressão de dados (Computação). 4. Compiladores (Computadores). I.
Azevedo, Rodolfo Jardim de, 1974-. II. Universidade Estadual de Campinas.
Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Estudo e avaliação de conjuntos de instruções compactos

Palavras-chave em inglês:

Computer architecture

Embedded computer systems

Data compression (Computer science)

Compiling (Electronic computers)

Área de concentração: Ciência da Computação

Titulação: Doutor em Ciência da Computação

Banca examinadora:

Rodolfo Jardim de Azevedo [Orientador]

Jorge Luiz e Silva

Roberto A Hexel

Guido Costa Souza de Araujo


Mario Lucio Cortes

Data de defesa: 14-03-2014

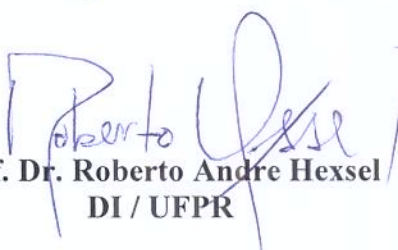
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

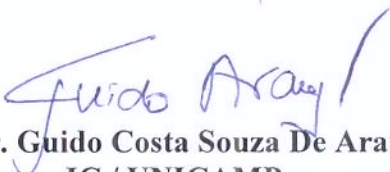
Defesa de Tese de Doutorado em Ciência da Computação, apresentada pelo
Doutorando **Bruno Cardoso Lopes**, aprovada em 14 de março de 2014,
pela Banca examinadora composta pelos professores doutores:



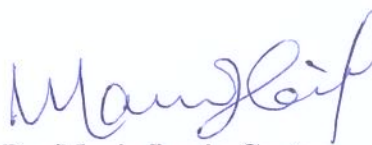
Prof. Dr. Jorge Luiz e Silva
ICMC / USP




Prof. Dr. Roberto Andre Hexsel
DI / UFPR



Prof. Dr. Guido Costa Souza De Araujo
IC / UNICAMP



Prof. Dr. Mario Lucio Cortes
IC / UNICAMP



Prof. Dr. Rodolfo Jardim de Azevedo
IC / UNICAMP

Design and evaluation of compact ISAs

Bruno Cardoso Lopes¹

March 14, 2014

Examiner Board / *Banca Examinadora*:

- Prof. Dr. Rodolfo Jardim de Azevedo (Supervisor / *Orientador*)
- Prof. Dr. Jorge Luis e Silva
ICMC - USP São Carlos
- Prof. Dr. Roberto A. Hexsel
DI - UFPR
- Prof. Dr. Guido Araújo
IC - UNICAMP
- Prof. Dr. Mário Lúcio Côrtes
IC - UNICAMP

¹Financial support: CNPQ 131894/2008-0, CAPES 2009 and FAPESP 2009/02270-0

Abstract

Modern embedded devices are composed of heterogeneous SoC systems ranging from low to high-end processor chips. Although RISC has been the traditional processor for these devices, the situation changed recently; manufacturers are building embedded systems using both RISC - ARM and MIPS - and CISC processors (x86). New functionalities in embedded software require more memory space, an expensive and rare resource in SoCs. Hence, executable code size is critical since performance is directly affected by instruction cache misses. CISC processors used to have a higher code density than RISC since variable length encoding benefits most used instructions, yielding smaller programs. However, with the addition of new extensions and longer instructions, CISC density in recent applications became similar to RISC. In this thesis, we investigate compressibility of RISC and CISC processors, namely SPARC and x86. We propose a 16-bit extension to the SPARC processor, the SPARC16. Additionally, we provide the first methodology for generating 16-bit ISAs and evaluate compression among different 16-bit extensions. SPARC16 programs can achieve better compression ratios than other ISAs, attaining results as low as 67%. SPARC16 also reduces cache miss rates up to 9%, requiring smaller caches than SPARC processors to achieve the same performance; a cache size reduction that can reach a factor of 16. Furthermore, we study how new extensions are constantly introducing new functionalities to x86, leading to the ISA bloat at the cost a complex microprocessor front-end design, area and energy consumption - the x86 ISA reached over 1300 different instructions in 2013. Moreover, analyzed x86 code from 5 Windows versions and 7 Linux distributions in the range from 1995 to 2012 shows that up to 57 instructions get unused with time. To solve this problem, we propose a mechanism to recycle instruction opcodes through legacy instruction emulation without breaking backward software compatibility. We present a case study of the AVX x86 SIMD instructions with shorter instruction encodings from other unused instructions to yield up to 14% code size reduction and 53% instruction cache miss reduction in SPEC CPU2006 floating-point programs. Finally, our results show that up to 40% of the x86 instructions can be removed with less than 5% of overhead through our technique without breaking any legacy code.

Resumo

Sistemas embarcados modernos são compostos de SoC heterogêneos, variando entre processadores de baixo e alto custo. Apesar de processadores RISC serem o padrão para estes dispositivos, a situação mudou recentemente: fabricantes estão construindo sistemas embarcados utilizando processadores RISC - ARM e MIPS - e CISC (x86). A adição de novas funcionalidades em software embarcados requer maior utilização da memória, um recurso caro e escasso em SoCs. Assim, o tamanho de código executável é crítico, porque afeta diretamente o número de *misses* na cache de instruções. Processadores CISC costumavam possuir maior densidade de código do que processadores RISC, uma vez que a codificação de instruções com tamanho variável beneficia as instruções mais usadas, os programas são menores. No entanto, com a adição de novas extensões e instruções mais longas, a densidade do CISC em aplicativos recentes tornou-se similar ao RISC. Nesta tese de doutorado, investigamos a compressibilidade de processadores RISC e CISC; SPARC e x86. Nós propomos uma extensão de 16-bits para o processador SPARC, o SPARC16. Apresentamos também, a primeira metodologia para gerar ISAs de 16-bits e avaliamos a compressão atingida em comparação com outras extensões de 16-bits. Programas do SPARC16 podem atingir taxas de compressão melhores do que outros ISAs, atingindo taxas de até 67%. O SPARC16 também reduz taxas de *cache miss* em até 9%, podendo usar caches menores do que processadores SPARC mas atingindo o mesmo desempenho; a redução pode chegar à um fator de 16. Estudamos também como novas extensões constantemente introduzem novas funcionalidades para o x86, levando ao inchaço do ISA - com o total de 1300 instruções em 2013. Além disso, 57 instruções se tornam inutilizadas entre 1995 e 2012. Resolvemos este problema propondo um mecanismo de reciclagem de *opcodes* utilizando emulação de instruções legadas, sem quebrar compatibilidade com softwares antigos. Incluímos um estudo de caso onde instruções x86 da extensão AVX são recodificadas usando codificações menores, oriundas de instruções inutilizadas, atingindo até 14% de redução no tamanho de código e 53% de diminuição do número de cache misses. Os resultados finais mostram que usando nossa técnica, até 40% das instruções do x86 podem ser removidas com menos de 5% de perda de desempenho.

Agradecimentos

Dedico esta tese aos meus avós, meus grandes ídolos: João, Carmem, José Galino e Dulce. Vocês me ensinaram a importância da humildade, da fé e da conexão entre as pessoas que amamos. Esta conquista é um pedaço de cada um de vocês.

אין עוד מלבדו

- ✠ Não existem palavras suficientes para expressar a gratidão e admiração que tenho pelos meus pais Salso e Lúdia e à minha irmã Lúvia. Meus espelhos. Vos agradeço por tudo, especialmente por acreditarem em mim e me apoiarem por todo esse caminho. Amo vocês. À Aninha, que participou de perto e carinhosamente sempre me incentivou a seguir em frente. À toda minha família, tios e primos, por todo carinho e suporte.
- ⌌ Ao meu irmão Thiago, que me introduziu à computação na infância, me mostrando os primeiros *hacks*. Sem sua influência eu nunca teria me tornado um cientista. Sem tua companhia, meu amigo, jamais haveria um primeiro passo em direção aos degraus do Altíssimo.
- ⌌ Ao Leonardo Ecco e ao Rafael Auler pela amizade e por toda dedicação ao trabalho em equipe. Esta tese é fruto do *nosso* trabalho.
- ⌌ Aos meus amigos de laboratório: Ecco, Piga, Max, Gabs, Janjão, Auler, Baldas, Klein, George, Nicácio, Raoni e Luiz. Aprendi e me diverti muito com vocês.
- ⌌ Aos meus amigos da Mansão Wayne: Thiago, João, Gabriel, Ferrugem e Dilly. Depois de tantos anos de conversas sinceras noite adentro, companheirismo e amizade, mais um ciclo se fecha. Vocês me ensinaram muito, serei sempre grato, a lembrança destes tempos será eterna.
- ⌌ Ao meu orientador Rodolfo, pela qualidade da orientação e por me ensinar a fazer ciência. Às agências de fomento à pesquisa Capes 03/2009-06/2009 e Fapesp 2009/02270-0.
- ⌌ Às *Nuven*s *Invisíveis*; não existe inspiração sem música.

Contents

Abstract	ix
Resumo	xi
Agradecimentos	xiii
1 Introduction	1
1.1 RISC: SPARC16	2
1.2 CISC: The x86 recycling mechanism	3
1.3 Contributions	4
1.4 Organization	5
1.5 Considerations	5
2 Basic Concepts and Related Work	7
2.1 Definitions	7
2.2 The SPARC Architecture	8
2.2.1 Instructions	8
2.2.2 Registers	10
2.2.3 ABI	11
2.3 The x86 ISA	13
2.3.1 Instructions	13
2.3.2 Execution Modes	14
2.3.3 Registers	14
2.3.4 ISA Extensions	15
2.3.5 Implementation	15
2.4 Code Compression	16
2.4.1 Software	16
2.4.2 Hardware	17
2.5 ISA re-encoding	19
2.5.1 Thumb and Thumb2	20

2.5.2	MIPS16 and MicroMIPS	21
2.6	Compiler Optimizations	22
2.7	Compression techniques summary	24
3	Motivation	29
3.1	Code Size Evaluations	29
3.2	ISA aging problem	31
4	A methodology to create 16-bit extensions	35
4.1	Methodology	35
4.2	Static Analysis	36
4.2.1	ISA Usage	36
4.2.2	Immediate and Register Encoding	37
4.3	Dynamic analysis	41
4.4	Integer Linear Programming Model	41
4.5	Considerations	45
5	SPARC16	47
5.1	Instructions	47
5.1.1	Calls and Branches	48
5.1.2	Load and Store	48
5.1.3	Mode exchange	49
5.1.4	The EXTEND mechanism	50
5.1.5	SETHI instruction	51
5.1.6	Alignment restrictions	51
5.2	Registers	51
5.3	Application Binary Interface	52
5.4	Hardware	53
5.5	Emulator	54
5.6	Toolchain	55
5.6.1	Compiler Frontend and Backend	55
5.6.2	Linker	55
5.6.3	C Library	56
5.6.4	The compilation and execution flow	56
5.7	Compiler Optimizations	57
5.7.1	Delay slots	57
5.7.2	Instruction size reducer	58
5.7.3	Assembler relaxation	58
5.7.4	Mixed Stack Access	59

5.8	Evaluation	61
5.8.1	Compression Ratios	61
5.8.2	Instruction Cache Behavior	63
5.8.3	Performance Estimation	64
5.9	Considerations	67
6	The X86 Recycling Mechanism	69
6.1	Radical Approaches	69
6.1.1	(A) Reduce all Operation Codes to 2 bytes	70
6.1.2	(B) Reduce all Operation Codes to 1 or 2 bytes	70
6.1.3	(C) Convert to a RISC-like ISA encoding	71
6.1.4	Evaluation	71
6.1.5	Re-encoding and Backward Compatibility	72
6.2	Recycling mechanism	73
6.2.1	Instruction lifetime cycle	73
6.2.2	Operation Code Revisions and Orthogonality	74
6.2.3	Backward compatibility	76
6.2.4	Revision Vector and Trap Mask	76
6.2.5	Trap Mechanism	78
6.3	Hardware	80
6.3.1	Page Table extension	80
6.3.2	Processor Front-end	81
6.3.3	Verification	82
6.3.4	ISA Domain Specialization	82
6.4	Software	82
6.4.1	Assembler and Linker	82
6.4.2	Operating System Loader	82
6.4.3	Emulation Routines	83
6.5	Security implications	83
6.6	Limitations	84
6.7	Evaluation	84
6.7.1	Methodology	84
6.7.2	Static Analysis	86
6.7.3	Dynamic Analysis	88
6.7.4	Performance Impact	89
6.7.5	Case study: AVX Re-encoding	93
6.8	Considerations	95

7	Conclusion	97
7.1	Contributions	98
7.2	Publications	99
7.3	Future Work	100
	Bibliography	101
A	Static Analysis	111
A.1	Instruction Usage By Group	111
B	SPARC16 ISA	115
B.1	List of SPARC16 Instructions	115
B.1.1	ADDCCri, ADDCCri_ext, ADDCCrr, ADDri, ADDri_ext, ADDrr .	115
B.1.2	ADDFP, ADDFP_ext	116
B.1.3	ADDSP, ADDSP_ext	116
B.1.4	ADDXri, ADDXri_ext, ADDXrr	116
B.1.5	ANDri, ANDri_ext, ANDrr	117
B.1.6	ANDNrr	117
B.1.7	BCC, BCC_ext	118
B.1.8	BA, BA_ext	118
B.1.9	BE, BE_ext	118
B.1.10	BNE, BNE_ext	119
B.1.11	CALL, CALL_ext	119
B.1.12	CALLR	119
B.1.13	CALLRX	119
B.1.14	CALLX, CALLX_ext	120
B.1.15	CMPri, CMPri_ext, CMPrr	120
B.1.16	JMPR	120
B.1.17	JMPRX	121
B.1.18	LDri, LDri_ext, LDrr	121
B.1.19	LDFP, LDFP_ext	121
B.1.20	LDSBri, LDSBri_ext, LDSBrr	122
B.1.21	LDSHri, LSHri_ext, LSHrr	122
B.1.22	LDSP, LDSP_ext	122
B.1.23	LDUBri, LDUBri_ext, LDUBrr	123
B.1.24	LDUHri, LDUHri_ext, LDUHrr	123
B.1.25	MOV, MOV_ext	124
B.1.26	MOV8to32, MOVrr	124
B.1.27	MOV32to8	124

B.1.28	NEGrr	124
B.1.29	NOP	125
B.1.30	ORri, ORri_ext, ORrr	125
B.1.31	ORNri, ORNri_ext, ORNrr	125
B.1.32	RDY	126
B.1.33	RESTORErr, RESTORErr_ext	126
B.1.34	RET	126
B.1.35	RETL	126
B.1.36	SAVEri, SAVEri_ext	126
B.1.37	SDIVri, SDIVri_ext, SDIVrr	127
B.1.38	SETHi	127
B.1.39	SLLri, SLLrr	127
B.1.40	SMULri, SMULri_ext, SMULrr	128
B.1.41	SRAri, SRArr	128
B.1.42	SRLri, SRLrr	129
B.1.43	STri, STri_ext, STrr	129
B.1.44	STBri, STBri_ext, STBrr	129
B.1.45	STFP, STFP_ext	130
B.1.46	STHri, STHri_ext, STHrr	130
B.1.47	STSP, STSP_ext	131
B.1.48	SUBrr	131
B.1.49	SUBXri, SUBXri_ext, SUBXrr	131
B.1.50	tRESTORE	132
B.1.51	UDIVri, UDIVri_ext, UDIVrr	132
B.1.52	UMULri, UMULri_ext, UMULrr	132
B.1.53	WRY	133
B.1.54	XNORri, XNORri_ext, XNORrr	133
B.1.55	XORri, XORri_ext, XORrr	133

List of Tables

2.1	SPARC <i>opcode</i> field encoding	9
2.2	SPARC load and store data types and alignment	9
2.3	SPARC call and branch target computation	9
2.4	SPARC register window addressing	10
2.5	X86 instruction encoding example	13
2.6	Software decompression summary	24
2.7	CDM techniques summary	25
2.8	PDC techniques summary	26
2.9	ISA re-encoding techniques summary	26
2.10	Code compaction techniques summary	27
3.1	X86 default floating-point emission type among distinct compilers	33
4.1	SPARC instruction usage - top 4 groups in mediabench	37
4.2	SPARC register usage statistics	40
4.3	SPARC most executed instructions in mediabench and MiBench	42
4.4	Description of ILP fields and inputs	43
5.1	SPARC16 formats	47
5.2	SPARC16 EXTEND formats	50
5.3	SPARC16 SETHI instruction	51
5.4	SPARC16 registers	52
5.5	SPARC16 compression ratios in mediabench, MiBench and SPEC CINT2006	61
5.6	SPARC16 speedup values against SPARC in MiBench's <i>rijndael</i> program for 128 and 4k byte cache sizes	67
6.1	List of x86 operating systems and software	85
6.2	Number of unused x86 operation codes by size. There were no unused 1 and 2 bytes operation codes.	86
A.1	Instruction usage by groups – mediabench	112

A.2	Instruction usage by groups – MiBench	113
A.3	Instruction usage by groups – Linux Kernel	114

List of Figures

1.1	Program sizes and Code Compression techniques	2
2.1	SPARC Format 1	8
2.2	SPARC Format 2	8
2.3	SPARC Format 3	8
2.4	SPARC overlapping windows (extracted from [87])	11
2.5	SPARC stack frame delimited by fp and sp	12
2.6	SPARC procedure calling example	12
2.7	Intel IA-32e and IA-32 instruction formats	13
2.8	X86 32-bit general purpose registers	14
2.9	Intel IA-32e extended page table entry format	16
2.10	Thumb and ARM ADD instructions	20
2.11	Mapping between MIPS and MIPS16 instruction fields	21
2.12	MicroMIPS LW32 and LW16 instruction formats	22
2.13	MIPS32 and MicroMIPS instruction alignment	22
3.1	SPEC CINT2006 program sizes for several architectures	30
3.2	SPEC CINT2006 code size evaluations across sequential x86 releases	31
3.3	Number of x86 instructions and operation code size increase over the years	32
3.4	Percentage of the code size growth of SPEC floating point programs when compiled with SSE and AVX relative to IA-x87.	33
4.1	SPARC instructions usage by groups	37
4.2	Immediate size usage for SPARC format 3 instructions	38
4.3	Immediate size usage for SPARC format 1 and 2 instructions - calls and branches	39
4.4	SPARC register usage coverage	41
5.1	Assembly mode exchange from SPARC16 and SPARC	49
5.2	Assembly mode exchange from SPARC and SPARC16	49
5.3	SPARC branch with exchange instruction: sparcv8bx	50

5.4	SPARC jump and link with exchange: <code>jmp1x</code>	50
5.5	Unaligned SPARC16 <code>SETHI</code> instruction	51
5.6	SPARC16 decompression diagram	53
5.7	SPARC16 program emulation steps in QEMU	54
5.8	SPARC16 compilation and execution flow	56
5.9	SPARC16 delay slot fulfillment	58
5.10	SPARC16 instruction size reducer	59
5.11	SPARC16 mixed <code>fp</code> and <code>sp</code> optimization	60
5.12	Effect of optimizations in code size reduction of SPARC16 programs	62
5.13	Compression ratio comparison between SPARC16, Thumb2 and Mips16 . .	63
5.14	MiBench - SPARC and SPARC16 cache miss ratios	65
5.15	mediabench - SPARC and SPARC16 cache miss ratios	66
5.16	SPEC CINT2006 - SPARC and SPARC16 cache miss ratios	66
5.17	SPARC16 speedup against SPARC in MiBench's <i>rijndael</i> program for dis- tinct cache sizes	67
5.18	SPARC and SPARC16 cache sizes without performance degradation	67
6.1	Windows 7 and Ubuntu 12 re-encoded using instruction frequency from (i) a collection of operating systems and (ii) SPEC2006 programs	70
6.2	SPEC CPU2006 re-encoded by 3 radical changes to the x86 encoding . . .	71
6.3	Windows 7 and Ubuntu 12 re-encoded by approaches A and B using most used instructions from sources (i) and (ii)	72
6.4	X86 operation code (OC) reuse and lifetime	74
6.5	Operation Code Revision	75
6.6	The x86 operation code (OC) orthogonality and OCR's	75
6.7	CPU generated traps via <i>Trap Masks</i>	78
6.8	General trap mechanism using the Active Revision Vector	79
6.9	Intel IA-32e page table entry extended with the code version information. .	81
6.10	Linux and Windows OC count and outdated OCs over time	87
6.11	Linux and Windows dynamic outdated instruction count over time	89
6.12	A dynamic instruction frequency histogram sorted with respect to Win- dows 95 instructions usage, compared to Windows 7 instructions usage, in logarithm scale. Spikes show differences in usage pattern.	89
6.13	Performance overhead relative to the percentage of the ISA that is emulated with emulation penalty of 200 instructions.	90
6.14	Maximum ISA emulation ratio when tolerating up to 5% of overhead and using different emulation penalties.	91
6.15	Emulation experiment using Linux kernel modules and patched executables	92

6.16	Percentage of extension instructions executed in Windows and Linux dynamic traces.	93
6.17	Total code size of SPEC 2006 floating point programs in different scenarios, relative to the original AVX floating point arithmetic version.	94
6.18	Instruction cache misses of SPEC 2006 floating point programs with shorter AVX encodings, relative to the original AVX encoding.	95

Chapter 1

Introduction

Modern embedded devices are heterogeneous and can be divided into three main categories: low, mid and high-end devices [86]. *Low-end* micro-controllers cost less than a dollar while *mid-end* chips can cost anywhere between one and ten dollars. *High-end* microprocessors are the most expensive, ranging from ten to a hundred dollars. Each category targets a specific market [83]; from smart-card chips to complex full-fledged mobile phones. Nowadays, these chips tend to be all-in-one computer chips, namely *System-on-Chip* [10] (SoC), containing a processing core, memory and integrated peripherals.

A production cost of a SoC is proportional to the chip area ($cost \approx area^4$) [46] and is mitigated by decreasing the number of integrated peripherals or by reducing internal memory capacity - main memory and processor caches. Applications, by adding new features, demand extra memory or need at least the same amount as before - performance degradation happens otherwise. For instance, high-end devices increasingly requires responsive user interfaces, high level languages support and multitasking while low-end devices are already highly constrained and cost sensitive to additional hardware resources. Therefore, to reduce production cost, manufacturers need alternative ways to produce chips that support new modern software with the minimum increase in memory capacity as possible.

Code compression is a solution capable of providing a reduced alternative representation to program instructions, mitigating overall program memory consumption - thus allowing production of chips with the same area but able of running more complex software. Compression techniques reduce instructions during or after compilation and decompress them during execution, as illustrated in Figure 1.1. The instruction compression effectiveness depends on the ISA encoding entropy: variable length versus fixed length encodings. The former is dense and compact - shorter encodings are attributed to most common instructions - while the latter has lower density and more opportunity [29] for compression. This is the first main distinction between CISC [91] and RISC [80] ISAs;

the former are composed by variable length instructions and the second by fixed length ones.

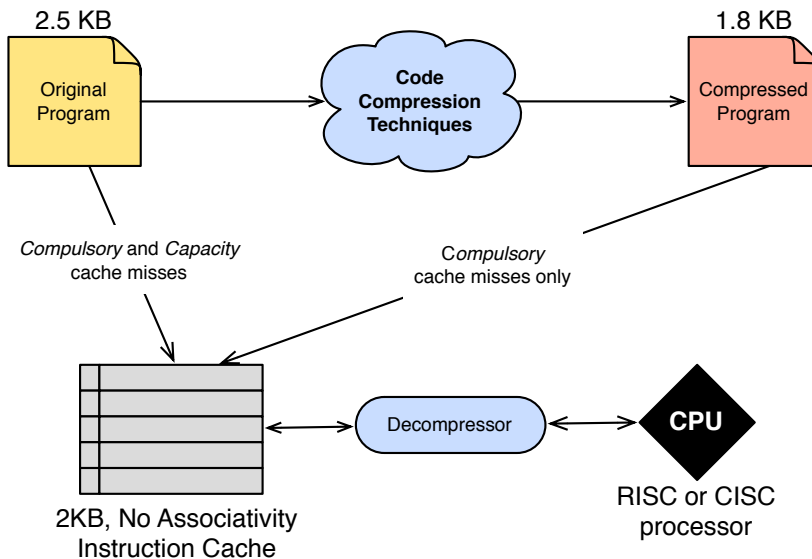


Figure 1.1: Program sizes and Code Compression techniques

RISC processors are traditionally the default core used in embedded SoC devices and have strong potential for running code compressed applications; *ARM* [7] and *MIPS* [76] are examples of RISC architectures used in low, mid and high-end devices. On the other hand, the Intel *x86* [52] ISA also targets embedded CISC architectures with the high-end *Atom* [45] and low-end *Quark* [51] processors. However, the x86 ISA lost its CISC variable length encoding properties: short encoding instructions from old ISA releases are now legacy and superseded by long encoding instructions introduced in late ISA extensions. Recent work [20] shows that CISC and RISC ISAs in modern embedded systems are equivalent in performance and energy efficiency.

In this thesis we study code compression opportunities for both RISC and CISC processors. We explore the memory consumption space issue and propose novel techniques to enhance compressibility and performance.

1.1 RISC: SPARC16

Code compression can be implemented both in software and hardware [17]. In a solely - compression and decompression - software approach there is always an associated performance degradation cost during decompression. The hardware approach focuses on decompression speed, usually at the cost of code size reduction. There are two different approaches for hardware decompression, one that compresses instructions or blocks of the

program in an *ad-hoc* manner and other that tries to create an alternative encoding for the instructions in a smaller size; reducing a 32-bit RISC ISA to a well-defined 16-bit format. This thesis focuses on the latter approach to find a new encoding to the SPARC ISA [87].

16-bit ISA extensions exist for RISC architectures such as MIPS and ARM, with more details given in Chapter 2. However, existing implementations are commercial products [24, 6, 57, 77, 31, 40, 50] and no published research presents how such extensions were designed. We show how 16-bit extensions can be designed based on a specific case study; the creation of a 16-bit extension to the SPARC ISA, the SPARC16 [33]. The method includes an extensive static and dynamic analysis of several benchmarks and binaries to find intrinsic ISA compression inefficiencies and opportunities. Also, an *Integer Linear Programming (ILP)* model optimally assists in the creation of formats and field encodings for the new 16-bit extension. A general method can be abstracted from our case study and applied to other architectures.

Our compiler toolchain is based on LLVM [64] and Binutils [42] Open Source projects and provides SPARC16 support for the whole toolchain: Frontend, Backend, Assembler and Linker. The linker is capable of linking object files from SPARC and SPARC16 together, allowing usage of existing SPARC libraries. Target specific optimizations enhance final SPARC16 code compressibility by exploring intrinsic 16-bit instructions properties.

The SPARC16 instructions are translated to their 32-bit counterparts during execution time by an additional hardware decompressor placed between the processor and the instruction cache - this mechanism was implemented into Leon3, a SPARC compliant processor. A SPARC16 emulator was developed to enable the testing of SPARC16 programs while providing information to allow instruction cache evaluation.

Finally, we evaluate compression and performance results for SPARC16 and compare the results with other related work. We also present compression ratio results achieved for SPARC16 programs, which can be lower than most popular 16-bit extensions such as MicroMIPS and Thumb2.

1.2 CISC: The x86 recycling mechanism

Old CISC ISAs like the IA-32 [52] suffer from the ISA aging problem: it is necessary to add new instructions in the already occupied opcode space, and eventually the ISA runs out of space for new opcodes. CISCs handle this problem by increasing instruction length, while RISC ISAs may need a new processor mode – one in which the opcode space is interpreted differently. For instance, modern x86 uses both approaches: it introduces additional instruction prefixes to expand the opcode space and also uses another mode, the IA-32e [52], to interpret instructions differently in the context of 64-bit programs.

The instructions decoder occupies a significant fraction of the chip size. Borin et al. [23] state that an Intel low power design estimation concluded that up to 20% of the die area is used by the microcode ROM alone, a component responsible for decoding the more complex instructions.

New processors with old ISAs bear an inherent disadvantage that goes beyond the hardware overhead: the variable-length encoding benefits instructions no longer used and penalizes recent additions, since the shortest encodings are already taken by the instructions introduced first. The consequences of being biased to the past is that not only modern code is larger, but it also reduces the number of instructions that fit into the instruction cache, directly affecting performance.

We show how to overcome the harmful effects of expanding aged ISAs. We seek a novel approach to maintain an ISA that is as efficient as a newly designed one in terms of code compaction and decoder size, while still being backward compatible with older software developed for it. To reach this goal, we propose the use of a recycling mechanism for the IA-32 ISA that allows selected short opcodes to change their previous functionality to serve a new, more useful, instruction.

This strategy also allows the elimination of deprecated instructions in order to simplify the hardware and help reduce x86 microcode space used by such instructions, allowing more efficient x86 hardware implementations of processors such as Atom [45] and Quark [51]; Intel low-power processors for the embedded market.

1.3 Contributions

The main contributions of this thesis are:

- A method for designing 16-bit ISA extensions from 32-bit ISAs.
- A 16-bit extension for the SPARC architecture.
- A report of the IA-32 evolution over time that shows how opcode usage of programs released from 1995 to 2012 evolved, and how several instructions stopped being used, from the static and dynamic point of views.
- A recycling mechanism that enables the IA-32 ISA opcode space to be better exploited.
- An analysis of how much code compaction can be achieved in SPEC FP 2006 [47] programs if we re-encode the newer IA-32 AVX extension with smaller opcodes, currently assigned to instructions that are not used anymore, and the ensuing beneficial effects on the instruction cache misses.

- A performance impact estimation of the proposed recycling mechanism when executing legacy code that uses re-encoded opcodes.

1.4 Organization

This PhD thesis is organized as follows:

- Chapter 2 describes related work on the area and basic concepts regarding the SPARC and X86 architectures.
- Chapter 3 focuses on the opportunities for code compression among several architectures, providing the motivation for the thesis. It explains why SPARC and X86 are the selected RISC and CISC ISAs for analysis, showing code size evaluation for several architectures while characterizing the ISA aging issue.
- Chapter 4 describes our method for generating 16-bit extensions, as a SPARC16 case study. We evaluate the SPARC ISA, according to several code analyses for distinct programs, both static and dynamic. We describe an Integer Linear Programming (ILP) model, which assists in the definition of SPARC16 instruction formats.
- Chapter 5 presents the resulting SPARC16 formats and encodings. We give details about the hardware implementation, emulators, compiler toolchain support and compiler optimizations. Finally, we show the archived SPARC16 compression ratios, compare them with other 16-bit extensions and provide performance results.
- Chapter 6 further explores compression opportunities in the x86 CISC case study. We propose three radical approaches to solve the ISA aging problem in x86 and then present our recycling mechanism, proposing an implementation and experimental results.
- We draw the thesis conclusion in Chapter 7.

1.5 Considerations

The work done in this thesis has contributions by two other students: Leonardo Ecco, a former Master student from UNICAMP and current PhD student at *Technische Universität Braunschweig*, and Rafael Auler, a PhD student from UNICAMP. Their contributions are described in details in Section 4.5, 5.9 and 6.8.

Chapter 2

Basic Concepts and Related Work

This Chapter presents the background in code compression and a brief description of SPARC and x86 architectures. Section 2.1 defines all basic terms used in the text. Sections 2.2 and 2.3 describe x86 [52] and SPARC [87] architectural properties; such as registers, instructions and instruction encoding.

We introduce code compression related work in Section 2.4, where we present software and hardware compression mechanisms. In Section 2.5 we detail ISA re-encoding methods and 16-bit extensions; Thumb/Thumb2 and MIPS16/MicroMIPS. Related work on code compaction compiler optimizations is presented in Section 2.6. In Section 2.7 we summarize of all techniques presented in this Chapter.

2.1 Definitions

Compression Ratio. We use the term *Compression Ratio* to represent code size reduction. Equation 2.1 shows how Compression Ratio should be calculated. Notice that, lower means better. Example: a Compression Ratio of 56% means that the program is reduced to 56% of its original size. All related overhead to implement compression must be considered within the ratio computation, a rule not strictly followed by all related work in the area.

$$Compression\ Ratio = \frac{Compressed\ Size + Overhead}{Original\ Size} \quad (2.1)$$

Operation Code. We define the term *Operation Code* or *OC* as the minimum necessary bits in an instruction which distinguishes it from others. In RISC architectures, the operation code is usually the same as the *opcode*. However, in CISC ISAs such as x86, extra bits besides the opcode are necessary to maintain this distinction. This set of bits will be henceforth referred as the operation code.

2.2 The SPARC Architecture

The Scalable Processor Architecture (SPARC) version 8 (v8) [87] is a published IEEE 1754-1994 [28] standard: it defines 72 instructions encoded in 32-bit formats and general purpose, floating point, control and status registers. In this thesis we always refer to SPARCv8 whenever the term SPARC is used.

The main SPARC characteristics include three available 32-bit instruction formats in which opcode and registers field sizes are fixed across them. Memory access is given by two registers or a register and immediate - arbitrary memory load and store is only done by specific instructions. 8 fixed global registers are available while a variable 24 register window (see Section 2.2.2) is always accessible from a large register bank¹.

2.2.1 Instructions

Instructions in SPARC are encoded in 3 main 32-bit formats. The formats 1, 2 and 3 are presented in Figure 2.1, 2.2 and 2.3 respectively.

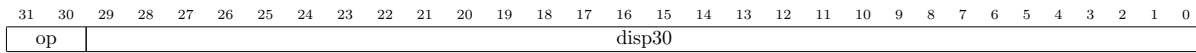


Figure 2.1: SPARC Format 1

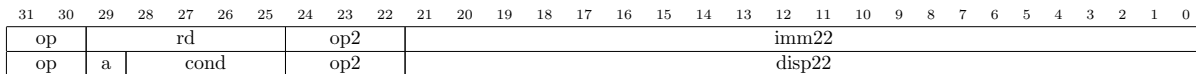


Figure 2.2: SPARC Format 2

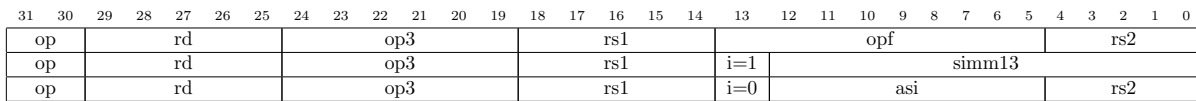


Figure 2.3: SPARC Format 3

Table 2.1 shows how the encoding of the *op* field determines the instruction format; format 1 (*op* = 1) holds the *CALL* instruction whereas format 2 (*op* = 0) contains all branch and the *SETHI* instructions. Format 3 is divided into two sub-formats: the first (*op* = 3) encodes all memory access instructions (load and stores) and the second (*op* = 2) has the logical, arithmetic, shift and all remaining instructions in the ISA.

¹The number of windows within the register bank is implementation dependent - ranging from 2 to 32

Format	<i>op</i>	Instructions
1	1	CALL
2	0	Bicc, FBfcc, CBccc, SETHI
3	3	Memory access instructions
3	2	Logical, arithmetic, shift and other instructions

Table 2.1: SPARC *opcode* field encoding

Load and store. The load and store instructions in SPARC have a different opcode for each distinct memory slot type size. For instance, `ldub`, `lduh`, `ld` and `ldd` instructions load a *byte*, *half*, *word* and *doubleword* data types from memory. The memory addresses accessed in load or store instructions must be aligned to the data type, otherwise a *memory not aligned* trap is issued by the processor. Table 2.2 shows the alignment restriction for each data type. For example, the `lduh` access 2 byte sized objects in the memory, and their address are always 2 bytes aligned in memory.

Instructions	Data Type	Memory Address Alignment
<code>ldub</code> , <code>ldsb</code> , <code>stb</code>	1 byte	—
<code>lduh</code> , <code>ldsh</code> , <code>sth</code>	2 byte	2 byte
<code>ld</code> , <code>st</code>	4 byte	4 byte
<code>ldd</code> , <code>std</code>	8 byte	8 byte

Table 2.2: SPARC load and store data types and alignment

Call and branches. Call and branch instructions compute targets relative to `pc` as shown in Table 2.3. The immediates in *disp30* and *disp22* are encoded without the two least significant bits - a 4 byte alignment enforcement. For example, consider a `call` instruction to address `0x40` and `pc = 0x00`. The address `0x40` is encoded in *disp30* with the value `0x10`. Hence, the final address is $00_{16} + (4 \times 10_{16}) = 40_{16}$.

During the target calculation, the immediates in *disp30* and *disp22* are multiplied by 4, because the displacement is implied to be 4 byte aligned - this is done by the linker during relocation. Therefore, the final target address is formed by the required alignment.

Instructions	Format	Target Address
<code>call</code>	Format 1 (Figure 2.1)	$pc + (4 \times disp30)$
<code>bne</code> , <code>be</code> , <code>ble</code> ,...	Format 2 (Figure 2.2)	$pc + (4 \times disp22)$

Table 2.3: SPARC call and branch target computation

2.2.2 Registers

The main register types available in SPARC and of interest in this thesis are:

Control and Status Registers Register Y is used by multiply and divide instructions.

The higher 32-bit result of 32-bit multiplications are stored in the Y register, while in a division, the 32-bit most significant bit from the divisor must be stored in Y prior to computation. The program counters **pc** and **npc** holds the address of the current instruction in execution and the address of the next instruction to be executed, respectively. The **psr** register maintain a group of processor status information.

General Purpose There are two types of general purpose registers: integer and floating point². Integer general purpose registers names start with *r*. The integer unit can contain 40 to 520 general purpose registers partitioned into 8 global registers (*globals*, **g0** to **g7**) and a variable number of 24 register sets. Each 24 register set is divided into 8 input registers (*ins*, **i0** to **i7**), 8 output registers (*outs*, **i0** to **i7**) and 8 local registers (*locals*, **l0** to **l7**).

Register address within the window	Register address <i>r</i>
in [0] — in [7]	r [24] — r [31]
local [0] — local [7]	r [16] — r [23]
out [0] — out [7]	r [8] — r [15]
global [0] — global [7]	r [0] — r [7]

Table 2.4: SPARC register window addressing

At a given execution point, a program can access 8 global registers and a 24 *r* registers. The *r* registers are mapped to the current *register window* according to Table 2.4. The set of *outs* of the current window is an alias to the set of *ins* of the next window, as illustrated in Figure 2.4. Note that locals are never shared among windows and globals are not part of the window mechanism.

The current window pointer *CWP* is part of the **psr** and tracks the current window using 5 bits (allowing up to 32 windows). *CWP* is incremented through the use of the **restore** and **rett** instruction and decremented by the **save** instructions or by exceptions. Window overflow and underflow generate exceptions which are handled by the operating system.

The register **g0** always returns the zero value when read. Register **o7** is written with the return address of the current function when the **call** instruction is executed. When an exception occurs, **l1** and **l2** registers receive the values of the **pc** and **npc** registers.

²SPARC Floating point instructions are not considered in this thesis

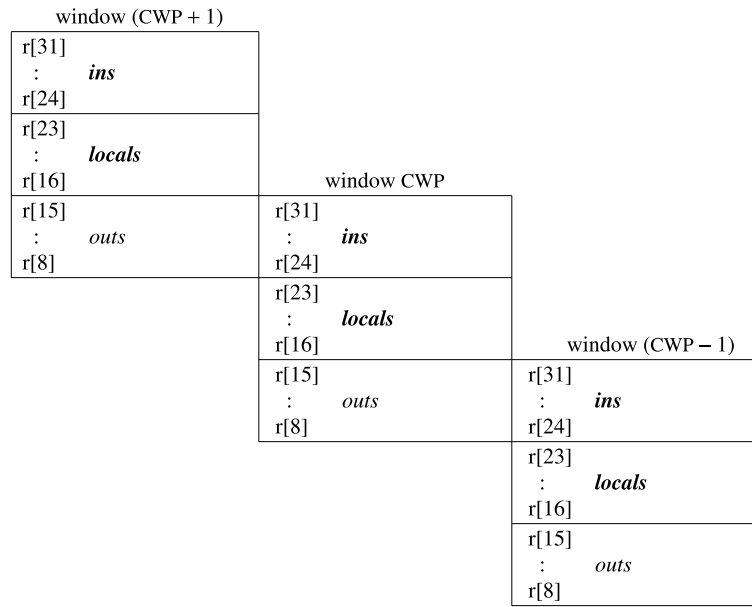


Figure 2.4: SPARC overlapping windows (extracted from [87])

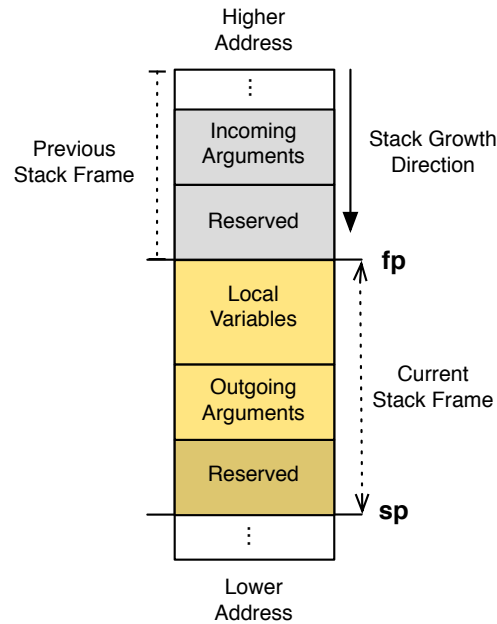
2.2.3 ABI

The Application Binary Interface (*ABI*) defines a set of conventions to be followed by compilers and operating systems to make sure applications and libraries from different sources might interact with each other. Examples of conventions described in the ABI include function calling sequence, object file headers, relocations and dynamic linking. The SPARC ABI [53] defines set of rules, which must be followed by every SPARC compliant compiler, operating system and libraries.

The registers `o0` to `o5` are used to pass arguments to callee functions whereas `i0` to `i5` receive the arguments in the function. If there are more than 6 arguments, the remaining are passed on the stack.

The stack pointer (`sp`) and frame pointer (`fp`) registers determine a function stack frame, as illustrated in Figure 2.5 - these register names are aliases to `o6` and `i6` respectively. Function extra *incoming arguments* are fetched from the stack by positive offsets against `fp`, while negative offsets give access to *local variables* on the stack. The `save` instruction, is used on function entry (prologue) to adjust the stack frame and provide local variables storage. On function exit (epilogue) the `restore` instruction unrolls the stack to its previous frame. Note that, as explained before, these instructions also handle the register window transitions.

Consider the C program in Figure 2.6b. In the equivalent assembly code from Figure 2.6a, the function `sum` is called from `main` with the first argument (the value 10) in `o0` and the second argument (value 20) in `o1`. Arguments in `sum` are read from `i0` and

Figure 2.5: SPARC stack frame delimited by `fp` and `sp`

i1. Within *sum*'s prologue, the `save` instruction allocates a 96 byte stack frame, updates `fp` and `sp` and changes the register window, remapping `o0` to `i0` and `o1` to `i1`.

```

1 sum:
2   save %sp, -96, %sp ; prologue
3   add %i0, %i1, %i0 ; args in
4   mov %i0, %i0
5   ; epilogue
6   restore ; restore %g0,%g0,%g0
7   ret    ; jmp %i7+8, %g0
8   nop    ; delay slot
9   ...
10 main:
11   save %sp, -96, %sp ; prologue
12   mov 10, %o0 ; arg out
13   mov 20, %o1 ; arg out
14   call sum
15   nop    ; delay slot
16   ...

```

(a) Assembly file

```

1
2 int sum(int a, int b) {
3     return a+b;
4 }
5
6 int main() {
7     int a = 10;
8     int b = 20;
9     int res;
10    res = sum(a, b);
11    ...

```

(b) C source code

Figure 2.6: SPARC procedure calling example

The function return value must always be returned in `i0`; that is why the `mov` instruction in *sum* writes the function result in `i0`, prior to the epilogue. The `restore` instruction unroll the stack frame and the `ret` instruction returns to the previous function using the return address in `o7`.

2.3 The x86 ISA

The x86 instruction set family is the collection of all machine instructions derived from the Intel 8086 family of CISC processors. The ISA is backwards compatible for all processor families; recent machines can run old programs and libraries assembled back in 1978.

2.3.1 Instructions

The instructions in the x86 ISA have a variable length format, and the basic encoding to represent a single instruction is usually determined via the *opcode* and *prefix* fields. Some instructions further require the use of the *ModR/M* field to be decoded. The layout is given in Figure 2.7.

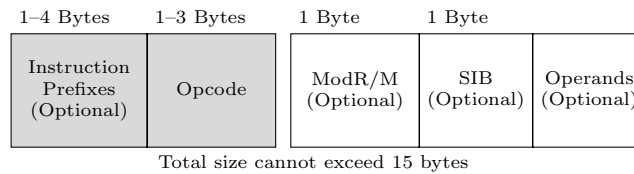


Figure 2.7: Intel IA-32e and IA-32 instruction formats

For example, an instance of a logic `or` instruction between a 16-bit immediate and a 16-bit value held in memory that is indexed by the register `rcx` may be represented by the assembly text form `orw $12804, (%rcx)`. Table 2.5 depicts its equivalent encoding in machine language.

Prefix	Opcode	ModR/M		SIB	Operands	
		Mod	Reg/Opc	R/M		
66h	81h	00b	001b	001b	N/A	04h 32h

Table 2.5: X86 instruction encoding example

The ModR/M byte is part of the opcode encoding in this instruction because its subfield Reg/Opc is used as an opcode extension. Hence the instruction has 5 bytes: 3 bytes are used for opcode and prefix and 2-bytes for the immediate. To easily refer to the necessary bits required to uniquely identify our definition of x86 instruction and avoid confusion with the x86 *opcode* byte, we use the term *operation code*, as defined in Section 2.1.

2.3.2 Execution Modes

Different execution modes are used to support different instruction encoding and also to enable the execution of old instructions and legacy software, which depends on a memory and register layout from obsolete versions of the x86 ISA.

There are four major execution modes in modern x86 processors: 64-bit long, 32-bit protected, 16-bit protected and 16-bit real. The difference between protected and real mode relies on the handling of memory segments. The former uses a descriptor table to index physical pages while the later directly access memory with a fixed sized page addressing.

The 64-bit long mode executes instructions from the x86_64, which defines larger general registers, memory addressing and different encodings for some instructions. In this thesis we focus on 32-bit protected mode and disconsider all other modes for being out the scope of this thesis.

2.3.3 Registers

The x86 ISA has 4 register classes, namely:

General registers There are four 32-bit general purpose registers: **eax**, **ebx**, **ecx** and **edx**. Historically, these registers had pre-assigned roles which became deprecated. 8-bit and 16-bit access to part of those register is possible, as illustrated in Figure 2.8, to maintain backward compatibility with old instructions which operate on those register sizes.

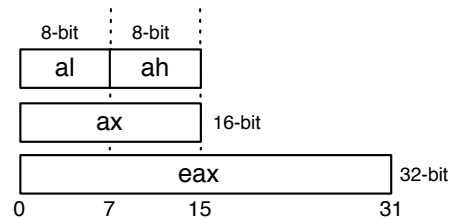


Figure 2.8: X86 32-bit general purpose registers

Segment registers Four registers to support segment addresses for code, data, stack and extra uses. **CS**, **DS**, **ES**, **FS**, **GS**, **SS**. These registers are reserved and not used by the compiler, and are set by using special instructions.

EFLAGS A register which holds processor state and information on instruction side effects. Arithmetic overflow, sign changes and zero results are examples of information stored in **EFLAGS**.

2.3.4 ISA Extensions

The X86 ISA evolved from the initial 8086 revision and incorporated a number of new instructions on each microarchitecture release over time.

The 80387 Also known as x87, the extension introduced floating-point support instructions to the regular x86 ISA. The FP instructions can access eight 80-bit floating point registers, `ST(0)` to `ST(7)`, organized as a stack and special instructions push and pop values to and from it.

Multimedia extensions Intel introduced the *MMX* extension, a *SIMD* ISA for multimedia processing. The extension added 8 64-bit registers (`MM0` to `MM7`), which are aliased to the x87 stack registers. The instructions work on packed data; the 64-bit registers are used as two 32-bit, four 16-bit or eight 8-bit registers, allowing parallel data processing. MMX provides only integer operations. AMD released *3DNow!* in 1998, an enhanced version of MMX, capable of processing 32-bit floating point data for common arithmetic operations (add, subtract and multiply).

The next multimedia extension appeared in the Pentium III series with the *SSE* [52] extension, adding 8 128-bit registers (`XMM0` to `XMM7`) and 70 instructions, which can operate on four 32-bit floating point and integer data packed in a 128-bit register. SSE was extended in further releases from 2001 to 2013 with the introduction of *SSE2*, *SSE3*, *SSSE3*, *SSE4.1*, *SSE4.2*, *AVX* and *AVX2*. Each subsequent SSE version introduced new instructions to operate on more fine-grained packed element sizes (e.g. sixteen 8-bit data packed in a 128-bit register) for floating point and integer data. AVX and AVX2 extend the register sizes to 256-bit and 512-bit while supporting all previous instructions operating on 128-bit registers.

Specific extensions for security, cryptography and virtualization were also incorporated in microarchitecture releases but are out the scope of this thesis.

2.3.5 Implementation

Instructions in modern Intel x86 processors are translated into *microcode* operations by the processor *decoder front-end*. The front-end is fast and decodes CISC x86 instructions into RISC like microcode instructions, which are easier to process, demanding less control logic while allowing for a more regular pipeline. Other benefits from microcode include the possibility to change the micro-architecture while still providing a fixed interface via CISC instruction decoding in the front-end.

In 32-bit mode, x86 supports 4GB virtual address space. The full address space is visible to any process by using a paging system. The page mechanism provides isolation:

Raeder [26] studies compression of Java bytecodes with a 70% compression ratio and 30% performance loss. Every compression approach in this section has performance degradation penalties during execution.

2.4.2 Hardware

Two hardware models can be used for code decompression: *Cache Decompressor Memory (CDM)* and *Processor Decompressor Cache (PDC)*. In CDM, the hardware decompression engine is placed between the cache and memory, whereas in PDC, it stays between processor and cache. Compression methods using PDC yield better performance [70, 16, 19, 78] than CDM [93, 5]. In PDC, because the cache holds compressed code, the number of cache hits increases, diminishing the number of accesses to the main memory.

CDM techniques. Wolfe [94] proposed in 1992 the *Compressed Code RISC processor (CCRP)*, using compile time Huffman [48] compressed cache lines. A *Line Address Table (LAT)* is used to decompress the cache lines during execution by fetching the original cache lines from main memory. The technique achieved 73% compression ratio. Benes [14, 15] further improved on this work by using specific Huffman decoder circuitry, capable of decompressing 32-bits in 25ns.

IBM released in 1998 the PowerPC 405 CodePack [55, 31, 40, 50], using a dictionary method to compress instructions, a different approach from MIPS and ARM. Two dictionaries are used: each one is linked to a distinct 16-bit part of the original instruction and are encoded separately. Each 16-bit part is encoded by a tag and an index; tags occupy 2 or 3 bits while indices from 0 to 16 bits - the smallest possible encodings takes 7 bits and the larger 38 bits. The compression ratio ranges from 60% to 65% and the performance impact is $\pm 10\%$.

Pannain [79] proposed the *Pattern Based Compression (PBC)* operand factorization method. The PBC separates instruction expression trees into two components: *tree patterns* containing opcode fragments and *operand patterns* with registers and immediates. The components are compressed separately, yielding 43% and 48% compression ratios for Huffman and *Variable Length Code* algorithms, respectively.

Centoducatte [25, 5] further introduced the *Tree Based Compression (TBC)*, where expression trees are not decomposed in components, but grouped in classes and compressed entirely. The ratio achieved reaches 60.7%.

Azevedo [9] proposes the *Instruction Based Compression (IBC)* where instructions are grouped in classes and compressed, replacing the original instruction by references to the instruction class and an index to an instruction table. Implementations for MIPS and SPARC reached average compression ratios of 56% and 61.5% respectively. The SPARC

implementation has a 5.8% performance degradation penalty.

PDC techniques. Lefurgy [67, 68] presents a compression method similar to Liao's; instead of abstracting common code into procedures, *codewords* are used to represent such sequences and to index into dictionary entries, allowing intermixing with uncompressed code. Average compression ratios of 61%, 66% e 75% are obtained for PowerPC, ARM and x86 processors.

Lekatsas [70, 71] compresses SPARC instructions by decomposing them into four groups: (1) instructions using an immediate, (2) branch instructions, (3) fast access and (4) not compressed. Each group is compressed by a different technique reaching an average 65% compression ratio, 28% power consumption reduction and 25% performance gain. Another work [69] by Lekatsas uses a decompressor architecture capable of performing an instruction decompression in one cycle without processor cycle-time degradation. The method uses dictionaries, targeting the Xtensa-1040 processor. A 25% performance gain together with 65% compression ratio is achieved in this approach.

Benini [16] uses a 32-bit 256 word dictionary to compress DLX processor instructions. The dictionary is built upon dynamic instruction usage and a 32 byte cache line compression unit. A 72% compression ratio and a 30% power consumption is achieved.

Billo [19] and Wanderley [78] propose a compression mechanism for the SPARC architecture and combine a PDC with dictionaries built upon static and dynamic usage of instructions, achieving compression ratios from 72% to 88%, up to 45% performance gain and 35% on power consumption reduction.

The use of a dictionary [27, 85, 44, 8, 62] to compress code tends to reduce energy consumption and improves the performance and compression ratio. Using bitmask and prefix based Huffman encoding [44], the compression ratio improves by 9–20%. Kumar [62] analyzes compression for variable-length ISA RISC processors with two approaches: using a bit-vector and using a reserved instruction to identify code words. Results demonstrate an speed-up of up to 15%, code size reduction (up to 30%) and bus-switching activity (up to 20%).

Bonny [21] uses hardware support to optimize the number of lookup tables generated by statistic compression schemes and encoding [22] unused bits in the instruction opcode space to improve the compression ratio. Compression ratios as low as 56% are achieved for ARM, MIPS and PowerPC processors.

Qin [82] implemented a fast parallel hardware decompressor without compression efficiency penalties. A code placement technique enables parallel decompression by splitting a single bit-stream fetched from memory into multiple bit-streams; each is input to a different decoder. This approach improved decode bandwidth up to four times with minor impact (less than 1%) on compression efficiency.

Corliss [30] proposes a post-fetch decompressor using a *Dynamic Instruction Stream Editing (DISE)* hardware, a programmable decoder that is used to add functionality to an application by injecting custom code snippets into the fetched instruction stream. Thus, program-specific dictionaries can be used, improving compressibility. The experimental results show reduction in code size (up to 35%), improvements in performance (up to 20%) and energy (up to 10%).

Xu et al. [96] propose a memory compression architecture for the first Thumb ISA, achieving a Thumb code size reduction from 15% to 20%. A high-speed hardware decompressor improves timing performance of the architecture, resulting in performance overheads limited within 5% of the original application.

Krishnaswamy [61] creates the ThumbAX extension, prefixing 16-bit instructions with an extra halfword to achieve equivalent functionality of a regular 32-bit ARM instructions, without the need of mode exchange. The mechanism is similar to Mips16 EXTEND and a very similar mechanism is present in Thumb2.

Santos [75] proposed the *Pattern Based Instruction Word (PBIW)* on the RVEX processor; a mechanism that maps the assembly file generated by a compiler into an encoding scheme of the target processor. Compression ratio ranges from 61% to 116% (program increase) and improvements up to 59% in cache hits. In addition, the circuit shrinks the total area by 15% on average while reducing dynamic power consumption.

2.5 ISA re-encoding

This section presents additional related work on generic and 16-bit ISA re-encoding.

Support for an existing ISA modification with the recycling goal has, to the best of our knowledge, never been proposed before, although ISA randomization to avoid security attacks has been studied [41, 12]. Flexible hardware-software interfaces, as opposed to a fixed ISA, were studied by Barat et al. [11] in the context of reconfigurable instruction set processors. Such systems share the same purpose of designing a functionality rich ISA in a way such that application code can be densely encoded, reducing the total number of executed instructions and increasing efficiency. However, these solutions focus on reconfigurable hardware [54, 39] that is capable of adapting to software needs rather than the redesign of an existing ISA.

The Altera Nios [3] processor architecture designed for use into Altera FPGAs is an example of a flexible processor architecture that is capable of removing some instructions from hardware to be emulated in software, sharing some similarities with our work. The designer may want to allocate FPGA hardware resources to other hardware IPs that increases the overall chip functionality rather than instantiating a resource-expensive processor. Therefore, a design space tradeoff can be explored by reducing performance of

specific workloads, removing some instructions from hardware and reducing the processor footprint.

The *DLX* architecture, created by Hennessy and Patterson [81], was the first 32 bits architecture to have a 16 bit extension. The extension, called D16, had instructions with 2 registers operands (the original had 3) and also smaller immediate field. This configuration allowed a 62% compression ratio and 5% performance loss [24].

2.5.1 Thumb and Thumb2

ARM introduced *Thumb* [6] as the first 16-bit extension in ARM7. Later on, *Thumb2* was released and superseded the initial Thumb, introducing additional features. Thumb2 enabled ARM processors are capable of running code in both 32 and 16 bits modes and allow subroutines of both types to share the same address space. Mode exchange is achieved during runtime through *BX* and *BLX* instructions; branch and call instructions that flip the current mode bit in a special processor register.

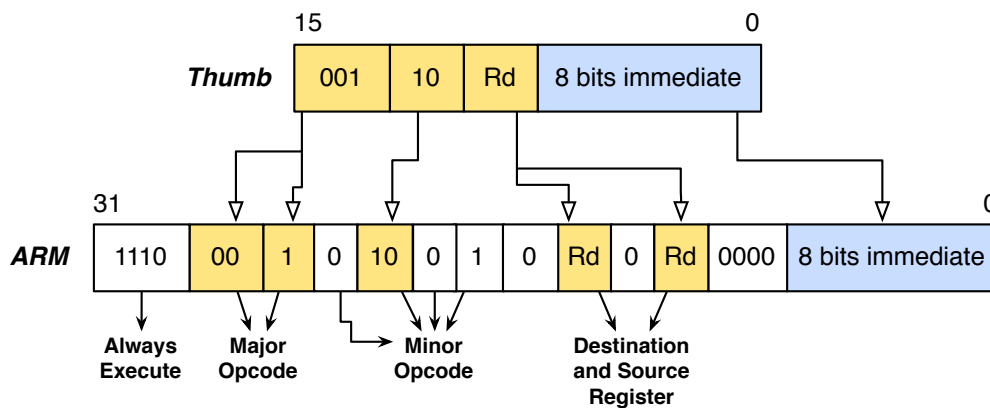


Figure 2.10: Thumb and ARM ADD instructions

The encoding of the Thumb extension has several constraints in contrast to the original ISA in order to hold more opcodes - the reduction in register field size is a significant one (Figure 2.10). A group of only 8 registers including the stack pointer and link registers are visible (the *Lo* group), but the remaining registers can also be accessed implicitly (the *Hi* group) or through other special instructions - *mov* and *cmp* instructions can move and compare registers between both groups.

Results presented by ARM for Thumb, show a compression ratio ranging from 55% to 70%, with an overall performance gain of 30% for 16 bit buses and 10% loss for 32 bit buses.

2.5.2 MIPS16 and MicroMIPS

The MIPS16 [57] ISA is the first 16-bit extension released for MIPS. It contains capabilities to exchange between modes using the Jump and Link with Exchange *JALX* instruction, share address space, has only 8 visible registers and reduced immediate size - from 16 bits to 5 (Figure 2.11). The *MOV32R* instruction move data between visible and hidden registers, while special instructions access the hidden group implicitly.

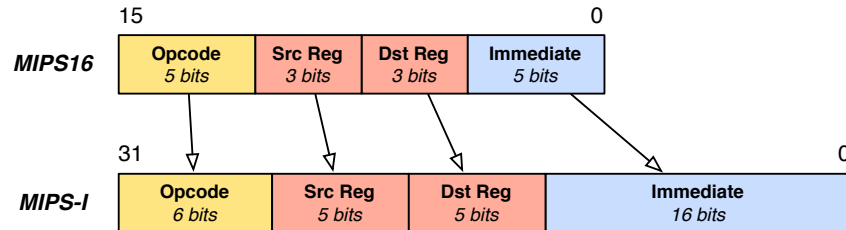


Figure 2.11: Mapping between MIPS and MIPS16 instruction fields

MIPS16 introduces the *EXTEND* instruction: an opcode and an immediate field that is used to extend the immediate of the following instruction. New features introduced by MIPS16 include:

SP relative addressing The stack pointer register *SP* is implicitly accessed by some instructions, allowing the use of more bits in the immediate field (8 bits).

PC relative addressing The program counter register *PC* is implicitly accessed by load instructions. The approach allows constants to be loaded using only one instruction when placed in the text segment of nearby functions.

Load and Store offset shift The immediates representing offsets in load and store instructions are shifted right according to its alignment to discard unused bits. Thus, shorter immediates are used and expanded to the original value during execution.

The instructions using relative SP and PC addressing can be combined to the offset shift in load and stores, allowing an effective 1K address range without the need for the *EXTEND* instruction. MIPS16 achieves a 60% compression ratio according to MIPS technologies [57].

The MicroMIPS [77], released in 2009, is a new 16-bit ISA for MIPS and is not compatible with MIPS16. It introduces 54 instructions and is supported as a distinct mode from MIPS32 and MIPS64. While in MicroMIPS mode, each instruction has a 16-bit and 32-bit version (Figure 2.12) and no *EXTEND* instruction is required, a design very similar to Thumb2. Additionally, 16-bit and 32-bit instruction can be mixed together without any alignment restrictions, as shown in Figure 2.13.

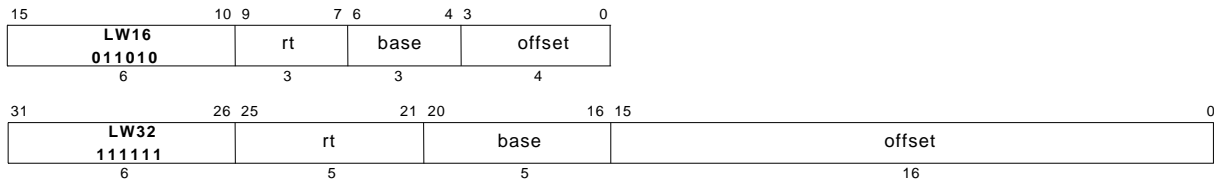


Figure 2.12: MicroMIPS LW32 and LW16 instruction formats

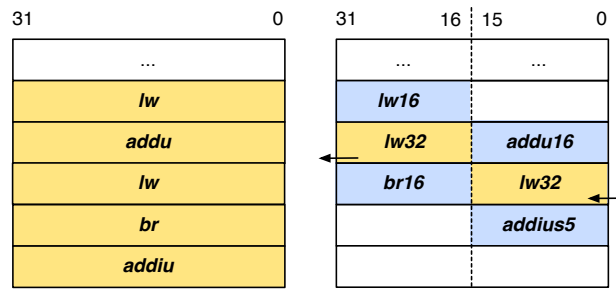


Figure 2.13: MIPS32 and MicroMIPS instruction alignment

Results provided by MIPS technologies [77] report an average of 65% compression rate for the CSiBE [18] benchmark and a 2% speed-up against MIPS32 for the Dhrystone benchmark.

2.6 Compiler Optimizations

The use of 16-bit extensions does not always guarantee a smaller code size. Code for loading large constants, for instance, is smaller and faster when using 32-bit instructions, since there is more space to encode immediates and extra registers. Hence, compiler code generators can apply optimizations to take advantage of mixed 16 and 32 bits ISAs extensions like MicroMIPS and Thumb2.

Krishnaswamy [60] proposes a coarse grained approach where only one bit-width type of instructions, 16 or 32-bit, could be emitted in each function. Profile information is used to select hot functions and several heuristics to decide the bit-width of each function in a program are evaluated: results range from 69% e 77% compression ratio against pure 16-bit code.

Edler [58] uses a compiler for *ARCOMPAT*, a mixed 16 and 32-bit ISA, to propose a special instruction selection heuristic for mixed ISAs. The heuristic avoids the emission of 16-bit instructions when it is not profitable. First, only 16-bit instructions are selected during instruction selection. Second, a special register allocator annotates all places where the usage of 16-bit instructions are responsible for generating spills. Finally, the instruction selection is invoked again, but using annotated data, avoids using

16-bit instructions in the potential spill sites. The feedback-guided instruction selection improves performance by 17% with 85% compression ratio.

The 32-bit ISA *UniCore32* and its 16-bit ISA counterpart *UniCore16* are analyzed by Xianhua [95]. UniCore16 has regular instructions, such as *add* and *mov*, capable of mode exchange. The compiler emits 32-bit instructions by default and heuristics may replace them by 16-bit instructions during link-time when profitable. The compression ratio is 73% against pure UniCore16 without any performance drawback.

Sutter [89, 90] applies link time optimizations to ARM binaries: reconstruction from final executables is done through the creation of an augmented whole-program control-flow graph (*AWPCF*), where all text and data sections are considered and optimizations such as whole-program analyses, duplicate code and data elimination techniques are applied. The mechanism reduces code size from 16% to 18%, provide 8% to 17% performance gain and power consumption reduced by 8% to 16%.

Kumar [63] evaluates link time optimizations using a peephole technique with finite state machines instead of string matching, achieving compression ratios from 98.9 to 99.1%.

2.7 Compression techniques summary

In this section we summarize the code compression related work into several tables; in Table 2.6 we present the software decompression approaches, whereas in Table 2.7 and 2.8 the hardware based CDM and PDC methods. Table 2.9 contains a comparison between 16-bit extensions and Table 2.10 summarize the compiler related optimization techniques.

Software Decompression				
Author	Architecture	Benchmarks	Compression Ratio	Performance Ratio
Fraser e Proebsting [37]	SPARC	lcc e burg	50%	2000%
Kirovski [56]	SPARC	Mediabench	60%	110%
Liao [73, 72]	TMS320C25	Aipint2, bench, compress, dfx2hsh, gnucrypt, gzip, hill, jpeg, rsaref, rx, set	88%	—

Table 2.6: Software decompression summary

CDM				
Author	Architecture	Benchmarks	Compression Ratio	Performance Ratio
Wolfe & Chanin [94]	MIPS	lex, pswarp, yacc, who, eightq, matrix25A, loop01, xlist, espresso e spim	73%	—
IBM [31, 55]	PowerPC	—	60%	—
Pannain [79]	MIPS	SPECint95	43%	—
Centoducatte [25, 5]	MIPS	SPECint95	60.7%	—
Azevedo [9]	MIPS	SPECint95	53,6%	—
Azevedo [9]	SPARC	SPECint95	61,4%	105,89%

Table 2.7: CDM techniques summary

PDC					
Author	Architecture	Benchmarks	Compression Ratio	Performance Ratio	Power Reduction
Lekatsas [70, 71]	SPARCV8	compress, diesel, i3d, key, mpeg, smo, trick	65%	75%	-28%
Lekatsas [69]	Xtensa 1040	compress, diesel, i3d, key, mpeg, smo, trick	65%	75%	—
Benini [16]	DLX	Ptolemy	72%	—	-30%
Lefurgy [67, 68]	PowerPC	SPECint95	61%	—	—
Lefurgy [67, 68]	ARM	SPECint95	66%	—	—
Lefurgy [67, 68]	i386	SPECint95	75%	—	—
Billo [19]	SPARCV8	susan, search, dijkstra, adpcm, pegwit, libmad	75%	78%	-45%

Table 2.8: PDC techniques summary

ISA Re-encoding				
Author	Architecture	Benchmarks	Compression Ratio	Performance Ratio
D16 [24]	DLX	—	52%	105%
Thumb [6]	ARM	Eqntott, Xlisp, Espresso, Dhrystone	55%~70%	110%~120%
Thumb2 [7]	ARM	—	5% < Thumb	102%
MIPS16 [57]	MIPS	—	60%	—
MicroMIPS [77]	MIPS	CSiBE, Dhrystone	65%	102%

Table 2.9: ISA re-encoding techniques summary

Compiler Code Compaction Optimization				
Author	Architecture	Benchmarks	Compression Ratio	Performance Ratio
Krishnaswamy [60]	ARM	mediabench	69.5%~77.2%	—
Edler [58]	ARCOMPAT	EEMBC 1.1	85%	83%
UniCore16 [95]	UniCore32	mediabench, Mesa	73%	100%
Sutter [89, 90]	ARM	mediabench, MiBench	82%~84%	92%
Kumar [63]	x86, ARM	mediabench	98.9~99.1%	—

Table 2.10: Code compaction techniques summary

Chapter 3

Motivation

Our work is primarily concerned with the study of code compression opportunities in ISAs. Our main motivations are:

- There are several 16-bit commercial ISA extensions available. However, there are no academic work or publications regarding the process of creating such extensions. We explore and expose the process of creating such 16-bit extensions with the SPARC16 case study.
- Old CISC ISAs like the x86 suffer from the ISA aging problem: as the interface matures, it is necessary to add new instructions to an already occupied opcode space, and eventually the ISA runs out of space for new opcodes. We investigate how to overcome the harmful effects of expansion characteristic of aged ISAs. We seek a novel approach to maintain an ISA that is as efficient as a newly designed one in terms of code compaction while still being backward compatible with legacy software.

In Section 3.1 we show a code size evaluation for several CISC and RISC architectures, exposing the reasons why and how we selected SPARC and x86 as the targets of this research. Additionally, in Section 3.2 we further explore how x86 became a concern, since its ISA evolved to a situation in which its code density is not as high as it used to be, opening room for code compression opportunities.

3.1 Code Size Evaluations

In order to apply and construct a method for generating a 16-bit ISA extension, we need an architecture for evaluation; an intensive analysis of an original ISA is necessary. Therefore we need to select an architecture with attractive compression opportunities.

We evaluated code size behavior of 15 ISAs using the SPEC CINT2006 benchmark [47]. The GCC [88] 4.2 cross compiler toolchain is used¹ for each architecture with the following arguments: `-Os` to compile for size (at the expense of speed) and `-mcpu` to select architecture specific information (e.g. the *core2* x86 sub-variant).

Figure 3.1 shows the code size evaluation for the 15 mentioned architectures. Results are presented by total size, arithmetic and geometric means for each architecture. The values are normalized against the smallest one within each category - arithmetic, geometric and total.

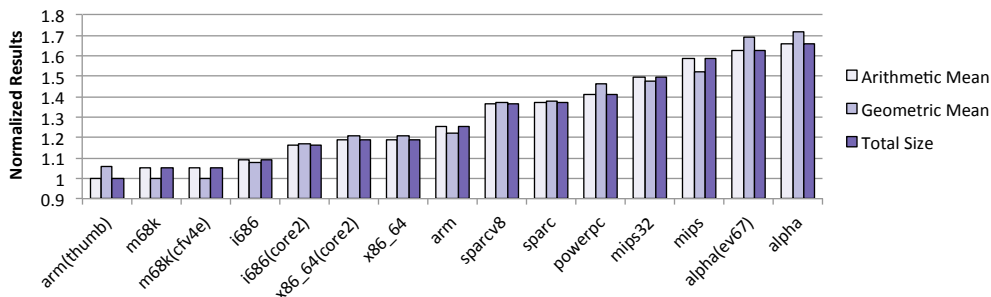


Figure 3.1: SPEC CINT2006 program sizes for several architectures

The architectures with smaller relative code sizes are Thumb, Motorola *M68k* and x86. Thumb, the 16-bit ARM extension, has the higher code density, with smaller code size than CISC x86 and M68k processors. The scenario gives the first hint about CISC compressibility: there should be more room for shrinking x86 and M68k binaries. Additionally, all the other RISC architecture in Figure 3.1 are 32-bit wide and have bigger programs than any CISC. Alpha represents the lower code density and is the RISC ISA with more compression potential. However, it was discontinued, available toolchains and libraries are considered obsolete and it was never used in embedded systems. As mentioned in Section 2.5, MIPS, ARM and PowerPC processors already have reduced bit-width ISA extensions in their architectures; Thumb and Thumb2 for ARM, MIPS16 and MicroMIPS for MIPS and CodePack for PowerPC. SPARC is left as the only 32-bit architecture without a reduced bit-width ISA extension and has the following characteristics:

- Low code density, 40% bigger than higher density architectures - x86 and Thumb.
- The SPARC ISA is an IEEE 1754 standard and still widely used – there are 50 registered members in SPARC International nowadays.
- No previous 16-bit compression mechanism is available for SPARC.

¹GCC 3.4 was used for Alpha and M68k

- Used in many academic projects as a testbed for compression algorithms.
- Up to date and stable software support: operating system kernel, libraries and compilers.

Therefore, we selected the SPARC architecture as our target for RISC ISA exploration and 16-bit ISA generation method case study.

3.2 ISA aging problem

A variable length encoding ISA is supposed to use smaller encodings to frequent instructions and hence achieve smaller programs than RISC ISAs. However, Figure 3.1 suggests that the CISC x86 ISA may have more space for compression than ARM's Thumb - x86 binaries are bigger than Thumb's.

The x86 is a 30 year old architecture with more than fifteen ISA extension releases until 2013; at each new release, new instructions and features were introduced. Since x86 encoding is variable, the addition of new instruction demands the use of more bits to encode an instruction. Therefore, as shown in Figure 3.2, the overall program size for SPEC CINT2006 (2% to 11%) increases in some subsequent x86 processor revisions.

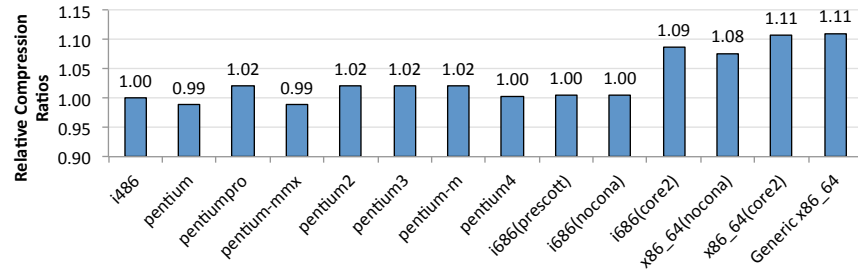


Figure 3.2: SPEC CINT2006 code size evaluations across sequential x86 releases

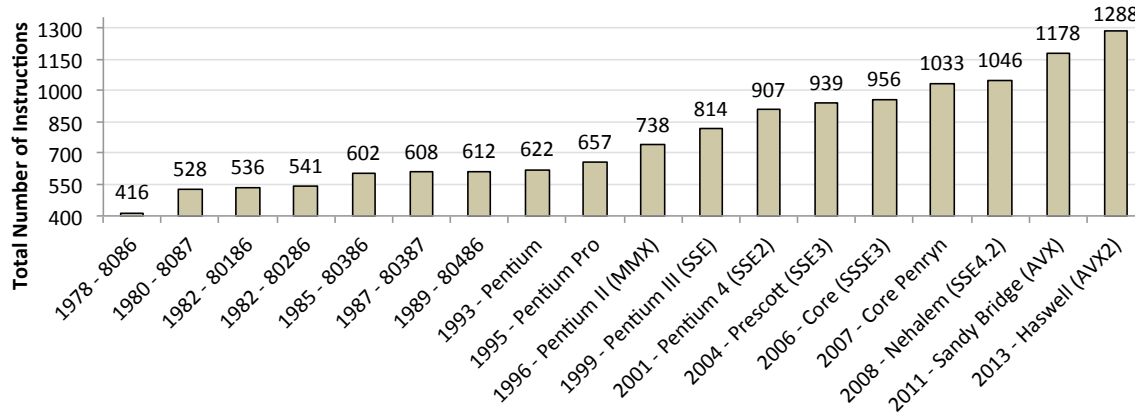
The i686² *core2*³ processor is the most recent in Figure 3.2 and is also the biggest code size against all x86 i686 architectures. In *x86_64* (x86's 64-bit mode) *core2* program sizes are also bigger than *nocona*⁴. The trend is that program size in x86 increases over time. To further confirm this assumption we rely on two factors: (1) the total number of instructions increases with x86 evolution and (2) the average operation code size also expands. Figure 3.3a shows that since the introduction of MMX, a large number of new

²32-bit x86, no x86_64 instruction included

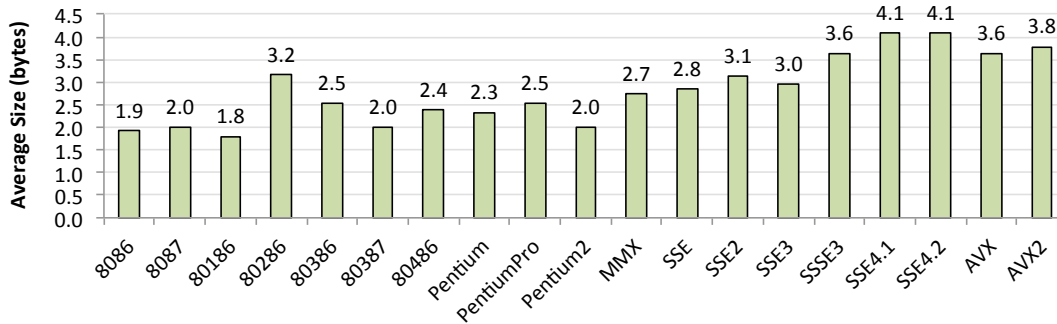
³Intel Core2 CPU with 64-bit extensions, MMX, SSE, SSE2, SSE3 and SSSE3 instruction set support.

⁴Improved version of Intel Pentium4 CPU with 64-bit extensions, MMX, SSE, SSE2 and SSE3 instruction set support.

instructions were added in every new release; there are about 800 new instructions that were introduced from Pentium MMX to the Haswell, released in 2013.



(a) x86 ISA growth over the years



(b) Average x86 operation code size

Figure 3.3: Number of x86 instructions and operation code size increase over the years

Moreover, Figure 3.3b shows that the average operation code size per instruction increases - with SSE4.1 and SSE4.2 achieving the bigger operation codes. Also, the most recent multimedia extensions AVX and AVX2 have 33% and 38% bigger operation codes than instructions introduced with MMX.

As mentioned in Section 2.3.4, such multimedia extensions focused in adding vector instructions that explore data parallelism. The first extensions to address floating-point calculations were 8087 and 80387, but their operand addressing is stack based, a rather old and inconvenient addressing method for modern compilers. Newer vector instructions, starting with the MMX extension, have register operands which allows the compiler to easily control register usage with established register allocation algorithms, and can perform multiple floating-point calculations at the same cycle. For these reasons, vector instructions naturally superseded the old x87 instructions.

Table 3.1 shows that some compilers already use vector extensions to emit floating

point operations by default. *ICC* and *Clang* emit SSE as the default ISA for floating point - it may only emit x87 instructions if the host target has no support for vector operations. On the other hand, *GCC* and *Visual Studio 2012* still emit x87 by default, taking the more conservative approach, since they are very old and stable compilers.

Compiler	Default FP	Comments
Visual Studio 2012	x87	<code>/arch: [IA32 SSE SSE2 AVX]</code> to switch
GCC 4.6.3	x86	<code>-mfpmath=sse</code> or <code>-mfpmath=sse,387</code> to mix both usages
ICC 10.0	SSE	Only use x87 if SSE not supported in the host
Clang 3.1	SSE	Only use x87 if SSE not supported in the host

Table 3.1: X86 default floating-point emission type among distinct compilers

Coming back to Figure 3.3b we notice that vector extension instructions may have 1 to 2 extra bytes for operation code in comparison with the old IA-x87 extension. Since compilers already started a transition to use multimedia instruction for floating point operations, future floating point application are deemed to be bigger; Figure 3.4 shows that for 7 SPEC CPU2006 floating-point programs, using vector extensions yields larger executable binaries than the x87 ones.

In this analysis, we compiled the programs using *GCC* 4.7 with the `-O2` optimization flag and the `-march=corei7-avx` architecture tuning. To generate x87 instructions, we used the `-mfpmath=387` option, while to generate the SSE ones, we used `-mfpmath=sse`, and finally we added the `-mavx` option when testing the AVX encoding⁵ for SSE instructions. Note that no vectorization optimizations were used.

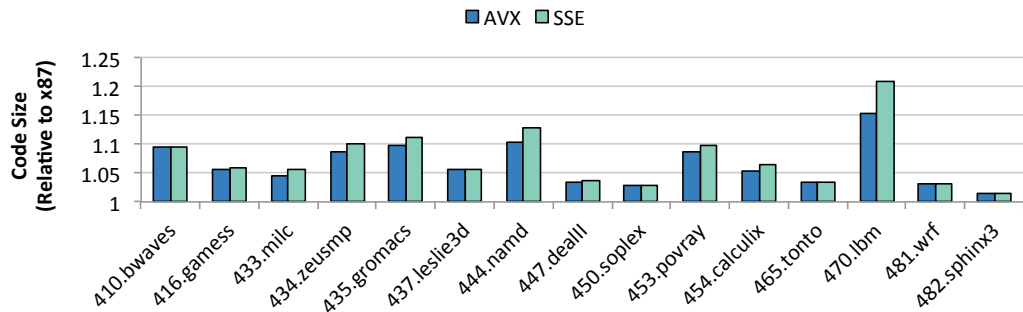


Figure 3.4: Percentage of the code size growth of SPEC floating point programs when compiled with SSE and AVX relative to IA-x87.

The *470.lbm* program is 20% and 15% bigger when using SSE and AVX against x87.

⁵Notice that the AVX introduced not only a new set of 256-bit vector instructions, but also new encodings for all previous SSE instructions. A program compiled using AVX will have SSE instructions encoded using the new operation codes provided by AVX.

Most programs are at least 5% bigger than x87 form, with some exceptions such as *482.sphinx3* and *481.wrf*.

Therefore, in the course of 30 years:

- The number of x86 instructions increased three times its initial size, a direct impact into front-end design.
- The operation code size doubled, a potential problem since this has an impact in instruction cache ratios.
- The x86 extensions are increasing the overall program size - up to 11% in integer and 20% in floating-point applications - opening a new opportunity for compression.

In Chapter 6, we present additional analysis on x86 code size and propose the recycling mechanism, aimed at improving compressibility and operation code space cleanup.

Chapter 4

A methodology to create 16-bit extensions

In this Chapter we propose a method to create 16-bit ISA extensions using a SPARC based case study¹. We use static and dynamic analysis from SPARC compiled programs for several benchmarks and use a *Integer Linear Programming (ILP)* model to generate 16-bit instructions formats and select 16-bit instructions.

Section 4.1 describes our methodology. Section 4.2 details the information obtained with the static analysis: Section 4.2.1 presents statistics about ISA coverage and Section 4.2.2 explores immediate and register usage. Section 4.3 presents dynamic instruction count information and the ILP method is described in Section 4.4.

4.1 Methodology

The first task in the design of a 16-bit extension is to define which instructions from the regular 32-bit ISA must be present in the extension. The main concern is to be representative enough to achieve good compression ratios. To find potential instructions for the 16-bit extension, we organize our analysis as follows:

- Instructions never used by any analyzed program are removed from our selection pool unless a constraint exists. For example, short functions may contain specific instructions which are executed very often but have a low static count. The absence of an instruction not necessarily means it is not used, it may only mean that the search base is incomplete.

¹A generic version of this method can be used to design ASIP ISAs and 16-bit extensions for other RISC architectures

- Organize instructions in groups of similar functionality or operand needs and collect statistics about the presence of each group in the analyzed programs. Most used instructions within the same group tend to share similar formats and are candidates to share a common 16-bit format. Less used instructions are likely to be discarded unless proven worthy by other analysis.
- Usage count of each instruction in the ISA, both statically and dynamically². The most used instructions are directly responsible for final 16-bit compressibility and must be the first candidates for inclusion, thus subject to fewer encoding restrictions for their 16-bit format than the least used ones.

4.2 Static Analysis

Based on the criteria mentioned in Section 4.1, we performed a static analysis in all the programs from the benchmarks *MiBench* [43], *mediabench* [66] and the *Linux Kernel*; all compiled for SPARC by GCC 4.2 with floating point support disabled.

To statically analyze binaries, we used the `objdump`³ disassembler tool wrapped by a *python* application which parses `objdump` assembly output recording several levels of instruction information such as used and defined registers, immediate size and other encoding properties.

4.2.1 ISA Usage

The total usage of SPARC ISA instructions by MiBench, mediabench and Linux Kernel is 42.8%, 42.3% and 62.7%. The later uses more instructions since it performs special tasks, such as privileged instructions and hand crafted assembly code. Roughly, 40% of the ISA instructions were never used, and this is an initial set of instructions to disregard.

We divide instructions by groups of similar functionality and obtain static usage by each group. In Figure 4.1, top used instructions in mediabench, MiBench and Linux Kernel are from the *alu_arith* group; 22.4%, 22.6% and 17.2% respectively. Table 4.1 shows *alu_arith* in mediabench, some of the most used instructions are `add_imm`, `subcc_imm`, `add_reg` and `subcc_reg`. Thus, as mentioned before, most used instructions within each group are the main candidates to have 16-bit counterparts.

Tables A.1, A.2 and A.3 in Appendix A.1 contain the complete instruction usage by groups for mediabench, MiBench and Linux Kernel, refer to that section for details.

²Inclusion of very frequent dynamic instructions may provide a good performance tradeoff

³The `objdump` is part of the Binutils[42] framework.

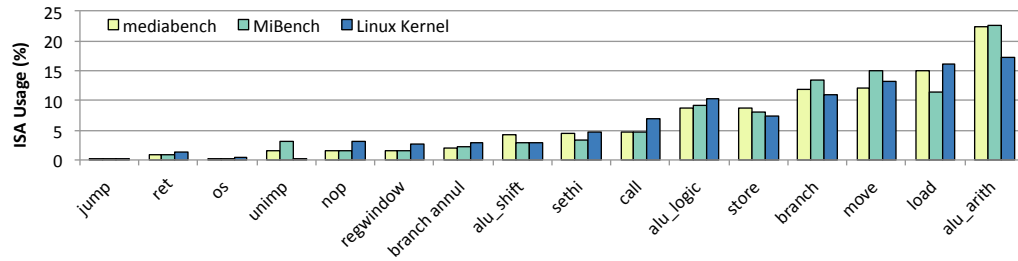


Figure 4.1: SPARC instructions usage by groups

Usage	Group	Instructions			
22.42%	alu_arith	(29.59%)add_imm (28.49%)subcc_imm (15.93%)add_reg (12.28%)subcc_reg (05.68%)sub_reg (02.37%)smul_reg	(01.45%)addcc_imm (01.28%)subx_imm (01.22%)addx_imm (00.33%)umul_reg (00.32%)smul_imm (00.28%)udiv_reg	(00.20%)subx_reg (00.17%)addcc_reg (00.16%)sdiv_reg (00.14%)addx_reg (00.04%)udiv_imm (00.04%)sub_imm	(00.02%)smulcc_reg (00.01%)udivcc_reg (00.00%)sdivcc_reg
15.00%	load	(61.16%)ld_imm (16.69%)ld_reg (05.59%)ldub_reg	(04.46%)lduh_imm (03.54%)ldd_imm (01.85%)ldub_imm	(01.71%)ldsb_reg (01.70%)lduh_reg (01.41%)ldsh_imm	(00.71%)ldsb_imm (00.65%)ldsh_reg (00.54%)ldd_reg
12.14%	move	(71.93%)or_reg	(28.07%)or_imm		
11.76%	branch	(32.66%)ba (22.71%)be (13.58%)bne (07.86%)ble	(06.06%)bl (04.79%)bleu (04.27%)bg (02.91%)bgu	(02.76%)bge (00.80%)bcs (00.67%)bcc (00.58%)bpos	(00.35%)bneg

Table 4.1: SPARC instruction usage - top 4 groups in mediabench

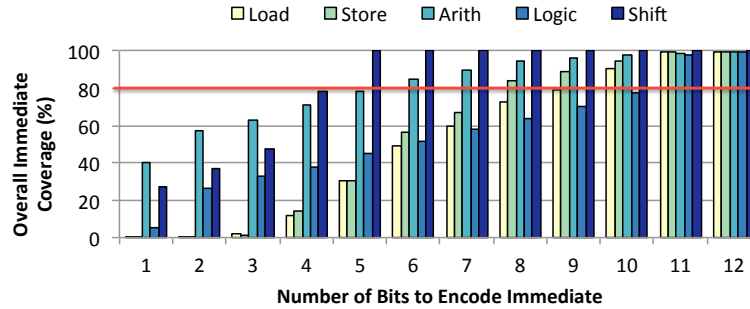
4.2.2 Immediate and Register Encoding

The fields needed when encoding instructions are opcode, immediate and registers. The opcode bit-width is determined by the number of instructions in the ISA and by considering a good compromise between available functionality and opcode space occupation. We must avoid opcode space waste by refusing to include infrequent functionalities. Additionally, we must consider that a large opcode field restricts the size of immediate and register fields.

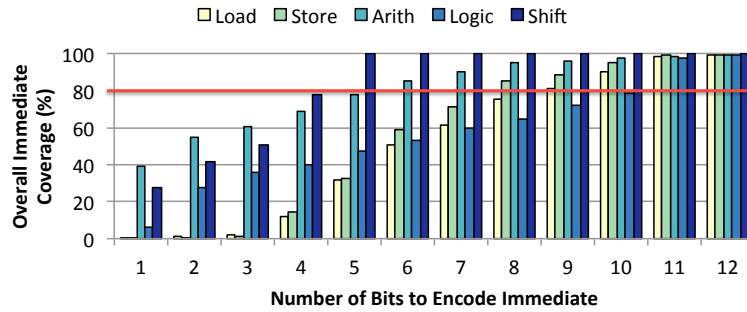
Immediates usually occupy most part of an instruction and are the main target for compression. Thus, as mentioned in Section 2, 16-bit extensions – Thumb, MIPS16 and MicroMIPS – are designed with instructions containing restrained immediate fields. SPARC – as described in Section 2.2.1 and in Figures 2.1, 2.2 and 2.3 – has the following immediate field sizes across three formats: 30, 22 and 13 bits. For instance, the top used

instruction groups (Figure 4.1) *alu_arith*, *load*, *move*, *store* and *alu_logic* are composed of instructions using the SPARC format with 13-bit immediate field.

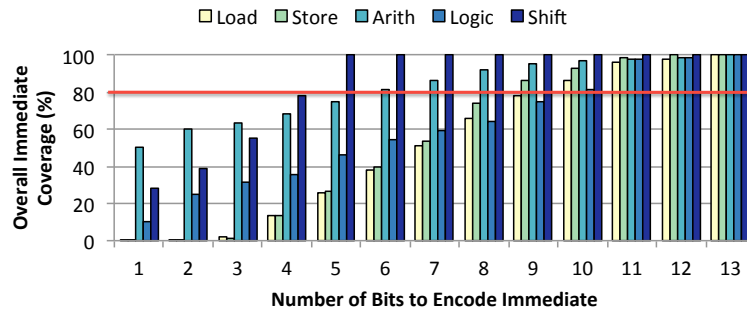
Figure 4.2 shows the number of bits needed to encode the immediate in all arithmetic, logic, shift, load and store instructions for the mediabench, MiBench and Linux Kernel. Notice that, although the immediate field is 13-bits wide, for the three cases in Figure ??, more than 80% of the arithmetic instructions require 6 bits or less. Also, near 80% of the load and store instructions can be represented with 9 or less immediate bits.



(a) mediabench



(b) MiBench

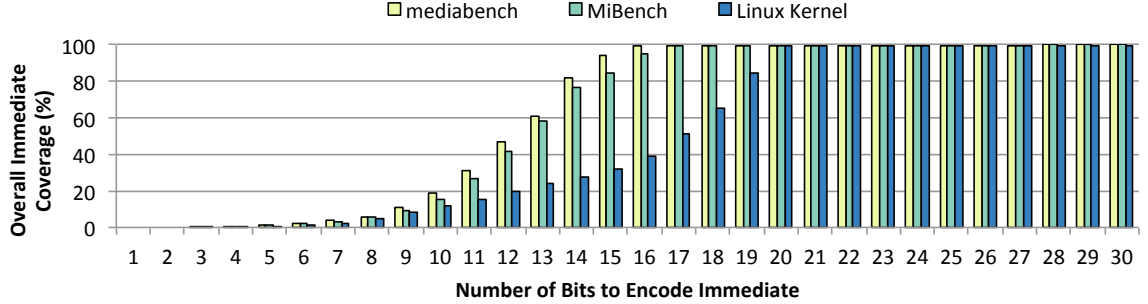


(c) Linux Kernel

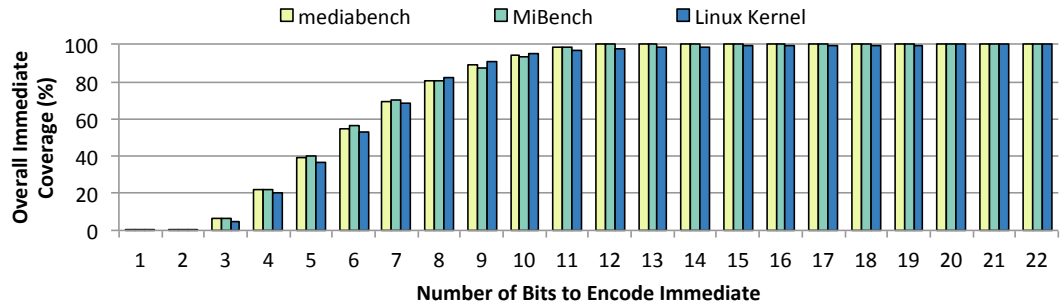
Figure 4.2: Immediate size usage for SPARC format 3 instructions

Figure 4.3 represents the immediate usage for *calls* and *branches*. 80% of call instructions (30-bit immediate field size) can be encoded using 14-bits (Figure 4.3a) in media-

bench and MiBench whereas the Linux kernel needs 19-bits. 80% of branch instructions (Figure 4.3b) need only 8 bits out of 22 in the three benchmarks.



(a) Calls



(b) Branches

Figure 4.3: Immediate size usage for SPARC format 1 and 2 instructions - calls and branches

Register fields are smaller than immediate ones but also important, since they are an intrinsic part of an instruction. Section 2.2.1 contains detailed information on register encoding format and Section 2.2.2 describes all SPARC registers and their associated functions.

Immediate field sizes may vary among instructions, but register field sizes are fixed, allowing all instructions to have the same set of register visibility. In SPARC, `fp` and `sp` are always visible and accessible by all instructions accessing registers. Furthermore, those registers are only referenced during load and store to local variables (by `ld_imm` and `st_imm` instructions), function frame allocation (`save` and `restore`) and stack frame addresses copies (`add_imm`). Hence, both registers are reserved and never used by the compiler register allocator.

Table 4.2a shows that `fp` is the fifth most used register; 30.7% and 26.53% of all `ld_imm` and `st_imm` instructions, as shown in Table 4.2b; suggesting that we may have 16-bit specific load, store and save instructions to implicitly access such registers, hiding

them from regular instructions, which leaves extra room for encoding regular registers, decreasing register allocation pressure.

We also analyzed immediates within `ld_imm` and `st_imm` instructions using `fp` and no immediate can be encoded with less than 5-bits⁴, as illustrated in Table 4.2c; such instructions highly demand large immediate fields. Note that the stack access in `ld_imm` and `st_imm` is always 4 byte aligned to the accessed data type (as explained in Section 2.2.1) and the encoding of the two least significant bits is needless. Hence, besides implicitly accessing `fp` and `sp`, we can save more space by avoiding to encode such two bits in the 16-bit instruction. A similar approach can be applied to `add_imm`, `save_imm` and `restore` instructions using `sp` or `fp`.

Registers	Usage
<code>g1</code>	16.77%
<code>i0</code>	9.15%
<code>o0</code>	8.97%
<code>o5</code>	6.46%
<code>i6/fp</code>	6.31%

(a) Overall top 5 used registers

Instructions	Usage - <code>fp</code> or <code>sp</code>
<code>ld_imm</code>	30.75%
<code>st_imm</code>	26.53%
<code>save_imm</code>	13.54%
<code>add_imm</code>	13.18%
<code>std_imm</code>	3.64%

(b) Top `fp` or `sp` usage in instructions

Bits	Coverage
1-4	0%
5	12.34%
6	36.89%
7	52.40%
8	79.43%

(c) `ld_imm` with `fp`: Necessary bits to encode immediates

Table 4.2: SPARC register usage statistics

The data in Figure 4.4a disregard `fp` and `sp` usage and show that 8 distinct registers are responsible for 62%, 64% and 81% of the total register usage in instructions from mediabench, MiBench and Linux Kernel respectively. In Figure 4.4b, 67%, 69% and 85% of the total registers defined in all instructions are represented by only 8 distinct registers. The result show that using 3-bits for register fields in the 16-bit extension is a good tradeoff: although 4-bits could be used, we reduce by a factor of four the number of registers and cover roughly 65% of all register usage⁵, leaving one more bit for opcode or immediate.

Furthermore, 27% and 37% of three address instructions in `alu_arith` and `alu_shift` groups behave like two-address instructions, where one of the source registers is the same as destination. In fact, even for instructions addressing only two registers in the `alu_arith` group, 26% of them use the same source and destination register. Hence, two-address 16-bit instructions can be created based on instructions with these profiles.

⁴SPARC ABI reserves a 16 byte space closer to `fp` which had no use in any analyzed binary

⁵Applications needing more registers are likely to have extra spill code, a potential drawback partially mitigated by switching to 32-bit mode

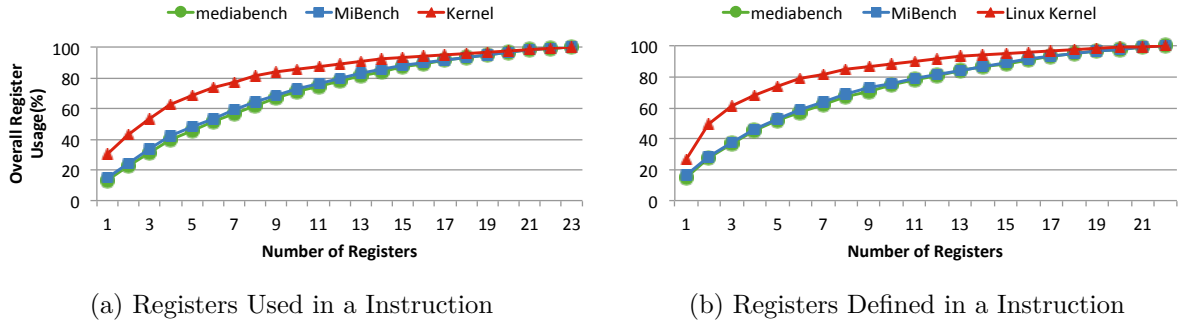


Figure 4.4: SPARC register usage coverage

4.3 Dynamic analysis

We analyzed the execution of SPARC binaries to dynamically collect information about the overall utilization of instructions. We executed all programs from the *MiBench* [43] and *mediabench* [66] benchmarks in the ArchC 2.1⁶ [84] simulator through user process simulation. ArchC has support for SPARC and provides a GCC 3.4 SPARC toolchain. Note that programs are compiled without floating point support, targeting the SPARC ISA.

Table 4.3 shows the list of most executed instruction in mediabench and MiBench benchmarks. In mediabench, `or_reg`, `or_imm` and `add_reg` are the top three most executed instructions while `or_reg`, `add_reg` and `ld_imm` in MiBench. Comparing the static ISA usage from Section 4.2 and the dynamic information obtained, we find some instructions with a low static but high execution count; as mentioned in Section 4.1, we might weight such instructions for possible inclusion in the 16-bit extension - mitigating a potential performance degradation.

For instance, `xor_reg` represents 10% of all executed instruction in MiBench but only 0.1% from all total static count in the same benchmark. The `bpos` instruction is in the same category: 0.03% on static count but 5.3% of the executed instructions in the same benchmark. Other instructions have the same profile and must be specially considered in the inclusion for the 16-bit extension.

4.4 Integer Linear Programming Model

We developed an *Integer Linear Programming* (ILP) model to represent the problem of finding the best field sizes for every instruction format in the 16-bit extension. The ILP

⁶ArchC does not support running the Linux Kernel

mediabench			MiBench	
(11.49%)or_reg	(1.34%)save_imm	(0.29%)addcc_reg	(20.29%)or_reg	(0.41%)or_imm
(7.48%)or_imm	(1.34%)restore_reg	(0.25%)umul_reg	(13.14%)add_reg	(0.30%)bgu
(6.37%)add_reg	(1.32%)ldub_imm	(0.25%)rd_reg	(10.18%)ld_imm	(0.15%)sethi
(5.90%)ld_imm	(0.86%)bneg	(0.18%)xor_reg	(10.0%)xor_reg	(0.09%)nop
(5.12%)subcc_reg	(0.82%)ld_reg	(0.16%)ldsh_reg	(8.87%)add_imm	(0.09%)jmpl_imm
(5.0%)srl_imm	(0.78%)addcc_imm	(0.15%)smul_reg	(6.47%)sll_imm	(0.08%)call
(5.0%)add_imm	(0.73%)ldd_reg	(0.14%)bpos	(5.88%)srl_imm	(0.08%)bleu
(4.98%)subcc_imm	(0.69%)bgu	(0.11%)subx_imm	(5.30%)bpos	(0.05%)bne
(4.03%)sethi	(0.67%)st_reg	(0.10%)xnor_reg	(5.30%)addcc_imm	(0.05%)be
(3.54%)st_imm	(0.64%)sra_imm	(0.10%)ldsh_imm	(4.16%)ld_reg	(0.05%)andcc_imm
(3.01%)and_imm	(0.63%)andcc_imm	(0.09%)sth_reg	(3.10%)st_imm	(0.04%)save_imm
(2.98%)std_imm	(0.61%)sub_reg	(0.09%)lduh_reg	(2.21%)and_reg	(0.04%)restore_reg
(2.91%)bleu	(0.57%)bl	(0.07%)lduh_imm	(1.22%)st_reg	(0.04%)bg
(2.71%)be	(0.51%)ble	(0.06%)stb_reg	(1.06%)subcc_imm	(0.01%)lduh_imm
(2.36%)ldd_imm	(0.48%)addx_imm	(0.05%)subx_reg	(0.74%)andn_reg	(0.0%)wr_reg
(2.35%)sll_imm	(0.40%)orcc_reg	(0.05%)std_reg		
(1.94%)bne	(0.40%)bge	(0.03%)sth_imm		
(1.84%)ba	(0.40%)bg	(0.03%)andcc_reg		
(1.83%)and_reg	(0.38%)ldub_reg	(0.01%)wr_imm		
(1.35%)jmpl_imm	(0.33%)nop	(0.01%)stb_imm		

Table 4.3: SPARC most executed instructions in mediabench and MiBench

solution assisted in the creation and definition of SPARC16 instruction formats.

An input instance to the ILP model consists of a tuple (S, F, I, c) . Each element $s \in S$ is a set of fields. Table 4.4a shows the fields for each of the sets $s \in S$. A field can be a primary opcode, a secondary opcode, a register or an immediate. The primary opcode is mandatory for every $s \in S$. Secondary opcodes and immediates are optional, but limited to one per $s \in S$. Each register in $s \in S$ is restricted to three bits, but its inclusion in a format is optional.

An assignment of size to each of the fields of an element $s \in S$ corresponds to a format. Formats follow two rules: the sum of sizes of each of its fields equals sixteen and the size of a register field is always three. For each $s \in S$, we generated every possible format following the aforementioned rules and placed them into the set F . For example, Table 4.4b presents generated formats for the RRI set - containing one primary opcode, two registers and an immediate. The maximum opcode size is 7 bits which can encode 128 different instructions; usage of more bits to opcode field would reduce available bits to encode registers and immediates.

The set I contains SPARC16 candidate instructions. We took as candidates SPARC instructions and pseudo-instructions. Pseudo-instructions were included because we wanted to hide the existence of certain registers, such as `g0`, `sp`, `fp` and `ra`. Not only does this approach mitigate the impact of having only three bits to index the register bank, but also to increase the size of immediate fields, since the pseudo-instructions reference registers

$s \in S$	Fields			
	1 st Opcode	2 nd Opcode	Register(s)	Imm
I	opc_1			imm_1
RI	opc_1		reg_1	imm_2
RRI	opc_1		reg_1, reg_2	imm_3
RR	opc_1		reg_1, reg_2	
RRR	opc_1		reg_1, reg_2, reg_3	
I2	opc_1	opc_2		imm_4
RI2	opc_1	opc_3	reg_1	imm_5
RRI2	opc_1	opc_4	reg_1, reg_2	imm_6
RR2	opc_1	opc_5	reg_1, reg_2	
RRR2	opc_1	opc_6	reg_1, reg_2, reg_3	

(a) Fields $s \in S$

Formats	Field sizes (bits)			
	opc_1	reg_1	reg_2	imm_3
F1	1	3	3	9
F2	2	3	3	8
F3	3	3	3	7
F4	4	3	3	6
F5	5	3	3	5
F6	6	3	3	4
F7	7	3	3	3

(b) Possible RRI formats

Table 4.4: Description of ILP fields and inputs

implicitly. The U

The cost function $c : I \times F \rightarrow \mathbb{I}$ specifies the cost of mapping $i \in I$ to format $f \in F$. Certain mappings are invalid, as for instance, associating an instruction which needs an immediate to a format that does not have an immediate field. The c function disregards those. The costs were calculated using static analysis data from instructions in mediabench and MiBench programs. For every instruction in those programs, we identified an equivalent in I , and attributed the cost of associating it to every valid format taking into account factors such as the immediate field size being large enough to accommodate constants and attempts to represent three-addresses instructions as two-addresses instructions, forcing a register to be simultaneously as source and destination of an operation.

In order to allow the ILP to discard candidate instructions, the special format 0 was created. The cost of associating $i \in I$ to 0 represents the cost of not supporting i in SPARC16. We calculated that by estimating the number of supported SPARC16 instructions that would have to be executed to achieve the same effect as i . Obviously, this is an speculative value, since the SPARC16 ISA is yet to be determined.

Therefore, the ILP problem can be formulated to the problem of mapping every instruction to a single format while minimizing the total cost incurred in this mapping. The mapping is subject to several constraints that guarantee that the obtained mapping

makes sense - i.e. that the primary opcode has the same number of bits in every chosen format, and that the number of bits attributed to the opcode fields affects the number of instructions that can be placed in a format. Below we describe the ILP model.

We have binary variables x_{if} that indicate if instruction i is going to be mapped to format f or not. We also have binary variables y_f that specify if the format f is going to be used. A format can have at most two opcode fields (a primary and a secondary). There are at most K opcode fields identified by OP_1, OP_2, \dots, OP_K . The primary opcode (OP_1) is shared amongst every $s \in S$, while the secondary opcodes (OP_2, \dots, OP_K) are exclusive. Each opcode has at most L bits. There are binary variables op_{kl} indicating that the k -th opcode uses l bits. There is a special integer variable T that specifies the number of primary opcode available slots. For each $k \in \{2, \dots, K\}$ and $l \in \{1, \dots, L\}$ we create integer variables G_{kl} that state the number of primary opcode slots, among the total T , that will be occupied by instructions mapped to a format that has the k -th opcode with l bits as its secondary opcode. Notice that we can map at most $2^l G_{kl}$ instructions to the format in question. The integer variable G_1 represents the number of primary opcode slots occupied by instructions mapped to a format that has no secondary opcode. We use $f \in s$ to denote that a format $f \in F$ was generated from a set $s \in S$. We use $i \in f$ to denote that instruction $i \in I$ can be mapped to format $f \in F$ and we call the set of formats to which i can be mapped F_i .

$$\text{Min} \sum_{i \in I} \sum_{f \in F_i} c(i, f) x_{if} \quad (4.1)$$

We solve the Equation 4.1, subject to:

$$\sum_{f \in s} y_f \leq 1, \text{ for } s \in S \quad (1)$$

$$x_{if} - y_f \leq 0, \text{ for } i \in I \text{ and } f \in F_i \quad (2)$$

$$\sum_{f \in F_i} x_{if} = 1, \text{ for } i \in I \quad (3)$$

$$y_f + y_{f'} \leq 1, \text{ for } f, f' \in F \text{ inconsistent} \quad (4)$$

$$\sum_{l \leq L} op_{kl} = 1, \text{ for } k \leq K \quad (5)$$

$$y_f - op_{kl} \leq 0, \text{ for each } k \text{ and } l, f \text{ has } op_{kl} \quad (6)$$

$$T - \sum_{l \leq L} 2^l op_{1l} \leq 0 \quad (7)$$

$$G_1 + \sum_{k=2}^K \sum_{l \leq L} G_{kl} - T \leq 0 \quad (8)$$

$$G_{kl} - 2^L op_{kl} \leq 0, \text{ for each } k \text{ and } l \quad (9)$$

$$\sum_{(i,f) \in G_1} x_{if} - G_1 \leq 0 \quad (10)$$

$$\sum_{(i,f) \in G_{kl}} x_{if} - 2^l G_{kl} \leq 0, \text{ for each } k \text{ and } l \quad (11)$$

Constraint (1) assures that at most one format $f \in F$ of each set $s \in S$ is chosen. Constraint (2) establishes that an instruction is assigned to a format only if the format

is chosen. Constraint (3) guarantees that each instruction is assigned to a format (recall that every instruction can be mapped to format 0). Constraint (4) assures that inconsistent formats, i.e. formats that attributed different sizes to the same field, cannot be used together. For example, this guarantees that all the chosen formats will have the same number of bits dedicated to the primary opcode (since OP_1 is shared amongst all formats). Constraint (5) establishes that each opcode field will have a fixed size of l of bits. Constraint (6) says that one format with the k -th opcode using l bits, can be used only if the solution uses that opcode with l bits. Constraint (7) calculates the total amount T of slots for the primary opcode. Constraint (8) guarantees that the number of slots of the primary opcode that are spread among the groups is at most T . Constraint (9) assures that group G_{kl} is going to be used only if in the solution, opcode k uses l bits. Constraints (10) and (11) limit the number of instructions that can be assigned to each format. For these constraints, $(i, f) \in G1$ refers to instructions $i \in I$ mapped to a format $f \in F$ with no secondary opcode, while $(i, f) \in G_{kl}$ refers to instructions $i \in I$ mapped to a format that has k with l bits as its secondary opcode.

4.5 Considerations

The ILP solution produces an initial set of formats; each holds one or more instructions. We also manually introduce other formats and add special instructions which are not considered by the ILP method. For instance, the RI2 format was created to hold instructions dealing with the stack pointer; LDSP, LDSP, STSP, STFP, ADDFP and ADDSP. Moreover, mode exchange instructions (e.g. CALLX) are introduced into ILP generated I format. Additional information on instructions and formats can be found in Chapter 5.

The method described in this thesis was developed together with the former University of Campinas Master student Leonardo Ecco. His Master thesis [32] describes the same method presented here. We both designed and determined the ILP restrictions and the rest of the work was divided as follows: the author implemented the tools necessary for dynamic and static analysis whereas Leonardo Ecco executed and collected the solutions to the ILP method.

Chapter 5

SPARC16

In this Chapter we describe the SPARC16 ISA, the related tools and evaluate the extension. First, SPARC16 instructions are described in Section 5.1; including detail on mode exchange, the **EXTEND** mechanism and code alignment restrictions.

Section 5.2 explains how original SPARC registers are visible in SPARC16; based on that, the resulting SPARC16 ABI is presented in Section 5.3. Sections 5.4 and 5.5 details the hardware implementation and the emulation tools used for SPARC16. The SPARC16 toolchain is presented in Section 5.6, with further implemented optimizations in Section 5.7. Finally, we evaluate the SPARC16 extensions in Section 5.8.

5.1 Instructions

The SPARC16 instructions formats are illustrated in Table 5.1. Every format is uniquely identified by a 5-bit major opcode (*op*). Additionally, a secondary opcode (*op2*) is used in some formats and increases the total number of available instructions.

Format	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
RR	op					op2					rs			rd				
RR1	op					imm					rs			rd				
I	op					imm												
LDST	op					op2	imm				rs			rd				
I2	op					op2	imm											
RI2	op					op2			imm					rd				
RRR	op					op2			rs2			rs1			rd			
RR12	op					op2			imm			rs			rd			
RI	op					imm											rd	

Table 5.1: SPARC16 formats

All SPARC16 instructions, their respective encodings and assembly syntax are described in Appendix B.1.

The **RRR** format does not have an immediate field and is used to encode instructions that operate on three registers - two sources and one destination - such as **ADDrr** (B.1.1) and **SUBrr** (B.1.48). Two-address instructions, like **ADDXrr** (B.1.4) and **ORNrr** (B.1.31), are encoded in the **RR** format.

Arithmetic and logic instructions with immediates use the **RRI** format, examples: **ADDCCri** (B.1.1), **ANDri** (B.1.5) and **SLLri** (B.1.39).

The **RRI2** format, very similar to **RRI**, represents instructions with a low usage rate in original analyzed SPARC programs. **SDIVri** (B.1.37) and **SMULri** (B.1.37) are examples of division and multiplication instructions encoded in this format.

The **ADDFP** (B.1.2) and **ADDSP** (B.1.3) instructions implicitly use the **fp** and **sp** registers, to provide stack offset computations. Both instructions are encoded in **RI2** format.

The **SAVEri** (B.1.36), defined in **I2** format, implicitly uses and defines **sp** optimizing the very common SPARC stack frame allocation pattern **SAVE %sp,imm,%sp**.

5.1.1 Calls and Branches

The **I** format is used to accommodate the **CALL** (B.1.11) instruction, which requires a large immediate field, but no registers. It also holds some branches; *branch always* (**BA** - B.1.8), *if equal* (**BE** - B.1.9) and *if not equal* (**BNE** - B.1.10), with the *annul* [87] field in bit 10. Format **I2** is used to encode branches with other condition codes.

SPARC calls and branches, as described in Table 2.3 (Section 2.2.1), assume 4 byte aligned target and hence discard the two least significant immediate bits before encoding. In SPARC16, such instructions assume 2 byte aligned target, only discarding the least significant immediate bit prior to encoding.

5.1.2 Load and Store

The *load word* (**LDri** - B.1.18) and *store word* (**STri** - B.1.43) instructions are encoded in the **RRI** format. *Byte*, *half*, and *double* loads and stores are encoded in the **LDST** format.

The load and store instructions in the SPARC architecture follow alignment rules, as described in Table 2.2 (Section 2.2.1). The SPARC16 extension enforces the same alignment restrictions by encoding offsets while discarding unnecessary bits. For instance, to load a word from the address $reg + 0x8$, the immediate $0x8$ is encoded as $0x2$, since the word alignment discards the least 2 significant bits from the address displacement. The offset is shifted back accordingly, prior to execution, in order to produce the right address. This encoding strategy reduces the number of bits to encode the immediate.

The **LDFP** (B.1.19), **LDSP** (B.1.22), **STFP** (B.1.45) and **STSP** (B.1.47) load and store instructions implicitly uses **fp** and **sp** as source registers, both are encoded in the **RI2** format.

5.1.3 Mode exchange

Special instructions are used to alternate between SPARC and SPARC16 modes.

From SPARC16 to SPARC code. Two special instructions can switch from SPARC16 to SPARC: *Call with Exchange* (**CALLX** - B.1.14) or *Branch with Exchange* (**BX16**). Changing from SPARC16 to SPARC is useful when: (1) an operation is faster or smaller using SPARC instructions or (2) to reuse SPARC libraries. Figure 5.1a shows how to switch mode using **CALLX**, while in Figure 5.1b, **BX16** is used to switch modes.

```

1  sparc16_code:
2  ...
3  callx printf
4  nop
5  ...
6
7  ; sparc v8 code
8  printf:
9  ...

```

(a) Using **callx**

```

1  sparc16_code:
2  ...
3  bx16 sparcv8_code
4  nop
5  ...
6
7  sparcv8_code:
8  ...

```

(b) Using **bx16**

Figure 5.1: Assembly mode exchange from SPARC16 and SPARC

From SPARC to SPARC16 code. Conversely, *Jump and Link with Exchange* (**JMPLX**) and *Branch with Exchange* (**SPARCV8BX**) instructions are used to switch from SPARC to SPARC16 - see Figure 5.2a and 5.2b.

```

1  sparcv8_code:
2  ...
3  sethi %hi(sparc16_code), %l0
4  or %l0, %lo(sparc16_code), %l0
5  jmplx %l0, 0, %o7
6  nop
7  ...
8
9  sparc16_code:
10 ...

```

(a) Using **jmplx**

```

1  sparcv8_code:
2  ...
3  sparcv8bx sparc16_code
4  nop
5  ...
6
7  sparc16_code:
8  ...

```

(b) Using **sparcv8bx**

Figure 5.2: Assembly mode exchange from SPARC and SPARC16

In Figure 5.3 and Figure 5.4 we show the encoding of both instructions in the SPARC opcode space; they were added to the SPARC ISA by using reserved and unused opcodes. The least significant bit in the 32-bit target address determines the target routine's mode – 0 means SPARC and 1 means SPARC16.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op		unused				sparcv8bx								imm19																	

Figure 5.3: SPARC branch with exchange instruction: `sparcv8bx`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
op		rd				jmplx								rsl				i = 1		imm												

Figure 5.4: SPARC jump and link with exchange: `jmplx`

5.1.4 The EXTEND mechanism

The use of large immediates in SPARC16 instructions is an expensive operation in terms of code size. The reduced immediate field forces the use of several instructions, resulting in a final code size equal or worse than performing the same operation with SPARC instructions. At the same time, the penalty of a SPARC mode exchange to perform a simple constant load is prohibitive.

The SPARC16 define a 16-bit **EXTEND** instruction - a mechanism similar to MIPS16 **EXTEND** [57] - used to increase bit availability when encoding large immediates. It is encoded by a reserved opcode, preceding a regular 16-bit instruction (i.e. not other **EXTEND**) and containing unused extra bits for use by the following instruction.

For example, the **MOV** (B.1.25) instruction has a 8-bit immediate field and can only load constant values up to 255; `mov16 0xff, %i0`. Using the extended version **MOV_ext** (B.1.25), increases the immediate field to 13-bits, allowing constants up to 8191; `emov16 0x1fff, %i0`.

Table 5.2 shows all SPARC16 formats augmented with the **EXTEND** instruction prefix. For instance, formats **I** and **I2** provide ten extra immediate bits to improve target displacement encoding. The mechanism is also used to provide additional registers for some instructions, a functionality not present in MIPS16. For instance, the extended format **RR** transforms a two-address instruction into three address, allowing an additional register source.

Format	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTEND I2	0	1	0	1	1	imm						op		op2		imm																
EXTEND I	0	1	0	1	1	imm						op		imm																		
EXTEND RI	0	1	0	1	1	0	0	0	0	0	0	imm		op		imm														rd		
EXTEND RR	0	1	0	1	1	0	0	0	0	0	0	0	0	rsext		op		op2				imm				rs		rd				
EXTEND RRI2	0	1	0	1	1	imm						op		op2		imm				rs				rd								
EXTEND LDST	0	1	0	1	1	0	0	imm						op		op2		imm				rs				rd						
EXTEND RRI	0	1	0	1	1	0	0	0	imm						op		imm				rs				rd							
EXTEND RI2	0	1	0	1	1	0	0	0	imm						op		op2		imm				rs				rd					

Table 5.2: SPARC16 EXTEND formats

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SETHI						unused	imm																							rd	

Table 5.3: SPARC16 SETHI instruction

5.1.5 SETHI instruction

Additionally, we provide other means of loading large immediates. SPARC16 provides the 32-bit **SETHI** instruction (Figure 5.1.5). The instruction functionality is similar to the one available in SPARC: it loads a 22 bit constant into the higher register bits. To load a constant using the **EXTEND** instruction, we have the maximum of 19 bits and **SETHI** provides a clear benefit over it.

5.1.6 Alignment restrictions

The SPARC16 design and implementation guarantees that **EXTEND**, **SETHI** and all other SPARC16 instructions can be used interchangeably with the minimum 2 byte code alignment restriction; whereas in SPARC, the required instruction alignment is 4 byte. Thus, as shown in Figure 5.5, the 4 byte SPARC16 **SETHI** instruction respects a 2 byte alignment but is not 4 byte aligned.

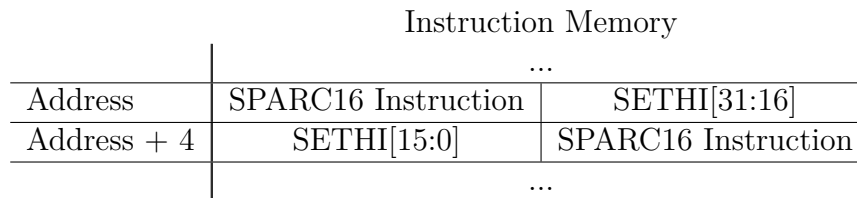


Figure 5.5: Unaligned SPARC16 SETHI instruction

5.2 Registers

The SPARC16 maintain the 32-bit registers from SPARC but only 8 are visible and can be explicitly referenced. To access hidden registers, two special instructions are provided – **MOV8to32** (B.1.26) and **MOV32to8** (B.1.27). The former moves data from a visible register to a hidden one and the latter performs its inverse operation. The **I2** format is used by these instructions with a three bit field *reg8*, used to index a SPARC16 visible register, and a five bit field *reg32*, used to index one of the 32 registers from the SPARC register bank. These instructions are also used to move function arguments into specific registers and to compute more elaborate arithmetic involving **sp** or **fp**. Table 5.4 summarizes register visibility in SPARC16.

Register	Visibility	ABI description
<i>%i0</i>	Visible	Input Register
<i>%i1</i>	Visible	Input Register
<i>%i2</i>	Visible	Input Register
<i>%o0</i>	Visible	Output Register
<i>%o1</i>	Visible	Output Register
<i>%o2</i>	Visible	Output Register
<i>%l0</i>	Visible	Local Register
<i>%g1</i>	Visible	Global Register
<i>%g0</i>	Hidden	Zero Value
<i>%fp</i>	Hidden	<i>Frame pointer</i>
<i>%sp</i>	Hidden	<i>Stack pointer</i>
<i>%ra</i>	Hidden	Return address

Table 5.4: SPARC16 registers

Some of the already mentioned SPARC16 instructions include implicit access to registers **sp**, **fp**, **g0** and **ra**, mitigating the handicap of a small set of visible registers. Also, an implicit register reference means three free bits to encode a larger immediate or more opcodes. Examples of such instructions are **LDFP**, **LDSP**, **ADDFP** and **ADDSP**.

5.3 Application Binary Interface

The Application Binary Interface (*ABI*) defines a set of conventions to be followed by compilers and operating systems to make sure applications and libraries from different sources can interact with each other. Function calling sequence, ELF headers, object files relocations and dynamic linking are examples of conventions defined in an ABI.

The SPARC16 ABI is an extension of the SPARC ABI [53] and inherits most of its conventions. The **o0-o5** and **i0-i5** group of registers are used to pass and receive the first six arguments within a function call, while the remainder arguments are passed on the stack. **MOV8to32** and **MOV32to8** instructions move the arguments to and from the hidden registers **o3-o5** and **i3-i5**. The stack layout remains the same, including the reserved space on the stack for variable arguments.

Regarding the calling convention, no modifications need to be implemented to support interoperability between SPARC16 and SPARC. The only necessary ABI addition relates on the definition of relocations. SPARC16 preserves all relocation definitions from SPARC and additionally defines its own group. Hence, the linker can link between modes since it understands relocations from either SPARC or SPARC16.

5.4 Hardware

SPARC16 is an extension to the SPARC instruction set and it is meant to execute on a regular SPARC pipeline. The SPARC16 instructions are translated to their 32-bit counterparts at execution time by placing a PDC decompressor between the instruction cache and the SPARC pipeline, as shown in Figure 5.6. A PDC design yields more performance than other mechanisms (see Section 2) and is adopted by Thumb [6] and MIPS16 [57].

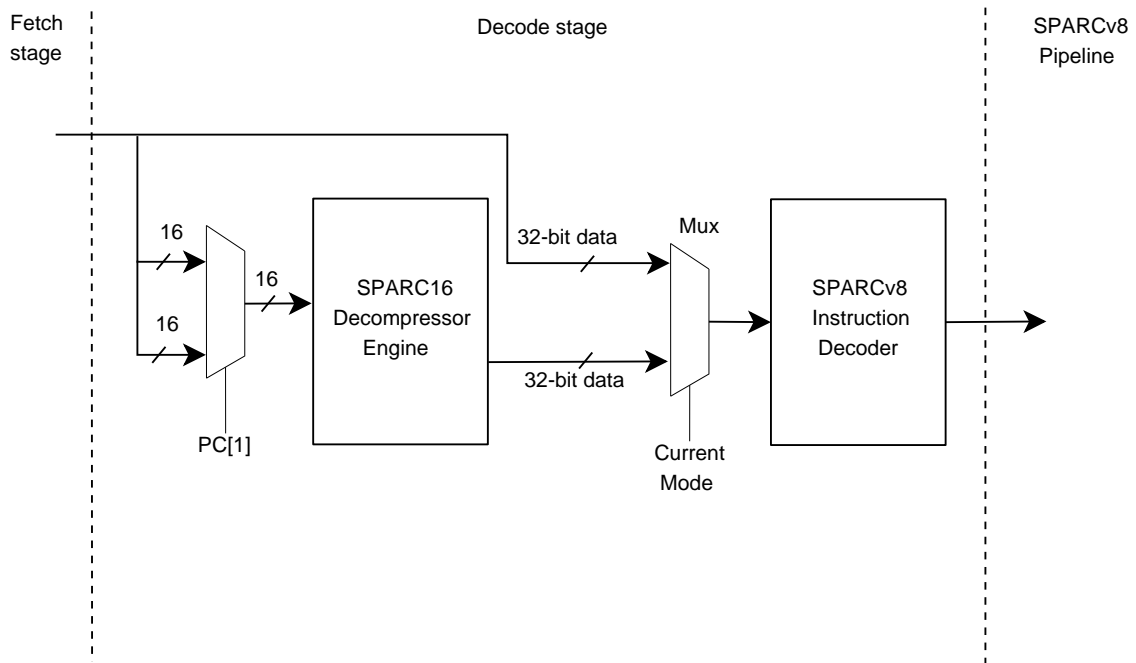


Figure 5.6: SPARC16 decompression diagram

The decompressor is integrated into a SPARC processor Leon3 [38]. The implementation [32] focused on three key factors:

- Integrate the decompressor with the minimum hardware overhead into the Leon3 processor.
- No processor cycle time degradation after integration.
- Guarantees that SPARC16 code is reachable through jumps and calls, even if the target is not 4-byte aligned.

We avoid overhead and cycle time degradation by carefully choosing the bit encoding for each 16-bit instruction, simplifying the conversion between SPARC16 and SPARC.

More details on the hardware design and implementation are provided by Leonardo Ecco in his Master thesis [32].

The implementation is incomplete and was not used in this thesis to test and validate SPARC16 compiled programs.

5.5 Emulator

In order to validate SPARC16 instructions by running real programs and to collect extra information about execution, we used the QEMU emulator [13].

QEMU is a system virtual machine capable of emulating several guest and host architectures. QEMU employs dynamic binary translation as its emulation technique and it first converts fragments of code from the guest application into an intermediate representation. Afterwards, it converts them into native code. Thus, the emulation in QEMU can be divided in three parts, as shown in Figure 5.7: the front-end, the *Tiny Code Generator* (*TCG*) and the back-end.

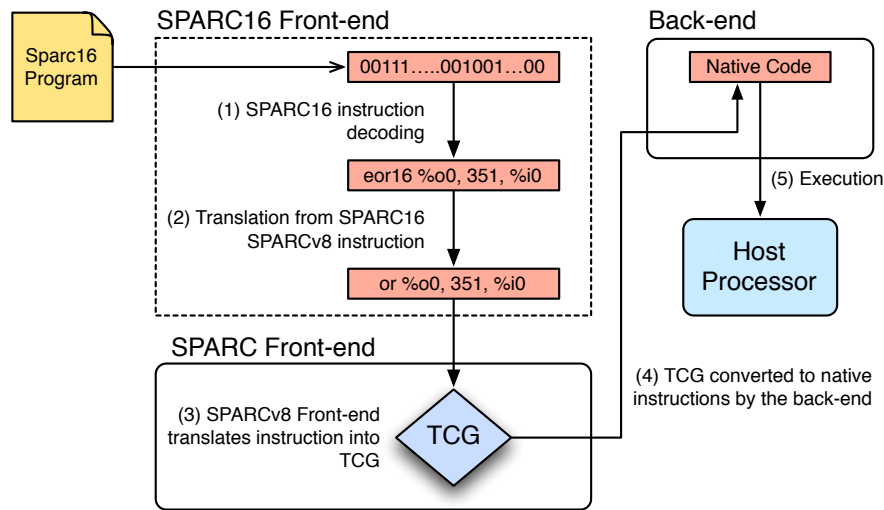


Figure 5.7: SPARC16 program emulation steps in QEMU

The front-end is composed of several target specific functions to parse and translate guest binary instructions into the generic and target independent *TCG* representation. The TCG is further processed and optimized to eliminate any redundancy. Next, the back-end for the underlying host target is invoked and host instructions are generated from TCG and executed. There are front-ends for several targets, such as ARM, MIPS, PowerPC, SPARC, x86 and x86_64. To run and emulate programs on a desktop, the most relevant back-ends are x86 and x86_64.

We developed and integrated a SPARC16 front-end into QEMU. The integration allows the execution of SPARC16 compiled programs, the extraction of execution traces and interoperability with SPARC code mode exchange instructions are fully supported. The QEMU based emulator was also used to evaluate all SPARC16 programs described in this thesis, allowing fast program execution and debugging.

5.6 Toolchain

A toolchain consists of a compiler front-end, back-end, assembler, linker and libraries to compile applications for a given architecture. We developed a SPARC16 toolchain based on LLVM 3.2 [64] from front-end to assembler; GNU Binutils 2.22 [42] for the linker and the uClibc [4] C library compiled for SPARC.

5.6.1 Compiler Frontend and Backend

The LLVM, acronym for *Low Level Virtual Machine*, consists of a compiler infrastructure composed of several tools and libraries. The project is composed by the *Clang* C/C++ front-end, the LLVM intermediate representation (LLVM IR) and target back-ends. The LLVM uses the *pass* concept where transformations and optimizations *passes* are plugged in a central *Pass Manager*, which controls and schedule compilation phases.

Unlike GCC, LLVM allows programs to be compiled and assembled without the need of an external assembler. Once a source code is provided for compilation, the Clang driver invokes the front-end and converts it into the LLVM IR. The LLVM IR is processed by target back-ends, generating assembly or object files.

We implemented SPARC16 support in the LLVM front-end and back-end. The Clang C/C++ front-end was adapted to support SPARC16 since the C language is not target independent - hence the need to express SPARC16 constraints. A complete SPARC16 target back-end library was developed and it is capable of generating SPARC16 object code and assembly files and also to apply SPARC16 specific optimizations.

5.6.2 Linker

The GNU Binutils project contains an assembler, linker and disassembler tools. Target information is unified and used by all tools through the BFD library; hence all target instructions are described using similar data structures which are linked against those tools. We implemented the SPARC16 linker by providing such data structures to Binutils and by writing custom functions to support SPARC16 specific constraints. The data

structures were automatically generated from a SPARC16 ArchC [84] description but other changes were manually introduced.

The approach used considers all external module calls as SPARC code. We manage multiple SPARC16 binaries by merging them, prior to code generation, as explained in Section 5.6.4. Thumb2 and MicroMIPS toolchains provide different mechanisms to allow a similar linking behavior between different modes. Using a custom implementation for SPARC16, we can reuse SPARC compiled libraries in our toolchain.

5.6.3 C Library

A C library provides a set of common functionality to C and C++ programs by known functions and interfaces; `printf` and `strcmp` for example, are widely used functions provided by `stdio.h` and `stdlib.h` library interfaces. That said, a C library is essential to any toolchain. The *uClibc* C library is aimed at embedded Linux devices, is smaller than the *GNU C Library* and was the chosen C library for our toolchain.

We use *uClibc* and compile it for the SPARC architecture; every time a call for a function in the C library occurs from SPARC16 code, we use a mode exchange instruction to accomplish the change. During link time, we rely on our linker to successfully link together the SPARC compiled *uClibc* and SPARC16 code.

5.6.4 The compilation and execution flow

The SPARC16 linker can link one SPARC16 object file with multiple SPARC object files or libraries. However, since during compilation, all external calls are considered as mode exchange, the linker does not support linking several SPARC16 object files. LLVM mitigates this restriction by providing the `llvm-link`; a tool that links multiple LLVM IR files prior to back-end invocation and only one object file is generated. The compilation and execution of SPARC16 programs is illustrated in Figure 5.8. The commands are:

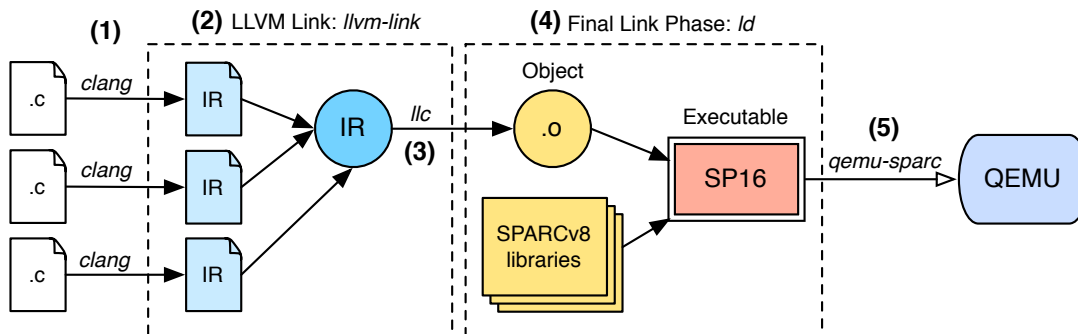


Figure 5.8: SPARC16 compilation and execution flow

1. Each source code file is consumed by the compiler front-end and a LLVM IR file is generated.

```
$ sparc16-clang -c a.c -flto -o a.bc
```

```
$ sparc16-clang -c b.c -flto -o b.bc
```

2. Multiple LLVM IR files are linked together by the `llvm-link` tool into a final LLVM IR module.

```
$ llvm-link a.bc b.bc -o final.linked.bc
```

3. The final LLVM IR is consumed by the target back-end and object code is emitted. The assembler is integrated into the back-end.

```
$ sparc16-llc final.linked.bc -o final.linked.o
```

4. The resulting object code is linked with libraries and the final executable is produced; the executable binary needs to be statically linked to all necessary libraries.

```
$ sparc16-clang -static final.linked.o -o final
```

5. The executable and its arguments are passed via command line to the `qemu-sparc` emulator.

```
$ qemu-sparc final
```

5.7 Compiler Optimizations

The LLVM SPARC16 compiler back-end minimizes program code size by using a set of SPARC16 target specific optimizations. Using the LLVM *pass* interface, optimizations are registered in different phases of the compiler back-end, specially placed where they can extract the best results. In each of the following sections we describe such optimizations and the problem they solve.

5.7.1 Delay slots

The SPARC architecture defines that branches and call instructions require a following delay slot instruction [87]. The delay slot can be filled with any instruction without data or control hazards, or by a simple no-operation `NOP` instruction. In SPARC16, the delay slots are 2 bytes wide and do not support the presence of `EXTEND` instructions.

Problem. The use of `NOP` instructions is suboptimal since other program instructions can be placed in the delay slot, yielding a smaller final program.

Solution. A dedicate SPARC16 pass calculates potential hazards and candidates for the delay slot: program instructions are moved to occupy the slots whenever it is safe, NOP instructions are used otherwise.

1	f :	1	f :
2	mov %o0, %g1	2	mov %o0, %g1
3	cmp %g1, %o1	3	cmp %g1, %o1
4	sub %o0, %o1, %o0	4	ble .LL5
5	ble .LL5	5	sub %o0, %o1, %o0
6	nop	6	add %o1, %g1, %o0
7	add %o1, %g1, %o0	7	.LL5:
8	.LL5:	8	...
9	...		

(a) Regular NOP usage
(b) Delay slot optimization

Figure 5.9: SPARC16 delay slot fulfillment

Figure 5.9a shows a regular NOP placement whereas in Figure 5.9b the **SUBrr** instruction is moved into the delay slot.

5.7.2 Instruction size reducer

All the instructions containing immediates – arithmetic, logic, load, stores, branches and calls – are conservatively emitted by the code generator with the **EXTEND** form; this approach guarantees that instructions supporting bigger constants are selected earlier during code generation.

Problem. It is suboptimal to use **EXTEND** when the immediate can fit in a regular 16 bit instruction.

Solution. In a late, post register allocation pass, the **EXTEND** instructions are removed whenever 16-bit instructions have enough immediate bits to represent the entire immediate. The optimization cannot to reduce the sizes of calls and branches because computing targets and relocations is only possible at assembly and linking time.

5.7.3 Assembler relaxation

The optimization described in Section 5.7.2 cannot handle branches and calls. Since their immediates are used to hold the target displacement relative to the **pc**, one change in the branch or call instruction size, triggers changes in the displacement of all other branches and calls.

1	f:	1	f:
2	eadd %g1, 3, %g1	2	add %g1, 3, %g1
3	eadd %g1, 2, %g1	3	add %g1, 2, %g1
4	eand %g1, 2, %g1	4	and %g1, 2, %g1
5	eor %g1, 11, %g1	5	or %g1, 11, %g1
6	...	6	...

(a) Suboptimal **EXTEND** instructions
(b) 16-bit instructions equivalent

Figure 5.10: SPARC16 instruction size reducer

An assembler relaxation algorithm is capable of solving this problem by iterating indefinitely over functions and reducing instructions sizes. The iteration ends whenever the size of instructions stops changing from one iteration to another.

Problem. **EXTEND** calls and branches are suboptimal in cases where the displacement size is enough to fit in a regular 16-bit instruction.

Solution. The LLVM SPARC16 assembler uses a pass [49] to relax all possible branch and call instructions in a module and reduce their instruction sizes to 16-bit whenever the target displacement fits.

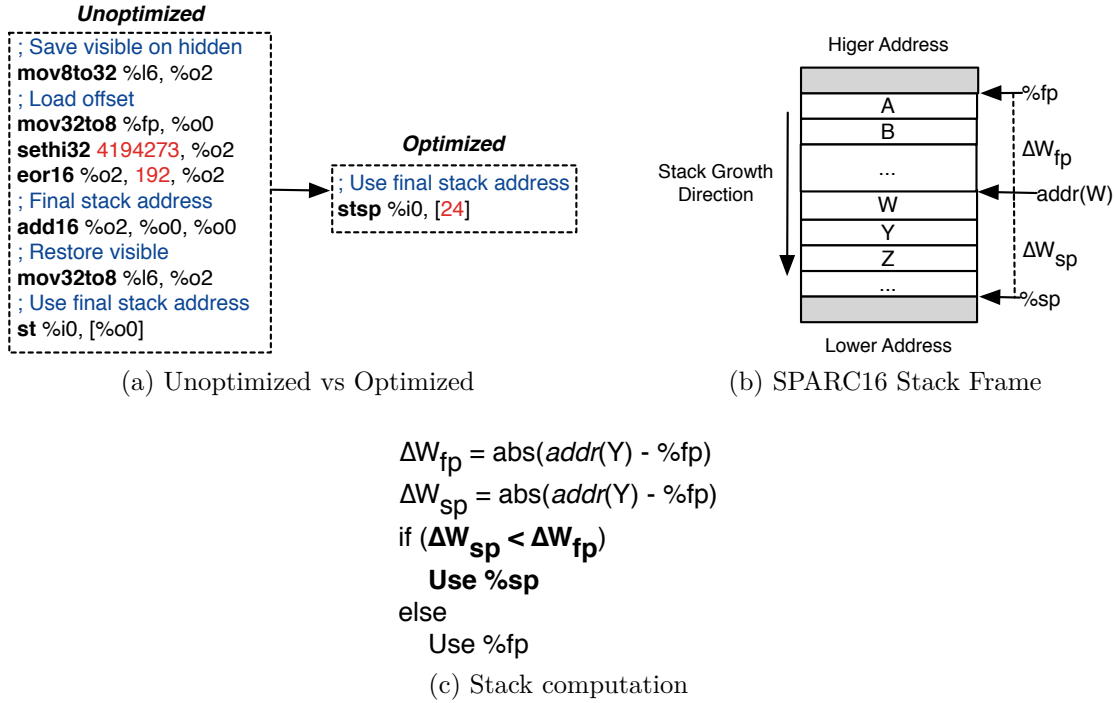
Note that *external* calls - those issuing mode exchange - to other modules have unknown displacements until the link stage and therefore are not covered by the assembler relaxation algorithm.

5.7.4 Mixed Stack Access

Immediate field sizes are heterogeneous across SPARC16 instructions and the **EXTEND** is always used to increase constant coverage. When the immediate needs more bits than **EXTEND** may provide, a **SETHI** plus **ORri_ext** is used, an approach similar to SPARC. We name this group of instructions **SethiEOr**.

However, the use of instructions from **SethiEOr** increases code size and must only be used if no other large approach is found. For instance, when the constant load is related to big stack offsets, additional instructions must be inserted to copy the stack pointer, as illustrated in the unoptimized fragment from Figure 5.11a.

Problem. The loading of large stack offsets is expensive in number of instructions: **LDFP** and **STFP** cannot encode such immediates and extra instructions must be used to obtain copies from the stack frame pointer and **SethiEOr** to load the large immediates.

Figure 5.11: SPARC16 mixed `fp` and `sp` optimization

Solution. We propose an optimization to this scenario, by accessing stack locations from closer locations when possible, avoiding the use of many instructions. The result is shown in the optimized fragment from Figure 5.11a.

The optimization is accomplished by using the nearer available stack pointer register: `fp` or `sp`. In SPARC16, `fp` points to the first stack position while `sp` to the stack end. Suppose the current compiled function accesses the element `W` from the stack. The access to `W` is usually done by computing the offset between `W` and `fp` such that $\Delta W_{fp} = \text{abs}(\text{offset}(W) - \%fp)$.

The ΔW_{fp} is encoded into the `LDFP` or `STFP` instruction immediate field, and `W` is loaded or stored to the stack. Our optimization, computes ΔW_{sp} , the `W` element offset from `sp`, and if $\Delta W_{sp} < \Delta W_{fp}$, uses `LDSP` or `STSP` instructions instead. Figure 5.11b and 5.11c illustrates the stack and optimization offset computation.

The approach is limited to functions which preserve the frame pointer register `fp`, do not use stack *alloca* functions and perform no *dynamic stack reallocation*.

5.8 Evaluation

We evaluate SPARC16 with respect to three key aspects: code compression ratios, instruction cache miss ratios and performance estimation.

5.8.1 Compression Ratios

We measured SPARC16 compression ratio for programs in the *mediabench*, *MiBench* and *SPEC CINT2006* benchmarks. We compile each program using LLVM based toolchains for SPARC16 (Section 5.6) and SPARC. The compiler flag `-Os`, which enables a set of code size optimizations, is used for all programs.

Program	Ratio
rawaudio	71.8%
rawaudio	72.1%
g271	68.7%
gsm	77.3%
jpeg	68.1%
mpeg2decode	75.1%
GeoMean	72.09%

(a) mediabench

Program	Ratio	Program	Ratio
basicmath	81.8%	bitcount	68.7%
susan	81.9%	jpeg	71.0%
lame	82.3%	dijkstra	75.9%
patricia	67.1%	stringsearch	69.2%
blowfish	71.8%	rijndael	77.6%
sha	72.4%	CRC32	69.6%
fft	80.3%	adpcm	72.1%
gsm	77.3%	GeoMean	74.43%

(b) MiBench

Program	Ratio
400.perlbench	71.7%
401.bzip2	72.9%
429.mcf	71%
456.hmmer	74.1%
462.libquantum	69.3%
GeoMean	71.78%

(c) SPEC CINT2006

Table 5.5: SPARC16 compression ratios in mediabench, MiBench and SPEC CINT2006

In mediabench (Table 5.5a), the best compression ratio is 68.1% for *jpeg* and the worst case 77.3% for *gsm*. Table 5.5b shows that MiBench ratios are heterogeneous, ranging from 67.1% in *patricia* to 82.3% in *lame*. Programs from SPEC CINT2006 have a very homogeneous compression ratio, Table 5.5c shows a geometric mean of 71.78%.

The optimizations mentioned in Section 5.7 perform an important role in the compression ratios achieved. The optimizations are responsible, in average, for reducing code size by 20% from SPARC16 programs. Note that we maintain a fair comparison against SPARC since the optimizations applied are only aimed at avoiding compiler mis-usage

of **EXTEND** and 16-bit instructions. Moreover, every LLVM target independent code compaction optimization applied to SPARC16 is also applied to SPARC.

Figure 5.12 shows the overall code size reduction achieved by applying optimizations to unoptimized SPARC16 programs. The *size reducer* and *assembler relaxation* cause a greater reduction in most analyzed programs, whereas the *mixed sp and fp* approach is effective in programs with functions containing large stack frames and many stack accesses.

In mediabench, shown in Figure 5.12a, relaxation reduces code size by an average 7% and the size reducer by 10%. This is roughly the same reduction caused by both optimizations in SPEC CINT2006 (Figure 5.12c) and MiBench (Figure 5.12b). The mixed **sp** and **fp** optimization causes a 2.1% and 2.8% code size reduction in *401.bzip2* and *456.hmm* programs from SPEC CINT2006 while *blowfish*, *susan* and *lame* from MiBench are reduced by 16%, 3.5% and 2.2% respectively.

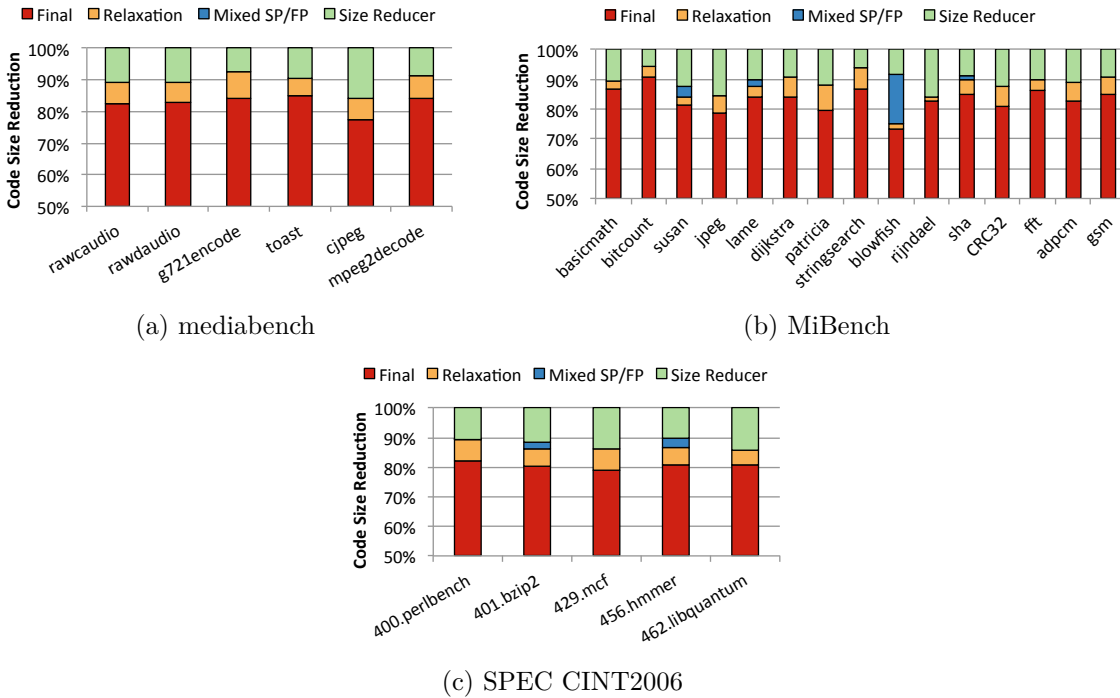


Figure 5.12: Effect of optimizations in code size reduction of SPARC16 programs

Comparison with Thumb2 and MIPS16. We also evaluate SPARC16 compression ratios against Cortex-A9 ARM/Thumb2 and Mips32/Mips16 by compiling programs for both architectures using LLVM 3.3. Figure 5.13 compares mediabench, MiBench and

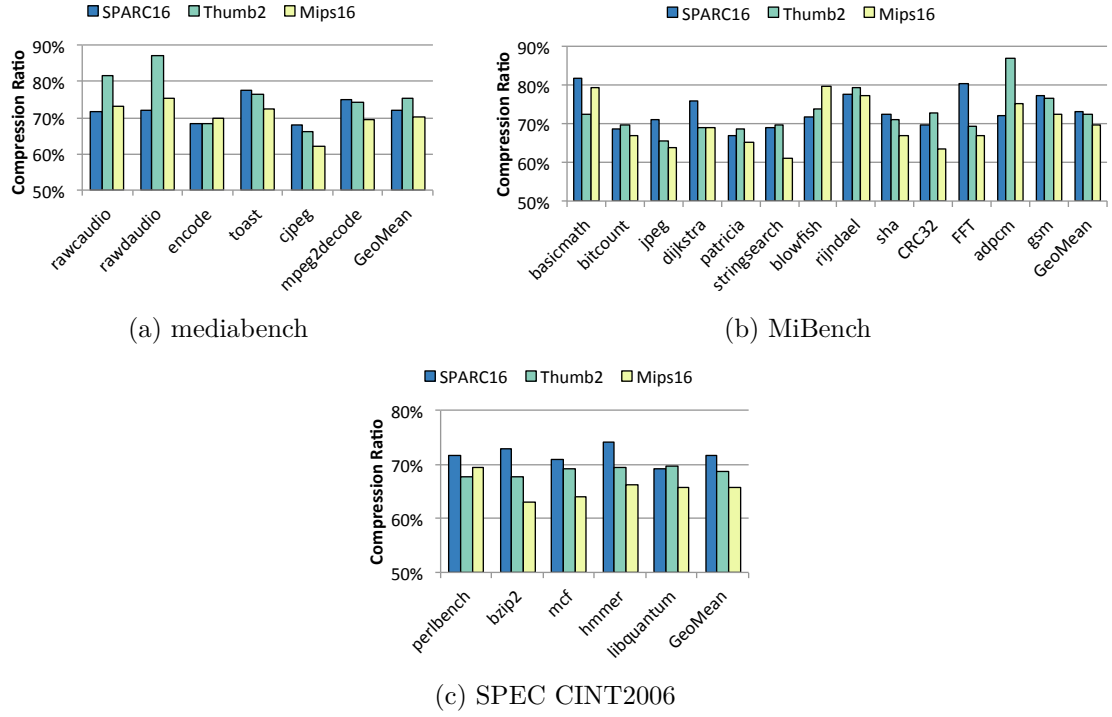


Figure 5.13: Compression ratio comparison between SPARC16, Thumb2 and Mips16

SPEC CINT2006 benchmarks; the geometric mean of all programs shows that SPARC16 achieves similar compression ratios to Thumb2 and Mips16.

Actually, SPARC16 yields better compression ratios than MIPS16 for 50% of Media-Bench programs, and better than Thumb2 for 50% of MiBench programs. For instance, with the *adpcm* program, SPARC16 compression ratio is 15% smaller than Thumb2. Hence, SPARC16 reaches a compression ratio very similar to that of Mips16 and Thumb2, a very interesting result in itself, given the characteristics of both extensions:

- Mips16 and Thumb2 code generation in LLVM are production quality and stable back-end implementations, capable of generating compacted code.
- Both extensions can address PC relative immediates (see Section 2.5.1 and 2.5.2), allowing constants to be placed near functions or in delay slots. This feature allows loading of 4 or 8 byte constants with only one 16-bit instruction.

5.8.2 Instruction Cache Behavior

We evaluate SPARC16 performance by measuring and comparing the impact of SPARC16 and SPARC code on the instruction cache. The reduction of instruction cache misses

improves performance and at the same time reduces power consumption.

We collected¹ execution traces of several programs and analyzed the traces using the Dinero IV [34] cache simulator. We simulated cache sizes from 128 bytes to 32 kbytes, in a 2-way, 32 bytes per cache line².

The dynamic instruction count for SPARC16 is usually higher than SPARC. The SPARC version of *cjpeg*, for instance, dispatches 18M cache demand fetches against 23M in SPARC16 - a 22% increase in the number of executed instructions. As noted, comparing the absolute miss count values would be misleading; thus, we used the cache miss rates.

Figure 5.14, 5.15 and 5.16 compare instruction cache miss rates between SPARC16 and SPARC for the MiBench, mediabench and SPEC CINT2006 benchmarks. In the *rijndael* program, a reduction of 5% in the cache miss rates happens in any cache size between 128 and 4k bytes. Additionally, in a 128 byte cache size configuration, the cache miss rates for *dijkstra* and *gsm* programs are reduced by 9% and 8% respectively.

Two SPARC16 programs have slightly worse miss rates than SPARC: *basicmath* and *CRC32*. The absence of alignment restriction between regular 2 byte and extended instructions in SPARC16 may generate more demand misses; whenever a extended instruction is present across cache lines, an extra cache miss is generated.

5.8.3 Performance Estimation

Using Equation 5.1 we estimate SPARC16 performance against SPARC.

$$\begin{aligned} Cycles &= (Fetches - Misses) + Misses * Penalty \\ Speedup_{sp16} &= Cycles_{v8} / Cycles_{sp16} \end{aligned} \quad (5.1)$$

We chose the *rijndael* program as our case study to investigate speedup estimation. As shown in Figure 5.14, *rijndael* cache miss rates differs significantly between SPARC and SPARC16 for several cache sizes.

Figure 5.17 shows the speedup estimation obtained for different penalty values and instruction cache sizes for the *rijndael* program. In Table 5.6 we present the performance speedup for lines (1) and (2). Line (1), with a 128 byte cache, slows down *rijndael* by 6% with a 10 cycle miss penalty. As we increase the miss penalty from 30 to 150 cycles, a speedup in the range of 4% to 13% is achieved. Line (2), with a 4k byte cache, presents better results; a 1% slow down for a 10 cycle miss penalty and 22% to 48% speedup with the miss penalty in the 30 to 150 cycle range.

¹Measured using SPARC16 and SPARC support in QEMU

²This cache configuration is similar to the cache configuration options available in SPARC Leon3 implementation

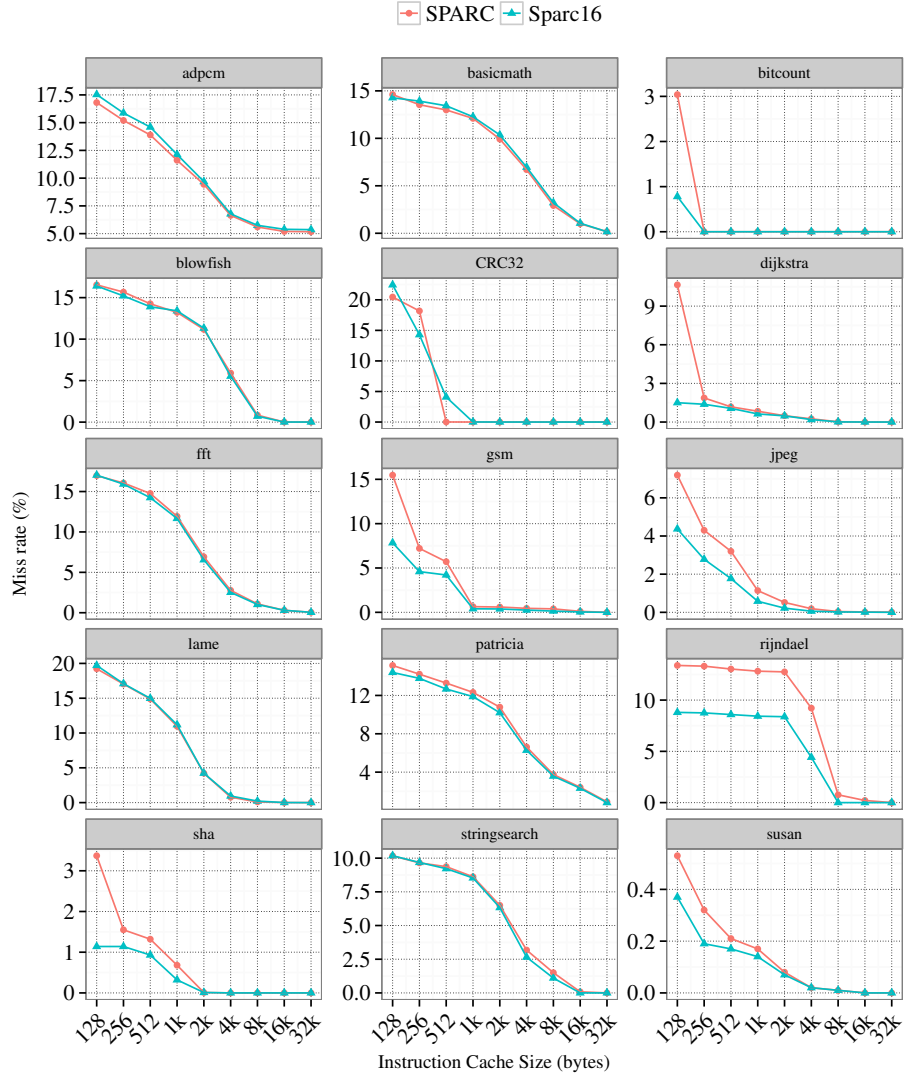


Figure 5.14: MiBench - SPARC and SPARC16 cache miss ratios

The memory access delay is the main reason why SPARC16 performs better than SPARC in several scenarios. In lines (3) and (4), we see a linear relation between speedup and penalty, showing cases where SPARC16 has no cache misses at all.

Moreover, in the 32k - line (5) - configuration, the entire *rijndael* program fits in the cache for both architectures - no capacity cache misses in neither architectures. Since the compulsory number of misses is bigger in SPARC16, it always performs worse than SPARC.

Using the performance estimations, we might design SPARC16 processors with smaller caches; an effective measure to lower the overall chip cost [46]. For instance, the SPARC

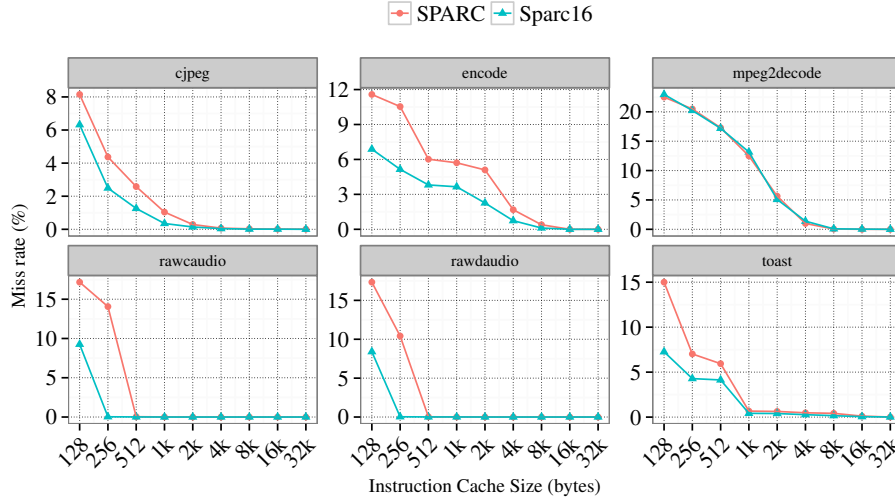


Figure 5.15: mediabench - SPARC and SPARC16 cache miss ratios

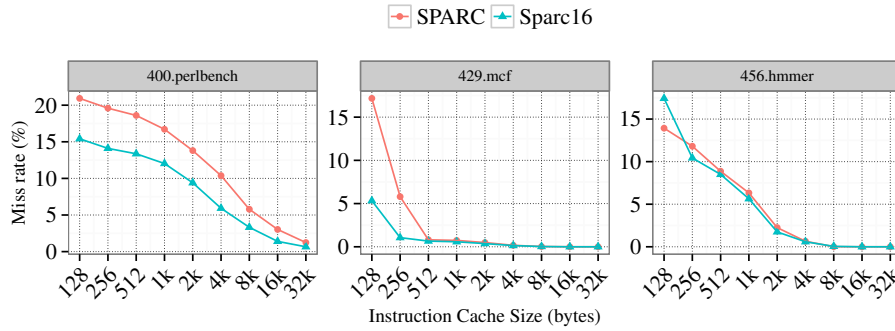


Figure 5.16: SPEC CINT2006 - SPARC and SPARC16 cache miss ratios

processor is present in ASIC cores [74] used in space applications [59, 2]. Hence, such systems are likely to benefit from a 16-bit extension to reduce memory footprint and chip cost.

We illustrate, in Figure 5.18, how small a SPARC16 cache can be when compared to SPARC cache configurations, in order to run the same programs without any performance degradation. Our evaluation considers programs from MiBench, mediabench and SPEC2006; in a 50 cycle penalty miss scenario, 4 programs can run in a 128 byte cache SPARC16 processor with the same performance as that the same 4 programs yield in a 256 byte SPARC processor - a reduction of 50% in size.

The reduction can achieve a factor of 16: one program in 150 cycle penalty configuration can use a 128 byte SPARC16 processor against a 2k byte in SPARC. Therefore, SPARC16 is one alternative for shipping devices with less memory while avoiding performance degradation.

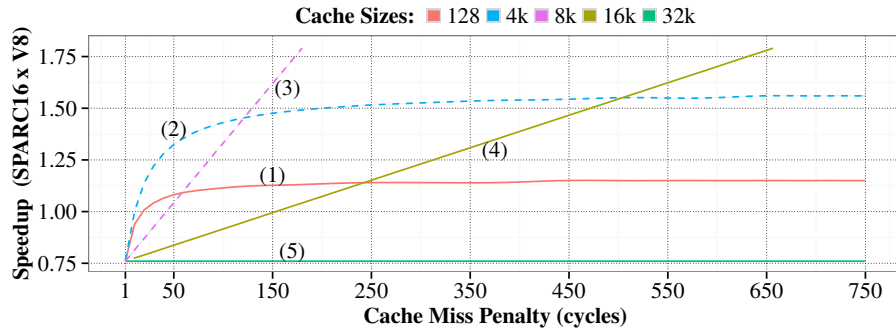


Figure 5.17: SPARC16 speedup against SPARC in MiBench’s *rijndael* program for distinct cache sizes

Miss penalty (cycles)	10	30	50	70	90	110	130	150
128 byte cache	-6%	4%	8%	10%	11%	12%	12%	13%
4k byte cache	-1%	22%	32%	38%	42%	44%	46%	48%

Table 5.6: SPARC16 speedup values against SPARC in MiBench’s *rijndael* program for 128 and 4k byte cache sizes

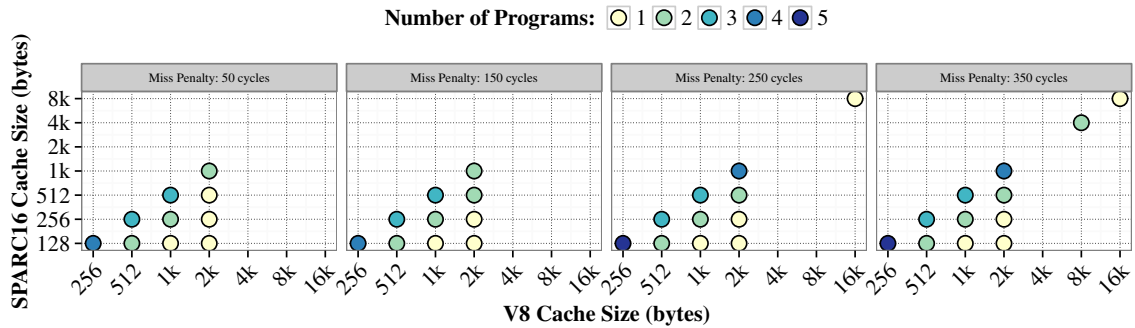


Figure 5.18: SPARC and SPARC16 cache sizes without performance degradation

5.9 Considerations

SPARC16 programs can achieve better compression ratios than other extensions, attaining compression results as low as 67%. SPARC16 also reduces cache miss rates by up to 9%, requiring smaller caches than SPARC processors to achieve the same performance; a cache size reduction that can reach a factor of 16. For several applications, SPARC16 is faster, requires smaller instruction cache size and is a feasible alternative to SPARC and other 16 bit architectures.

As mentioned in Section 4.5, the SPARC16 ISA definition is shared work. The same is valid for the instruction and register design decisions presented in Sections 5.1 and

5.2. Moreover, details on the PDC decompressor design and evaluation can be found in Leonardo Ecco [32] Master’s work.

Finally, the ABI definition (Section 5.3), emulation (Section 5.5), toolchain (Section 5.6), optimizations (Section 5.7) and evaluation (Section 5.8) were done uniquely by the thesis author.

Chapter 6

The X86 Recycling Mechanism

The IA-32 [52] suffers from the ISA aging problem: as the instruction set matures, it is necessary to add new instructions in the already occupied opcode space, and eventually the ISA runs out of space for new opcodes. Also, new processors with old ISAs bear an inherent disadvantage: the variable-length encoding benefits instructions no longer used and penalizes recent additions, since the shortest encodings are already taken by the instructions introduced first.

We show how to overcome the harmful effects of expansion characteristic of aged ISAs. We seek a novel approach to maintain an ISA that is as efficient as a newly designed one in terms of code compaction and decoder size, while still being backwards compatible with the oldest software developed for it. To reach this goal, we propose in this Chapter the use of a recycling mechanism for the IA-32 ISA that allows selected short opcodes to change their previous functionality to serve a new, more popular, instruction.

We start our analysis in Section 6.1 by radically re-encoding the x86 ISA using three different approaches. The general recycling mechanism is described in Section 6.2; the hardware and software implementations are described in Sections 6.3 and 6.4. In Sections 6.5 and 6.6 we discuss security implications and limitations of our approach. The mechanism evaluation is presented in Section 6.7, showing the experimental results. Finally, specific considerations about the research are given in Section 6.8.

6.1 Radical Approaches

In this section, we evaluate radical approaches to create space in an old ISA: completely re-encoding it, possibly removing unused instructions. We will discuss such alternatives for the x86 ISA and the impact they would have.

6.1.1 (A) Reduce all Operation Codes to 2 bytes

Instead of having a variable size operation code, we encode all operation codes into 2 bytes, still leaving space for future improvements. Notice that registers and immediates remain with the original encoding. The net result is an ISA with up to 64K instructions with plenty of room for future improvements.

6.1.2 (B) Reduce all Operation Codes to 1 or 2 bytes

Similar to the previous approach, but uses 1-byte to the 240 most used instructions and 2-bytes for the others. We specifically chose the 240 1-byte instructions after evaluating the resulting compression ratios when changing the number of 1-byte instructions from 1 to 255, for different program sets.

As shown in Figure 6.1, we re-encode all programs¹ in *Ubuntu 12* and *Windows 7* operating systems using two different sources for most used instructions:

- (i) A collection of Operating Systems: Ubuntu 4, 6, 7, 8, 10 and 12 and Windows 95, 98, XP, Vista and 7
- (ii) The SPEC2006 benchmark.

In both scenarios we found that the diminishing return point is around 50 1-byte instructions; we decided to use 240 because instead of the maximum number allowed (256) to leave encoding space for future expansion.

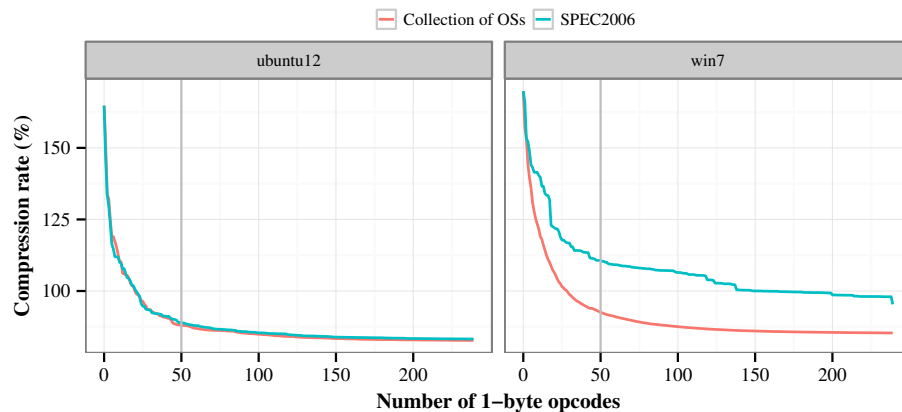


Figure 6.1: Windows 7 and Ubuntu 12 re-encoded using instruction frequency from (i) a collection of operating systems and (ii) SPEC2006 programs

¹All libraries, kernel and application code available in a standard OS release

As previously stated, the registers and immediates are encoded as originally. The net result is an ISA with up to 4336 instructions, with plenty of room for future improvements.

6.1.3 (C) Convert to a RISC-like ISA encoding

This extremely radical change will lead to all instructions having a 32-bit encoding. For the sake of simplicity, we consider the ARM instruction set as the evaluation ISA here; we use GCC 4.7 for the *ARMv7* architecture instead of x86 compilers.

6.1.4 Evaluation

All three encoding schemes will create binary compatibility problems. We present them here as exploratory evaluation. Maintaining binary compatibility would require a huge effort on virtual machine, binary translation, and compiler infrastructure, just to mention a few challenges.

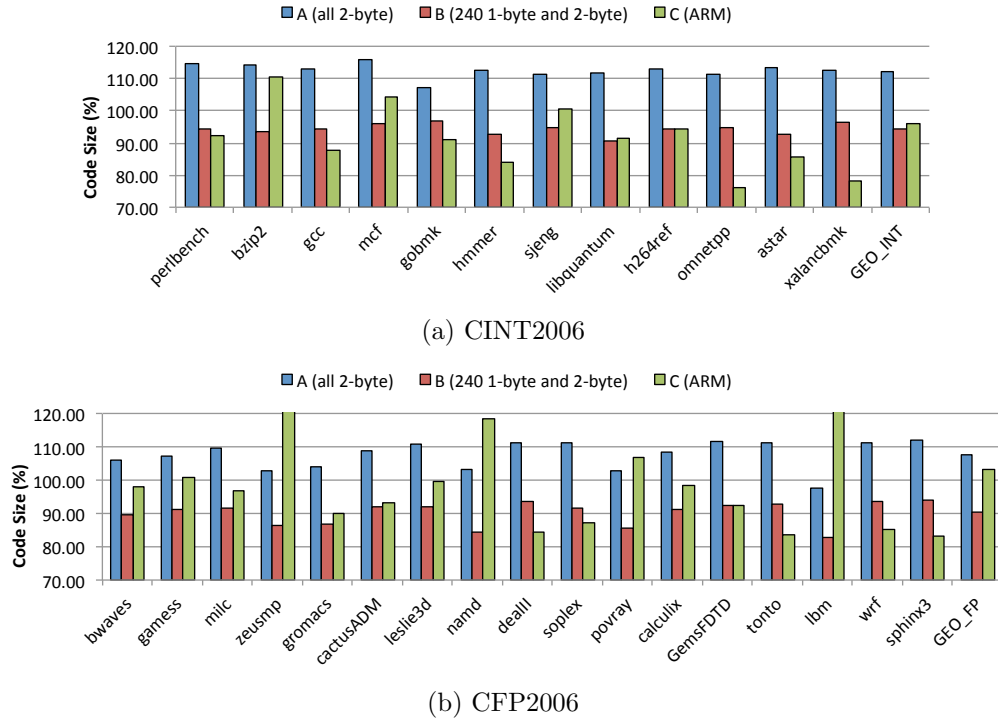


Figure 6.2: SPEC CPU2006 re-encoded by 3 radical changes to the x86 encoding

We evaluated the three proposed methods with the SPEC 2006 benchmark². In Fig-

²In the compiled SPEC2006 programs using the (B) approach, we consider the most used instructions coming from (ii)

ure 6.2, we show the compression ratio of re-encoded programs in scenarios (A), (B) and (C) against the original x86 encoded programs - 100% baseline.

The main finding is that x86 code size is larger than its ARM counterpart for most of the programs, giving to ARM the status of a RISC ISA with a smaller footprint than a CISC. Approach (B) shows that a variable operation code size can still yield a better encoding than a simple RISC (ARM) and the current ISA, but requires 1 byte operation codes. We re-encoded 5 SPEC 2006 programs using (B) and simulated the I-cache impact using a 32KB-4way³ configuration in the Dinero IV [34]. The cache miss reductions were: 7% for *mcf*, 5% for *omnetpp*, 49% for *lbm*, 6% for *astar*, and 19% for *milc*.

We also evaluated operation code re-encoding of all programs in Windows 7 and Ubuntu Linux 12. Figure 6.3 shows the operation code reduction when considering the (A) and (B) CISC re-encoding scenarios with the most used instructions coming from (i) and (ii).

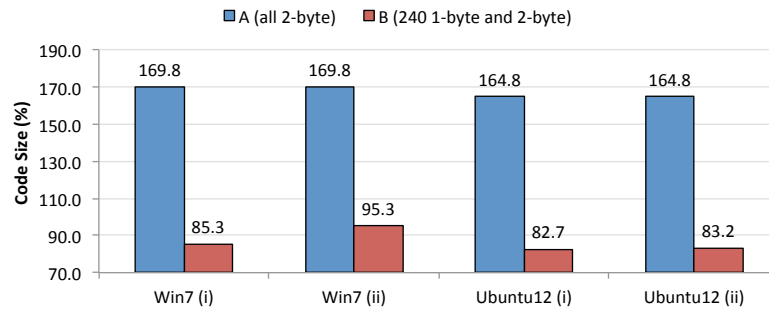


Figure 6.3: Windows 7 and Ubuntu 12 re-encoded by approaches A and B using most used instructions from sources (i) and (ii)

6.1.5 Re-encoding and Backward Compatibility

Applying a re-encoding method such as the approach (B) would increase x86 compressibility; with the positive effect of cleaning up the operation code space while improving instruction cache usage. However, the approach requires a complete backward compatibility breakage with older x86 compliant processors, a prohibitive side effect. In Section 6.2 we propose the recycling mechanism; a re-encoding approach that maintains backward compatibility.

³An usual cache configuration available in Intel Core-i7 processors

6.2 Recycling mechanism

This section discusses the concepts required to implement a recycling mechanism, implementation details will be given in Sections 6.3 and 6.4. The removal of outdated and unused instructions in CISC has two important advantages:

1. Opens room for encoding new instructions with less bits, improving program size and cache utilization;
2. Reduces the complexity of the processor implementation.

We introduce the term *processor revision* (*PR*) to denote a specific version of a CPU implementation and *software revision* (*SR*) to identify the PR a program is compiled for.

6.2.1 Instruction lifetime cycle

An OC⁴ is an intrinsic part of a processor and is not expected to change across processor revisions. An instruction is considered transient and can be imagined as incarnating an OC, for a short or long period of time, but always eligible to be removed or replaced by a new one. The time frame in which a specific instruction uses an OC is called *lifetime*.

The *recycling mechanism* is the process when an instruction departs from an OC, ending its lifetime, and is subsequently replaced by a new one. Hence, we provide OC reuse throughout sequential processor revisions. The process can repeat multiple times, with an OC spanning several different lifetimes. Recycling is the only way to change a PR, for example, the x86 ISA only has one PR throughout its history, since no OC was ever recycled.

An *outdating* stage - a time-frame dedicated to officially announce the departure of an instruction from an OC - must happen towards the end of an instruction lifetime. It guarantees enough development time to compiler writers and operating system vendors to properly update their products. This stage is not strictly necessary regarding the implementation process, but is highly encouraged. Figure 6.4 details the lifetime of instructions and the recycling process, highlighting the three key components, namely:

Creation: a new instruction incarnates an OC in an arbitrary PR_X . The OC may come from an invalid or reserved opcode – never used by any instructions – or previously freed from an instruction. The adoption of a new instruction takes time: usage by compiler intrinsics, vendor libraries and staging new compiler back-ends. After a long software evolution cycle, the new instruction is incorporated into programs shipped to the users.

⁴Operation Code

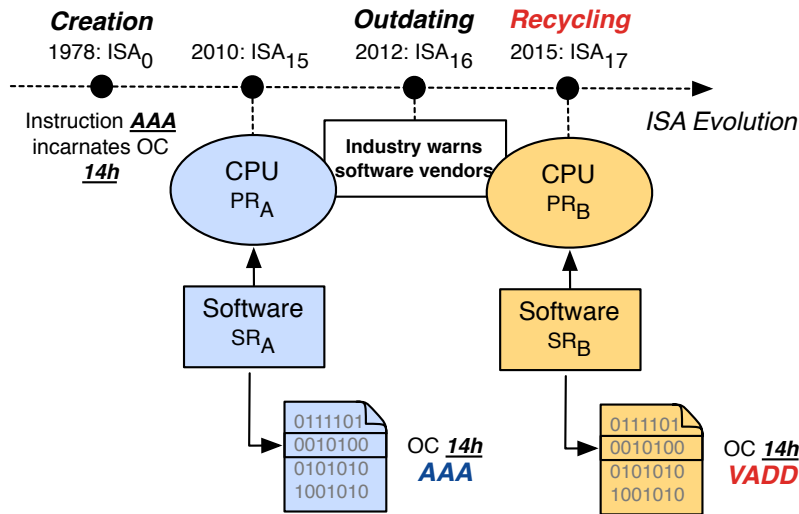


Figure 6.4: X86 operation code (OC) reuse and lifetime

Outdating: at a given time, industry propose new features, which may superseded previous functionalities provided by older instructions. Also, some highly specific processor capability – like the IA32 **AAA** instruction – becomes obsolete. Thus, somewhere between PR_A and PR_B , the manufacturer chooses to deprecate the instruction and announces it to the community. After this announcement, compiler manufacturers remove the support for that instruction, and operating systems vendors start working on emulation routines for that instruction.

Recycling: an instruction is removed from the OC in PR_B , and a new one is inserted for an incarnation, re-using the same OC. Note that recycling and creation steps occur together and in the same processor revision.

6.2.2 Operation Code Revisions and Orthogonality

Processors employ thousands of OCs internally. Operation codes are independent between each other and each has its own *Operation Code Revision (OCR)*. The OCR keeps track of how many instructions or set of lifetimes the OC spanned throughout its existence. In Figure 6.5, the OC 14h (OC_{14h}) has an $OCR_{14h} = 0$ prior to any recycle, whereas $OCR_{14h} = 3$ after its incarnated by **VADD512**.

This brings orthogonality to the whole mechanism since different OCs can be reused at any given time, by any new instruction, without any dependence or constraints between each other. Moreover, we define the OCR as responsible for changing PRs; at least one OCR is necessary to be changed in order to increase a processor revision. Instructions

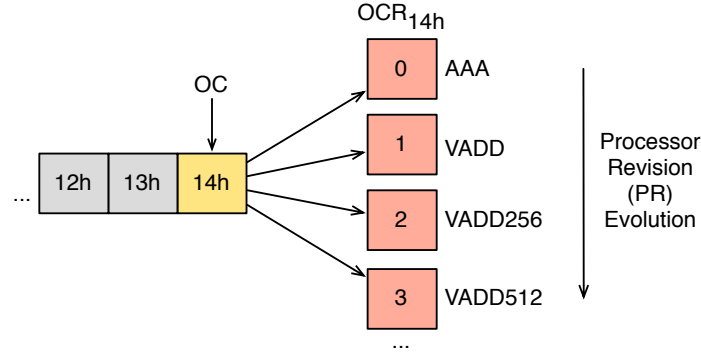


Figure 6.5: Operation Code Revision

incarnating an OC for the first time do not change the OCR.

Consider, in Figure 6.6, that the instruction **AAA** is recycled in PR_B , after an outdating period in PR_A . When recycled, the OC_{14h} is released from **AAA** in PR_A and the instruction **VADD** incarnates. The OCR_{14h} value, which is 0 in PR_A , is updated to 1 in PR_B .

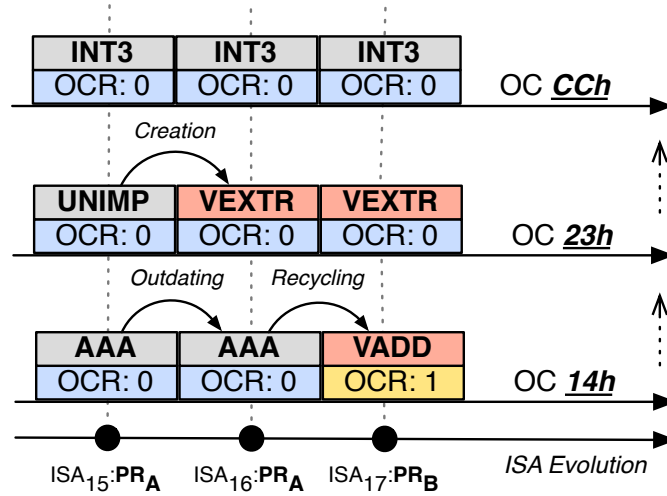


Figure 6.6: The x86 operation code (OC) orthogonality and OCR's

Since OCs are orthogonal, OC_{CCh} , OC_{14h} and OC_{23h} can incorporate different instructions regardless of other OCs. Note that OCs incarnated by the first time do not have their OCR updated. For instance, OC_{23h} is unimplemented in $ISA_{15}:PR_A$ but is incarnated by **VEXTR** in $ISA_{16}:PR_A$; the OCR and PR value is unchanged.

Sequential integers are used to illustrate the OCR concept, but these are not necessary by the implementation. More details are presented in Section 6.3.

6.2.3 Backward compatibility

The execution of old software may lead to compatibility problems in processors using recycled operation codes. In the example from Figure 6.4, if a program in SR_A is executed in CPU PR_B a problem arises: whenever the OC_{14h} is fetched, it is executed as the **VADD** instruction, not **AAA**, breaking backwards compatibility.

Our mechanism solves this problem by providing the old instruction behavior using a software emulation mechanism via CPU generated traps. The SR is added in each software binary, always matching the desired target PR during compilation time. Thus, back to our example, when SR_A is executed in PR_B , an emulation routine implements **AAA** behavior; the routine is called by a trap handler whenever OC_{14h} is fetched during program execution.

Section 6.4 describes the necessary software support for emulation.

6.2.4 Revision Vector and Trap Mask

The *Revision Vector* (RV) is the set of OCRs that uniquely identify a PR. Given PR_x , RV_x , the Revision Vector RV_x is defined in Equation 6.1, where each element OCR_i , denotes the current OCR value in PR_x for OC_i .

$$RV_x = \{OCR_0, \dots, OCR_N\} \quad (6.1)$$

The \ominus operator, in Equation 6.2, defines an *exclusive or* between two elements.

$$\begin{cases} A \ominus B = 0 & \text{if } A = B \\ A \ominus B = 1 & \text{if } A \neq B \end{cases} \quad (6.2)$$

The \otimes operator, in Equation 6.3, defines an *exclusive or* between elements in the same position from different sets:

$$\begin{aligned} X \otimes Y &= \{x_0, \dots, x_n\} \otimes \{y_0, \dots, y_n\} \\ &= \{x_0 \ominus y_0, \dots, x_n \ominus y_n\} \end{aligned} \quad (6.3)$$

Applying the operator \otimes between two RVs, yields the \ominus difference between each pair of elements. The result is called the *Trap Mask* (TM), defined in Equation 6.4.

$$\begin{aligned} TM_{x,y} &= RV_x \otimes RV_y \\ &= \{OCR_{x,0}, \dots, OCR_{x,n}\} \otimes \{OCR_{y,0}, \dots, OCR_{y,n}\} \\ &= \{OCR_{x,0} \ominus OCR_{y,0}, \dots, OCR_{x,n} \ominus OCR_{y,n}\} \end{aligned} \quad (6.4)$$

The *Revision Vector* and *Trap Mask* are used to implement the emulation mechanism. $TM_{x,y}$ gives the list of all OCs that require emulation when running SR_x in PR_y , where $x < y$.

The algorithm that decides whether a trap is needed by a given OC_w in SR_x against PR_y is listed in Algorithm 1.

Data: Given, SR_x , PR_y and OC_w

if $x < y$ **then**

$TM_{x,y} = RV_x \otimes RV_y$;

if $TM_{x,y}[w] = 1$ **then**

OC_w needs emulation

end

end

Algorithm 1: Trap mechanism algorithm

Example. Consider, for simplicity, an ISA with only 3 OCs; 0, 1, 2. Two sequential revisions A and B, have the following revision vectors: $RV_A = \{3, 0, 4\}$ and $RV_B = \{3, 1, 4\}$. To check whether emulation is needed when running SR_A in PR_B , the $TM_{A,B}$ is computed:

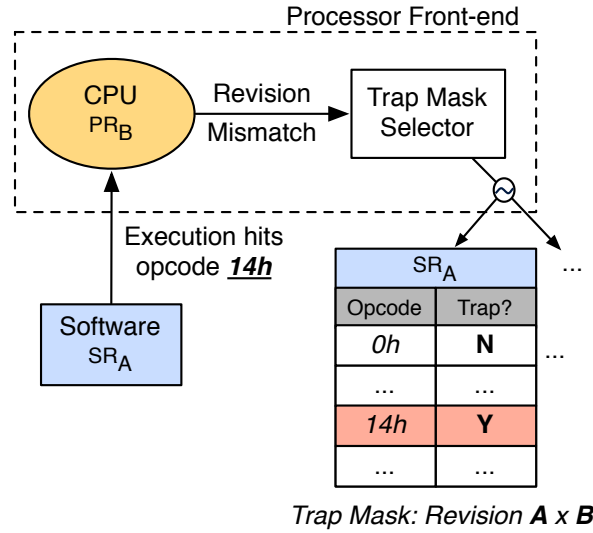
$$\begin{aligned}
 TM_{A,B} &= RV_A \otimes RV_B \\
 &= \{3, 0, 4\} \otimes \{3, 1, 4\} \\
 &= \{3 \ominus 3, 0 \ominus 1, 4 \ominus 4\} \\
 TM_{A,B} &= \{0, 1, 0\}
 \end{aligned}$$

Thus, when software with SR_A is executed in PR_B , the trap mask $TM_{A,B}$ indicates by $TM_{A,B}[1] = 1$, that the OC_1 needs emulation and a trap is generated. The software handling the trap is then responsible for providing emulation routines. By carefully choosing the instructions to retire, manufacturers can keep this emulation overhead low. The mechanism is also illustrated in Figure 6.7.

Additional Definitions. The operator \oplus is defined as a *logical or* between two bits. Given two sequential but not successive PRs, PR_a and PR_e , the operator *Pack* is defined as:

$$\begin{aligned}
 Pack(PR_a, PR_e) &= TM_{a,e} \oplus TM_{b,e} \oplus \dots \oplus TM_{d,e} \\
 &= \{ae_0 \oplus be_0 \oplus \dots \oplus de_0, \dots\}
 \end{aligned} \tag{6.5}$$

We also define the operation $|S|$ on set S , which counts all non zero elements in set S .

Figure 6.7: CPU generated traps via *Trap Masks*

6.2.5 Trap Mechanism

For a given PR_y , all SR_x where $x < y$, need proper emulation support. A feasible implementation approach is to efficiently support the most recent SRs, and older ones at a higher cost. In Section 6.7.4 we evaluate the trap mechanism performance.

Effective Revision Index and Active Revision Vector. We use 4 bits, called the *Effective Revision Index (ERI)*, to select the right trap mask for any given SR_x executed in PR_y , where $x < y$. Although we described how $TM_{x,y}$ can be computed for given x and y , we store all supported trap masks and reference them by using the *Active Revision Vector (ARV)*.

The ERI is used to index into the ARV and select the right trap mask. Since the ERI is 4-bit wide, it can only address the current revision, 14 previous ones, and all the other collapsed. The Algorithm 2 describes the trap mechanism using ERI and ARV.

Revisions older than the previous 15 ones, are collapsed into the same ARV position, $ARV[15]$. Hence, the $ARV[15]$ points to a trap mask which is a superset of all trap masks from older and currently unsupported revisions. The trap mask superset is computed with the *Pack* operator, defined in Equation 6.5. The cost of emulation for older revisions via $ARV[15]$ is higher, since $|Pack|$ is large, thus more instructions to emulate.

Example. In Figure 6.8, we execute a program with SR_{50} in a processor with revision PR_{60} . The computed ERI yields $ERI = 60 - 50 = 10$, the position to index into ARV: $ARV[10]$. The returned trap mask is $TM_{50,60}$. Additionally, if we execute SR_{40} in the

Data: Given, SR_x , PR_y and OC_w

if $x < y$ **then**

$ERI_{x,y} = y - x$;

 /* ERI is 4-bit wide

*/

if $ERI_{x,y} > 15$ **then**

$ERI_{x,y} = 15$

end

$TM_{x,y} = AVR[ERI_{x,y}]$;

if $TM_{x,y}[w] = 1$ **then**

OC_w needs emulation

end

end

Algorithm 2: Trap mechanism algorithm with ERI and ARV

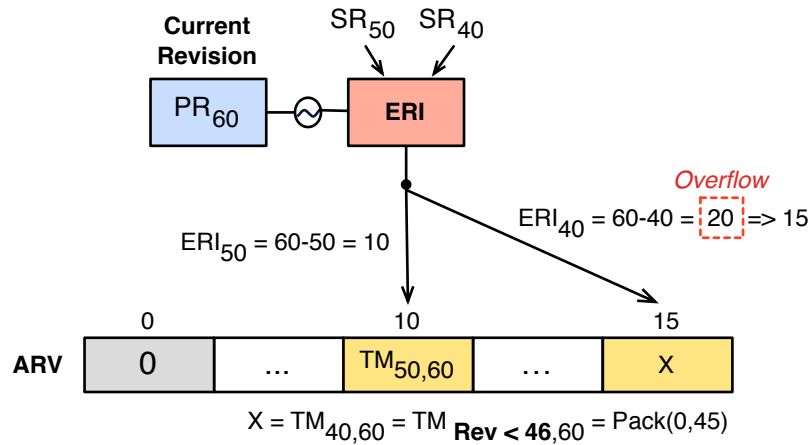


Figure 6.8: General trap mechanism using the Active Revision Vector

same processor, the computed ERI is $ERI = 60 - 40 = 20$. However, since the limit is 16 ARV positions, the effective result is $ERI = 15$, indexing into the last ARV position, $ARV[15]$. The resulting trap mask for $ARV[15]$ is the collapsed trap masks of all old and unsupported revisions: $TM_{40,60} = Pack(PR_0, PR_{60-15=45})$.

In the ERI and ARV scheme proposed above, TMs are pre-computed and hardwired into the mechanism. A more flexible implementation can allow the OS itself to provide TMs to ARV and to specify which revisions must be mapped to each ARV position.

6.3 Hardware

This section proposes one hardware implementation to the instruction recycling mechanism and discusses its implications. We use the x86 ISA as the case study and the previous experience on the 32-bit version to exemplify some important facts. The presented ideas can be applied to other architectures.

We must design the processor hardware with enough information to allow the processor to decide whether an instruction should be emulated or not. To accomplish this, the processor must be aware of the version associated with the currently running software and decide whether to use the trap mechanism.

There are many ways to implement, ranging from the creation of a new instruction – informing the processor the version of the next instructions (instruction granularity) – to a modified virtual memory architecture that embeds the version information in each memory page (page granularity). The instruction granularity implies the usage of a custom instruction just to signal the processor that new code is about to run. Thus, to avoid the necessity of adding glue code between each library call that potentially changes the software version, we present here the virtual memory approach.

6.3.1 Page Table extension

We expand page table entries to include the SR of the code in the ERI form along with access protection bits – translation look-aside buffers (TLB) must also cache this information. Linkers, then, can request to the loader the allocation of different pages for code with different revisions.

Figure 6.9 shows the current IA-32e 4KB page table entry format modified to support revision numbers (see Figure 2.9 for the original format). The figure shows that we can use bits 62 down to 59, to encode a 4-bit version number because these bits are currently ignored. Even though we could use more bits, ERI 4 bits are enough to encode the ISA version because we can always merge revisions through their TM. The instruction TLB must also expand each entry to hold the new 4-bit version number. For example the Intel

6.3.3 Verification

Any new piece of hardware increases the verification effort. It is not different with instruction recycling but the mechanism itself can reduce the overall verification effort by reducing the μ ROM size. However, verification must now consider ERI and ARV. Also, by tackling the ISA increase over the time, we are saving future efforts on front-end verification.

6.3.4 ISA Domain Specialization

By carefully removing instructions and creating a new revision, it is possible to drastically reduce the ISA size, allowing the entrance in new markets where the x86 complexity is considered too big. Alternatively, ISA extensions could be created for specific domains without bloating other processors in the family by providing a simple emulation library to OS vendors.

6.4 Software

The software portion is most notably composed of the emulation code layer that supports the execution of removed instructions, but is not restricted to it. Linkers, loaders and the system call API need to be aware of code versions in order to correctly fill out modified page table entry structures. The SR should be annotated in the file, in a field in the ELF file header. All unannotated files should be considered as SR_0 .

6.4.1 Assembler and Linker

The *Assembler* must be updated to support the new instruction using recycled OCs. The system *Linker* must keep track of the code version of each linked module used to build the final executable. If there is a SR mismatch between two different modules to be linked together into the final executable, the linker must allocate them to different pages, each one augmented with its own SR. No modules with different SR can be put together into the same page. To ease this task, intermediary object files should bear this distinction as well, and assemblers should recognize a new directive that specifies the module's SR.

6.4.2 Operating System Loader

The *Loader* recognizes the SR information for each loadable segment present in the binary envelope (PE, ELF, etc.) and allocates memory pages with page table entries are marked with the corresponding SR. To this extent, memory page allocation system calls must also

be expanded with the SR parameter. Finally, the absence of SR information assumes the most recent revision.

6.4.3 Emulation Routines

Emulation routines mimic the behavior of removed instructions, with important restrictions: the use of outdated and removed instructions must be avoided, emulation cannot implement the behavior of instructions that change internal processor registers, including instructions bearing special characteristics, such as atomicity, which cannot be reproduced via software. For instance, `clflush`, the flush cache line instruction, cannot be emulated and thus cannot be replaced. However, the number of such instructions is small.

Aside from *how* the emulation routines are implemented, another important problem is *where* the routines reside. It is possible to place the emulation code either in the machine firmware (system BIOS) or in emulation drivers that are loaded as early as possible by operating systems. The first alternative has the advantage to emulate old instructions from the operating system itself, since the routines do not rely on the operating system to be loaded, but are already present in the firmware.

In this thesis we chose to use the operating system to handle the emulation traps. After receiving the emulation trap, the operating system decodes the instruction, checks for the software revision, and dispatches the execution to the instruction implementation. We expect, as briefly mentioned in Section 6.2.1, that every operating system uses the same implementation for the same instruction and this implementation should be made available by the processor manufacturers to guarantee compatibility.

On the other hand, really old operating systems would not be able to run on new processors because they may use removed instructions. We do not expect this to be a problem because neither are current platforms capable to run old operating systems for a similar reason: they lack drivers to support modern hardware. As we pointed out before, modern hardware manufacturers, such as Asus, Gigabyte and Evga, do not provide drivers for old operating systems.

Notice that, by emulating instructions in software, in the long term, we may end up having more emulated instructions than real ones, which is not a problem since removed instructions will be seldom used.

6.5 Security implications

By changing the ERI/SR of one page or file, one can change the behavior of software. Notice that changing a memory page or tampering with a file may already require higher

privileges on a machine, allowing the user to harm the system in other ways. We leave the security implication of the proposed method to be analyzed in future work.

6.6 Limitations

One limitation of this mechanism is the place to store the emulation libraries and how to distribute them. We have discussed alternatives through firmware and operating system and both requires a good communication channel to provide new implementations of recycled instructions to guarantee backward compatibility. By relying on software vendors, it is also possible for them to deliberately not support one specific revision on their systems, restricting the software to run in that OS. Last, there may appear intellectual property issues when moving one implementation from hardware to software.

6.7 Evaluation

This section presents - through an extensive static and dynamic analysis of x86 executables, a study of the instructions that stopped being used over time and how this provides input to the recycling mechanism. We show the benefits of this improved operation code space with an analysis of the decrease in the code size and instruction cache misses when frequent AVX instructions are re-encoded into smaller, recycled operation codes, and finally concludes with an investigation of the performance impact of emulating instructions that were removed from hardware.

6.7.1 Methodology

We measured and generated the x86 data for the static analysis using two different disassemblers: the Agner's *object file converter* [36] tool and the disassembler library of the **Bochs** virtual machine [65]. We used them as libraries in a higher level tool designed for the purposes of our measurements. To collect dynamic execution data, we used the **Bochs** virtual machine.

We organized a number of virtual machines, each containing a complete 32-bit x86 software environment from a specific year. Table 6.1 depicts the software systems release date and contents. We focused on analyzing common software used by people at home or in the office, and we studied static and dynamic frequencies of x86 instructions of different types and their evolution in time both in Windows and Linux desktops.

For example, our first Windows-based environment uses the Windows 95 operating system, the Internet Explorer 3 browser, and the Office 95 productivity suite to show how x86 software from 1995 to 1996 used the IA-32 instruction set.

Static	Dynamic	Release	Operating System	Additional Software
Yes	Yes	1996-1997	Slackware Linux 3	Netscape 4.0.1, StarOffice 3.1
Yes	Yes	2003-2004	Ubuntu 4.10	Firefox 0.9.2, OpenOffice 1.1.2
Yes	No	2005-2006	Ubuntu 6.10	
Yes	No	2006-2007	Ubuntu 7.10	
Yes	Yes	2007-2008	Ubuntu 8.10	Firefox 3.0.3, OpenOffice 2.4
Yes	No	2009-2010	Ubuntu 10.10	
Yes	Yes	2011-2012	Ubuuntu 12.04	Firefox 11, LibreOffice 3.5
Yes	Yes	1995-1996	Windows 95	I.E. 3, Office 95
Yes	Yes	1998-2000	Windows 98 SE	I.E. 5, Office 2000
Yes	Yes	2001-2004	Windows XP SP2	I.E. 6, Office 2003
Yes	Yes	2007-2009	Windows Vista	I.E. 7, Office 2007
Yes	Yes	2010-2012	Windows 7 SP1	I.E. 8, Office 2010

Table 6.1: List of x86 operating systems and software

The static analysis uses an instruction crawler that analyzes the entire virtual machine disk image for executable files; we also focus on 32-bit mode x86 and not on 64-bit instructions. In the static analysis, single instructions are cataloged even though they may never execute.

On the other hand, for the task of dynamic instruction frequency measurement, we established a common set of activities to be performed by an user and the virtual machines were executed with a modified version of **Bochs** that records a histogram for executed instructions. A detailed list of the executed tasks used to obtain the execution traces follows:

1. Operating system boot;
2. Start a text processing application and perform a systematic set of text formatting and editing actions on the *Alice in the Wonderlands* text document;
3. Start a spreadsheet application opening a CSV file with 10,000 lines per 20 columns of floating-point data. Sort the rows, plot a scatter graph and perform linear and log regressions;
4. Open an internet browser with a saved HTML file with contents from a news site;
5. Decompress a 90MB zip file;
6. Shutdown the system.

Using the steps above, we captured how programs of different systems and releases used the x86 ISA to perform common user activities. Our goal was to show the effect of time in instruction type usage in order to detect operation codes no longer used and present the instruction life cycle of a real instruction set. The dynamic analysis does not consider Ubuntu 6, 7 and 10 because **Bochs** cannot run them.

6.7.2 Static Analysis

We used static analysis to gather the following information about instructions:

- The total number of 32-bit mode operation codes present in all virtual machines from Table 6.1.
- OCs that were never used in any of the analyzed binaries.
- Instructions that become obsolete over time.

Total Operation Codes Recorded in All Disks. The total number of unused operation codes in all disks is 505, Almost the size of the entire x86 ISA in 1993. This means that considering all 1646 analyzed x86 prefix plus operation code combinations, about 30% of them were never found in any virtual machine disk we scanned. From this count, we excluded 149 combinations that use the **48h** prefix, which requires the 64-bit mode, because our analysis focuses on 32-bits virtual machines.

Type	Number of unused operation codes			
	3 Bytes	4 Bytes	5 Bytes	6 Bytes
AVX	3	61	5	0
SSE	74	238	7	1
Other	40	76	0	0
Total	117	375	12	1

Table 6.2: Number of unused x86 operation codes by size. There were no unused 1 and 2 bytes operation codes.

Table 6.2 shows the number of unused instruction operation codes by size. The Table shows the number of these instructions that belong to vector extensions, because albeit unused, there is a high chance these operation codes may be used in the future, as they are still in adoption. SSE category includes all Intel SSE extensions and AMD SIMD extensions. AVX considers AVX and AVX2 extensions. The others include the MMX extension.

If we consider for reutilization an operation code that is never found in any of these disks, we are limited to reuse operation codes of at least 3 bytes of a MMX instruction. On the other hand, it is also valid to consider for recycling operation codes that may appear in the disk but are never truly executed, and this will be discussed shortly, in the dynamic analysis.

Usage of Instructions Over Time. In this analysis, vector extensions were not considered because they belong to a large category of instructions that are still in adoption, and we need to present a separated analysis for them. We also did not consider privileged instructions.

Figures 6.11a and 6.11b show the number of different operation codes used in Linux and Windows systems over time. As expected, the number of used operation codes increases because software is absorbing new instructions. Examples of instructions that were not used before but started to appear in the disk at this time scale include: `vmclear` and `vmptlrd` virtualization instructions, `xadd` exchange and add instructions, which had its usage increased thanks to the rise of multiprocessing systems, `xchg`, for the same reason, and several `cmov`, conditional move, variants, which were first introduced in Pentium Pro.

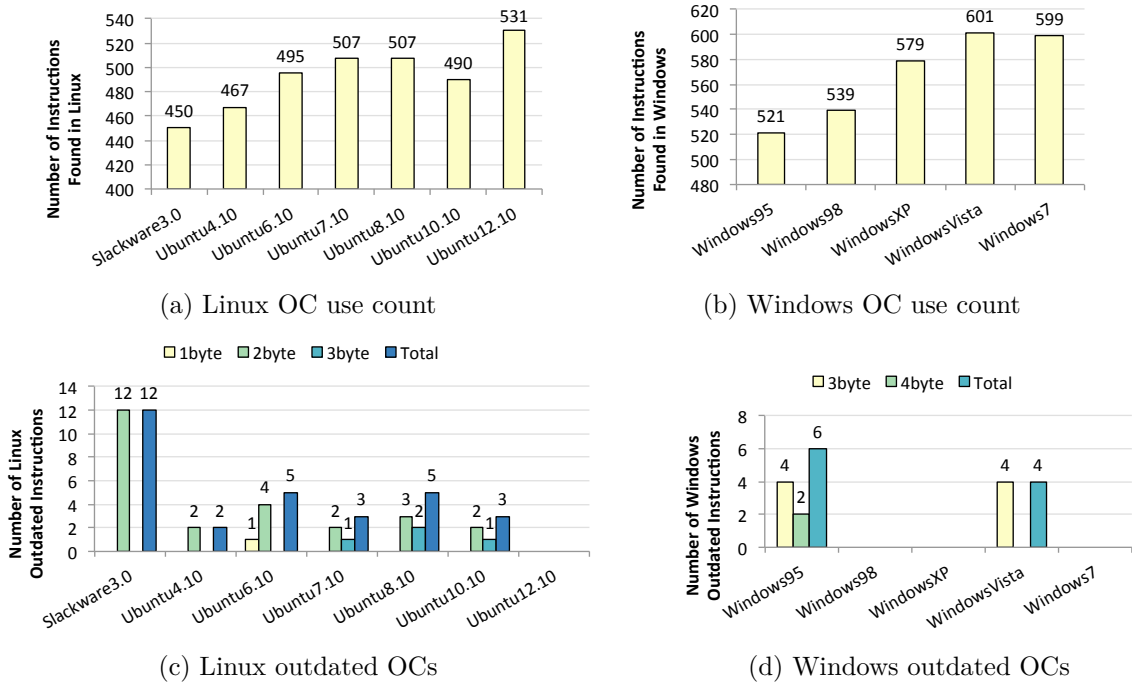


Figure 6.10: Linux and Windows OC count and outdated OCs over time

When the crawler sees an operation code in a given year, for example, 2004, and no longer can find it in any other subsequent year (2006 up to 2012), it marks this OC as

unused or outdated. Figure 6.10c and 6.10d shows the number of outdated OCs by its size for Linux and Windows; the Slackware bar shows that 12 2-byte operation codes were last seen in 1996. This means that future software releases stopped using these instructions. Not surprisingly, some outdated operation codes discovered by our static analysis were also deprecated in Intel IA-32e 64-bit mode, including `les`, load far pointer using ES, `push` and `pop` using ES or CS. They were outdated starting with Ubuntu 7.

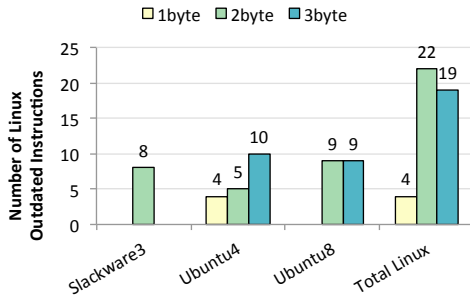
The most important OCs for recycling are the smallest ones because of their potential for providing code compaction. As these chart shows, Ubuntu 6.10 was the last release to use a 1-byte operation code instruction, the `les` instruction. It is specially important to remove a 1-byte instruction, because we may use this opcode as an escape code to encode 256 new 2-byte instructions.

6.7.3 Dynamic Analysis

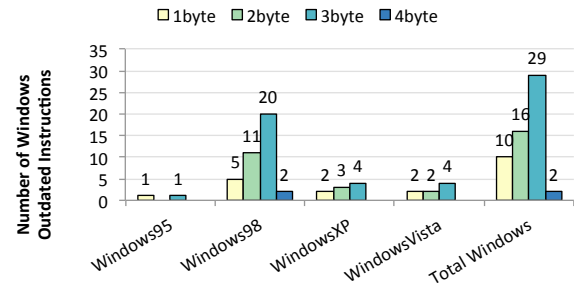
The dynamic analysis is less conservative than the static analysis. Even though the dynamic analysis is powerful enough to count instructions generated by self-modifying code as part of a JIT engine, for example, this category of instructions is small compared to the amount of operation codes that are present in disk images but are never used at runtime. Besides, the amount of used operation codes at runtime is tightly dependent on the selected workload. For instance, the instruction `rdtsc` would never execute in workloads that are not using any kind of self-performance measurement. Nevertheless, the dynamic analysis is important to reveal which OCs run on a set of programs and input activities. As in the previous static analysis, vector, privileged and 64-bit instructions are not considered.

Figure 6.11 presents the same charts from the static analysis that indicate outdated instructions over time, but this version shows the dynamic count of the number of OCs that stopped being used in future releases. For example, the Windows 98 count revealed that it used 38 operation codes that were not found in the dynamic traces of future releases. Notice that in the dynamic analysis, the number of outdated instructions is much larger than that found in the static analysis, suggesting that although instructions may stop being used, they may still appear in software.

In Windows based systems, there were 10 1-byte operation codes that stopped being used from Windows 95 to Windows 7. This gives much more room for recycling: we could remove these instructions from hardware and encode an astonishing number of 2560 new 2-byte OC instructions in their place, removing the need to have 3 to 6 bytes OCs. Another option would be to encode 10 1-byte instructions with high compaction. Observe that many operation codes that were not found to be outdated in the disk (static analysis) in Windows, were outdated in the dynamic analysis. This suggests that Windows may



(a) Linux outdated instruction count



(b) Windows outdated instruction count

Figure 6.11: Linux and Windows dynamic outdated instruction count over time

keep many old libraries in disk for compatibility. For Linux based systems the results are slightly more modest: 4 1-byte outdated operation codes from Slackware 3 to Ubuntu 12.

To show that instructions have a well-defined lifetime, in which programs frequently use them in the past and then cease their usage afterwards, Figure 6.12 shows a dynamic instruction frequency histogram in logarithm scale for 2 traces: the Windows 95 (1995) workload and the Windows 7 (2010) workload.

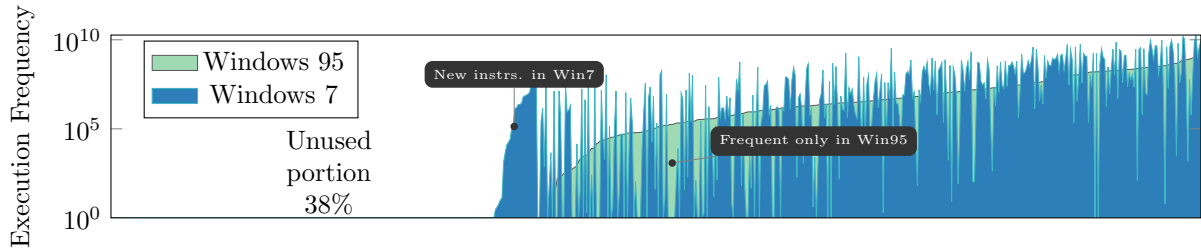


Figure 6.12: A dynamic instruction frequency histogram sorted with respect to Windows 95 instructions usage, compared to Windows 7 instructions usage, in logarithm scale. Spikes show differences in usage pattern.

The abscissa records different instruction types and the ordinate shows the usage frequency of the instruction. The operation codes are ordered by frequency count in the oldest workload. The fact that the Windows 7 frequencies appear spiked with many low bars is evidence that many popular instructions in Windows 95 had a very different usage pattern 15 years later. They either stopped being used or had their usage reduced.

6.7.4 Performance Impact

Using an analytical model and an execution experiment, we estimate how performance degrades with the emulation of deprecated instructions.

Analytical model. We built an analytical model based on dynamic execution traces of several operating systems and SPEC CPU2006 benchmark. In this model, we add a penalty, in number of instructions, for each occurrence of removed or recycled instructions in these traces.

We define I_{total} as the total number of executed OCs in these scenarios. I_{live} is the set of all executed instructions which are not candidate for removal or recycling and are not subject to generating traps.

I_{trap} , described in Equation 6.6, is the total number of executed instructions with penalty overhead P considered for each removed or recycled instruction. The number of times a specific OC_x is executed is given by the function $E(OC_x)$. $T = t_0, \dots, t_n$ is the set of each OC_t that needs trap, for every $t \in T$.

$$I_{trap} = I_{live} + P \times \sum_{t \in T} E(OC_t) \quad (6.6)$$

The executed instructions increase ratio R_{exec} generated by the emulation is defined in Equation 6.7.

$$R_{exec} = \frac{I_{trap}}{I_{total}} \quad (6.7)$$

Assuming a penalty of 200 extra instructions ($P = 200$) for emulating each removed instruction, Figure 6.13 shows how incrementally moving x86 instructions from hardware to software emulation impacts performance while providing ISA operation code space reuse.

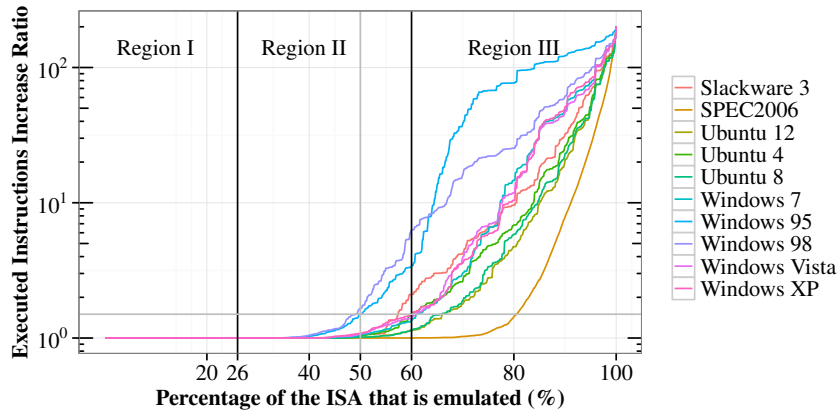


Figure 6.13: Performance overhead relative to the percentage of the ISA that is emulated with emulation penalty of 200 instructions.

We also sorted the abscissa in ascending order of the sum of the frequencies of all

traces so that the least used instructions are removed first, and the subsequent ones are removed as the x axis increases, in a cumulative fashion that provides a total removed percentage of the ISA. In this analysis we did not consider multimedia extensions because it would incorrectly report them as a large percentage of unused instructions, when in fact they are still in adoption by recent software.

For example if 50% of the x86 ISA is emulated in this fashion, the model shows that the execution of the Windows 95 and 98 workloads would suffer a 50% increase in the number of executed instructions – 1.5 ratio. These operating systems achieve higher ratios prior to the others because they use instructions that we removed first.

Figure 6.13 is divided into three important regions: Region I ranges from 0% to 26% and contains only instructions that do not appear in any investigated trace, Region II goes up to 60% and is composed of instructions that may be removed with a tolerable overhead depending on the emulation penalty, and the Region III encompasses only heavily used instructions that would add an unacceptable amount of overhead regardless of the emulation speed. In Region III, there is also a second order effect not considered in our analytical model: as most of the ISA was removed, the emulation penalty becomes higher because the emulation routines are constrained to use a restricted set of instructions.

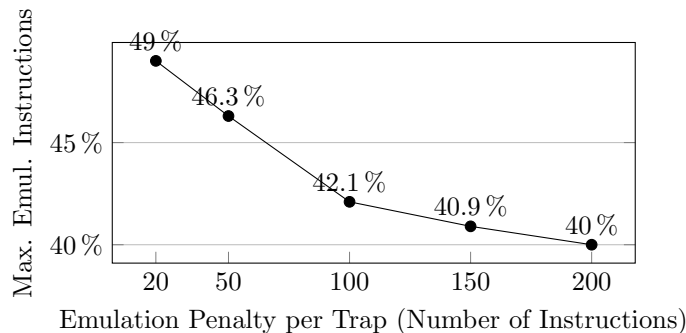


Figure 6.14: Maximum ISA emulation ratio when tolerating up to 5% of overhead and using different emulation penalties.

A closer look into the data represented in Region II from Figure 6.13, shows that when admitting a maximum of 5% of overhead, 40% of the ISA can be emulated. We compute the model for other penalty values $P \in 20, \dots, 200$ and show, in Figure 6.14, how much percentage of the ISA we can emulate when tolerating a 5% overhead. The result also indicates a tradeoff between more efficient emulation routines and the percentage of the ISA that could be emulated.

Emulation experiment. We implemented a Linux kernel module that installs a x86 *interrupt handler* for interrupt `3h` - used as a software debugging trap by issuing instruc-

tion `INT3` - to emulate the behavior of selected x86 instructions. We also patched the SPEC CPU2006 binaries to replace the first byte of these selected instructions with the `INT3` opcode that uses only one byte, `CCh`. The selection contemplates variants of the `mov` instruction that store immediate values in a register and the `cmp` instruction that performs a comparison between the `AL` register and an 8-bit immediate value.

When running the SPEC programs and the next instruction is either the `mov` or `cmp`, the processor generates a trap because the instruction was previously patched with the `INT3` opcode. Then the kernel module queries a shadow image of the binary that contains all original instructions prior to the patching and retrieves the original instruction. We used the `Bochs` decoder to detect the instructions, jump to and execute emulation routines and return to the next instruction in the program. Figure 6.15 illustrates the emulation mechanism.

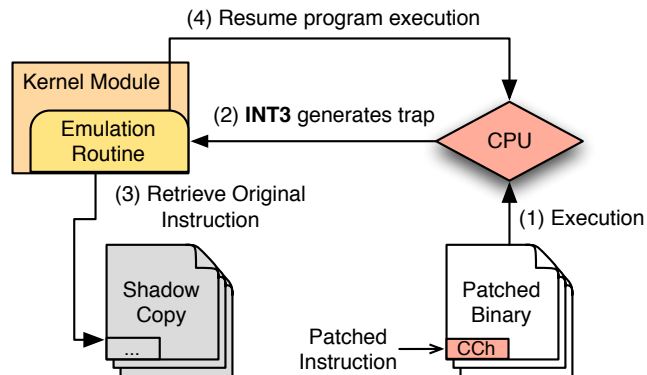


Figure 6.15: Emulation experiment using Linux kernel modules and patched executables

After testing this approach with all SPEC CPU2006 programs, we found that the average emulation time per trapped instruction was 160 processor cycles. We consider our implementation to be naive because the decoder we used could be built to handle only specific opcodes - in contrast to the full-fledged decoder ripped from `Bochs` - and for this reason we present the emulation penalty we found as an upper bound for simple x86 data processing instructions. Notice that not every instruction can be emulated in software, but it is trivial to deprecate and emulate data processing instructions.

The analytical model considers a 200 cycle penalty whereas our experiment achieves 160 cycles in average. Thus, we estimate that 40% of the x86 ISA, even after excluding multimedia extensions, could be emulated with minor performance overhead in the execution traces analyzed, that include operating systems from 1995 up to 2012.

6.7.5 Case study: AVX Re-encoding

As explained in Section 3.2, the IA-32 ISA recently had almost all new extensions focused in adding vector instructions. Moreover, as shown in Figure 3.3b and 3.4, the average operation code size of new instructions and the program size of SSE and AVX compiled programs are increasing.

We can see in Figure 6.16 that our dynamic analysis indicates that the older IA-x87 floating point extensions are still widely used by modern software, showing no signs that it can be outdated. This chart shows the percentage of instructions executed in the dynamic frequency count, starting at 93%, which belongs to specific IA-32 extensions. The data is presented from Windows 95 to Windows 7 and Slackware 3 to Ubuntu 12. For example, over 4% of total instructions executed in our Windows 7 workload were extensions instructions.

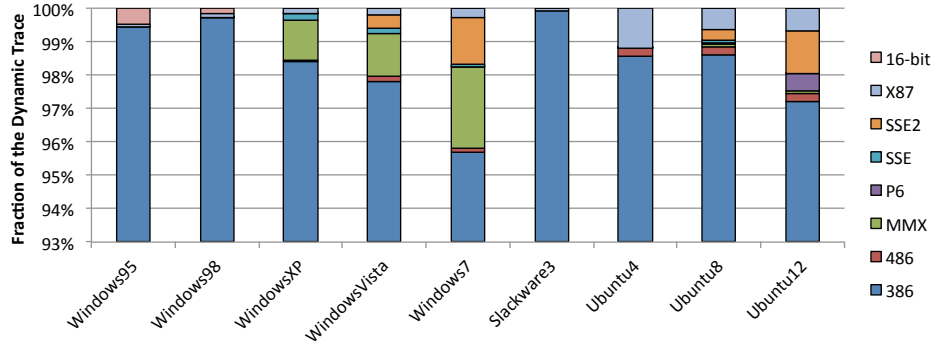


Figure 6.16: Percentage of extension instructions executed in Windows and Linux dynamic traces.

The outdateding mechanism proposed in this thesis allows the recycling mechanism to be used in outdated instructions found in our analysis to support a smooth transition to eliminate old extensions. Thus, we analyzed the improvement on total program code size and on instruction cache misses if AVX most frequent instructions had their encodings changed in favor of using formerly removed, shorter, operation codes.

We considered 4 re-encoding scenarios where the AVX statically most frequent instructions were the target for re-encoding. Each scenario has the form $AVX-(n,m)$ where n and m represent the number of re-encoded 1-byte and 2-bytes operation codes, respectively. For example, in the $AVX-(5,6)$ scenario there are 5 1-byte and 6 2-byte recycled operation codes used for AVX re-encoding.

Figure 6.17 presents the total code size of SPECfp programs in each scenario, using their old AVX version as the baseline for comparison. The chart also shows the size of a version of the program compiled using the old IA-x87 extension for floating point arith-

metic. The original AVX version of these programs is, on average, 6% bigger than their IA-x87 version in total code size. On small programs that have a higher percentage of floating point instructions, as in the `470.1bm`, the total code size can be 15% bigger. However, vectorization support is not available in the old IA-x87. The goal of this re-encoding is to offer modern vectorization support with the similar code compaction achieved with the first instructions introduced in the IA-32 ISA.

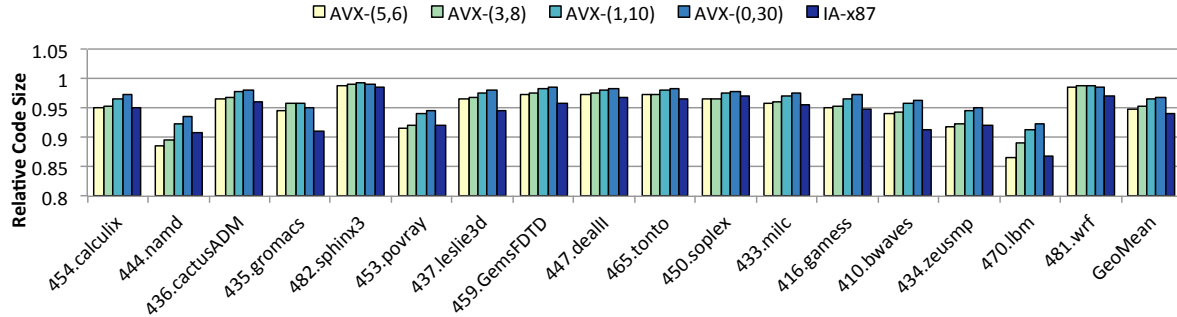


Figure 6.17: Total code size of SPEC 2006 floating point programs in different scenarios, relative to the original AVX floating point arithmetic version.

Even though AVX re-encoding may seem expensive in face of the high number of instructions in this extension, in practice, it is enough to encode 5 of the most frequent AVX instructions using 1-byte operation codes and 6 of the remaining most frequent AVX instructions using 2-byte operation codes – the AVX-(5,6) scenario – to generate SPECfp AVX executables, on average, 5.3% smaller than their old AVX versions, almost the code compaction achieved if the user gives up modern vectorization support and uses the old IA-x87 extension for floating point arithmetic.

If there is no 1-byte operation code available to encode the most frequent AVX instruction, the AVX-(0,30) scenario shows that even if there are 30 2-byte operation codes, the total code size is, on average, only 3.2% smaller than the the old AVX version.

The most important benefit of re-encoding is the program size effect on the instruction cache. Considering the AVX-(5,6) scenario, we used the dinero cache simulator [34] to measure the instruction cache miss reduction impact of re-encoded AVX programs from the SPECfp benchmark. Based on a recent *Ivy Bridge Core i7* cache configuration, we settled dinero to use a 32K instruction L1 cache, without L2, in two different configurations: 4-way and 8-way associative.

Figure 6.18 presents our results for the reduction of instruction cache misses when long AVX operation codes are replaced with shorter reused operation codes. Since these frequent instructions now use less space and the cache can hold more instructions, we can expect a cache performance increase. This effect is shown by the graph, which compares

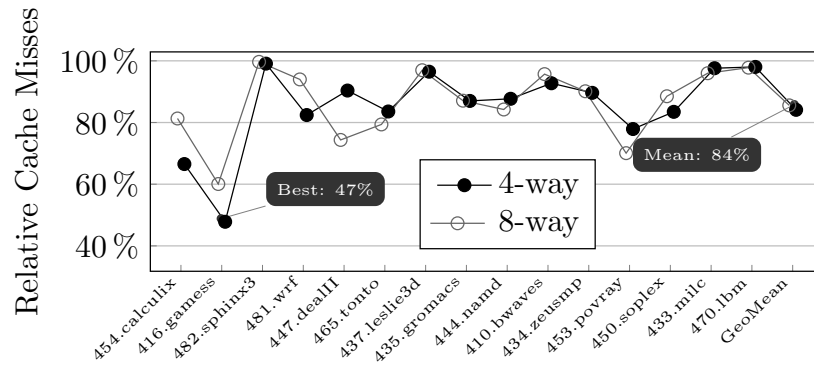


Figure 6.18: Instruction cache misses of SPEC 2006 floating point programs with shorter AVX encodings, relative to the original AVX encoding.

the cache misses of the new encoding against the original, long, encoding, whose cache misses were used as our baseline for comparison. For example, 416.gamess suffers 53% less cache misses if the X86 AVX instruction set uses a more compact way to encode its operation codes, representing our best case for cache miss reduction. In the worst case, 482.sphinx had only 0.4% less cache misses using the 8-way cache configuration. Although 470.lbm achieves a 85% compression ratio, no significant reduction in the number of cache misses occurs because the original AVX encoded version already has a very small number of cache misses.

Even though the cache misses reduction for the 8-way cache were more modest, in some special cases, the 8-way cache had even greater reduction than the 4-way cache, showing us that the cache effects, although difficult to predict, are always beneficial. On average, there was 16% less cache misses using the 4-way cache and 15% less cache misses using the 8-way cache.

6.8 Considerations

We showed in this Chapter that the x86 ISA is growing fast and reached more than 1300 different instructions in 2013; we analyzed x86 code from 5 Windows versions and 7 Linux distributions from 1995 to 2012 and show that up to 57 instructions became unused with time. We propose a mechanism to recycle instruction opcodes through legacy instruction emulation without breaking backwards software compatibility. We also include a case study of the AVX x86 SIMD instructions with shorter instruction encodings from other unused instructions to yield up to 14% code size reduction and 53% instruction cache miss reduction in SPEC CPU2006 floating-point programs. Our results show that up to 40% of the x86 instructions can be removed with less than 5% of overhead through our

technique without breaking any legacy code.

The x86 recycling mechanism is a shared work with Rafael Auler, a PhD student from University of Campinas. Rafael alone was responsible for the dynamic analysis, performance estimation and hardware considerations for the recycling mechanism. We divided the Linux kernel module work: he embedded the **Bochs** emulator and setup the trap handler while the author wrote the tool to patch the binaries, implemented the remaining part of the kernel module and run the benchmarks into a specially crafted Linux machine. We also shared ideas about the research direction and by analyzing all obtained data together. The author of the thesis solely designed the recycling mechanism logic and evaluated the AVX re-encoding scenario and instruction cache results.

Chapter 7

Conclusion

In the research described in this thesis we extensively explored compressibility properties of two CISC and RISC processors: the x86 and SPARC ISA.

We analyze some instructions from the x86 ISA family, an old but very popular CISC ISA, stopped being used with time and how this fact can be used to reorganize the operation code space to provide a simpler processor hardware that implements less instructions and offers new software an attractive code compaction rate, as good as when the ISA was first proposed, while still providing backward compatibility. Our recycling mechanism is a novel approach and in it we propose that outdated instructions be emulated in software, removing ISA design mistakes from the past and providing increased hardware efficiency for current software, allowing a smooth processor evolution.

We analyzed disk images of systems from 1995 to 2012. In Linux systems, 30 instructions disappeared from Slackware 1996 to Ubuntu 2012. For Windows, 10 instructions disappeared. In a dynamic user-oriented workload, this number raised to 45 in Linux and to 57 in Windows.

Re-encoding the AVX extension to use shorter 1-byte opcodes lead to an average 5.3% overall code compaction in SPEC CPU2006 floating point programs compiled to use AVX instructions, and reached up to 14% in `1bm`. The re-encoding also lead to a 53% instruction cache miss reduction in `416.gamess` and 16% on average. This is a similar result to the best evaluated radical approach without any backwards compatibility break. Finally, we implemented a Linux kernel driver that handles the emulation traps. After testing the emulation of popular instructions of SPEC CPU2006 programs, the performance impact experienced was, on average, 160 x86 instructions per emulated instruction. We then estimated the overhead in several dynamic traces of old and new operating systems and concluded that up to 40% of the x86 ISA could be emulated when tolerating 5% of performance overhead.

We introduced a method to analyze and detect compression opportunities in RISC

ISAs through the SPARC case study, followed by an extensive evaluation of SPARC programs from different benchmarks and the Linux Kernel. An Integer Linear Program uses extracted information from static analysis and assists in the selection of the best 16-bit instruction formats, minimizing the code size. We designed a complete 16-bit instruction set extension for the SPARC architecture: the SPARC16.

An FPGA implementation, software emulation and compiler code generator support were used to evaluate the SPARC16 extension. We achieve an average compression ratio of 72% for mediabench programs, 74% for MiBench and 72% for the SPEC2006. We showed that some target specific compiler optimizations are crucial to allow code size reduction benefits from using 16-bit instructions optimizations and can improve code compression by 20%.

We achieved SPARC16 compression ratios similar to the ones obtained by production quality MIPS and ARM 16-bit extensions, achieving better compression ratios than both in several of the programs; better than MIPS16 for half of MediaBench programs, and better than Thumb2 for half of MiBench programs. For instance, in the *adpcm* program, SPARC16 compression ratio is 15% better than Thumb2.

We evaluated SPARC16 performance by analyzing effects of code size reduction in the instruction cache. Although the execution of SPARC16 programs yields more instructions than SPARC, lower cache miss ratios are achieved by SPARC16 compiled programs; the best ratio achieves 9% reduction in *dijkstra* from MiBench.

Finally, we demonstrate how reduction in cache misses affects performance and that SPARC16 is an alternative for shipping devices with reduced cache sizes, requiring smaller caches than SPARC processors for the same set of programs, without performance degradation; a 94% reduction in the cache size in the best scenario.

7.1 Contributions

The main contributions of this thesis are:

- A method for designing 16-bit ISA extensions from 32-bit ISAs.
- A 16-bit extension for the SPARC architecture and code compression evaluations.
- Compression ratio evaluation of three distinct 16-bit extensions.
- A report of the IA-32 evolution over time showing how opcode usage of programs released from 1995 to 2012 evolved and how many instructions stopped being used in static and dynamic.

- A recycling mechanism that allows for the IA-32 ISA opcode space to be better exploited.
- An analysis of how much code compaction can be achieved in SPEC FP 2006 [47] programs if we re-encode the newer IA-32 AVX extension with smaller opcodes, currently assigned to instructions that are not used anymore, and the consequent beneficial effects on the instruction cache misses.
- A performance impact estimation of the proposed recycling mechanism when executing legacy code that uses removed instructions.

7.2 Publications

The list of publications derived from this thesis is:

- B.C. Lopes, R. Auler, E. Borin, R.J. Azevedo. ISA Anti-Aging: Recycling Old Instructions and Reducing ISA Complexity. Intel Compiler, Architecture and Tools Conference - (CATC) 2013. Haifa, Israel, Nov 19, 2013.
- B.C. Lopes, R. Auler, E. Borin, R.J. Azevedo. ISA Aging: A X86 case study. Seventh Annual Workshop on the Interaction amongst Virtualization, Operating Systems and Computer Architecture - WIVOSCA 2013. Tel Aviv, Israel, Jun 23, 2013.
- L. E. Luiz, B.C. Lopes. SPARC16: A New Compression Approach for the SPARC Architecture. Computer Architecture and High Performance Computing, Symposium on, 2009, ISSN 1550–6533.

We also submitted, but these are not yet published work:

- Patent process on *Recycling Mechanism for CISC Architectures*.
- Patent process on *SPARC16: a 16-bit ISA extension for the SPARC architecture*.
- Submitted the work entitled *Design and evaluation of compact ISA extensions* to ELSEVIER *Microprocessors and Microsystems: Embedded Hardware Design (MICPRO)* journal.
- Submitted the work entitled *The x86 Recycling Mechanism* to *The 41st International Symposium on Computer Architecture (ISCA14)* conference.

7.3 Future Work

The definition of SPARC16 has room for further research. The list of possible future direction follows:

- SPARC16 lacks a feasible FPGA hardware implementation and only very simple programs can be executed. Improving the hardware support would allow real evaluation of power consumption and performance in SPARC16.
- Apply the ILP method to other architectures.
- Evaluate floating point support into SPARC16.
- Port an operating system to run on SPARC16 and evaluate the benefits and drawbacks.
- Apply linker code compaction techniques to SPARC16 programs.
- Evaluate code size and performance improvements of using SPARC16 hidden registers as spill slots rather than traditional stack locations.
- Explore how SPARC16 register window can be used to improve code compression of SPARC16 programs.

The proposed x86 recycling mechanism is still in early stages of research. The next items to be approached can be:

- We need a CISC hardware prototype or FPGA implementation to evaluate the real impact that the recycling mechanism would have on microcode ROM space and in the processor front-end. We also need the implementation of the mechanism into a software stack: Operating System, Linker, Loader and Compiler, so the mechanism could be completely evaluated.
- Evaluate other re-codification case studies besides AVX on SPEC2006.
- Increase the static analysis with more Operating System versions and programs. The same goes for dynamic analysis, with the addition on running more programs with different applications; games, photo and video editors and mobile applications.
- Analyze security implications further.
- Evaluate a firmware (BIOS) based trap mechanism handler.

Bibliography

- [1] *VAX Architecture Handbook*. Digital Equipment Corp., 1979.
- [2] D. Keymeulen A. Stoica, S. Katkoori R. Zebulum, M. Mojarradi, and T. Daud. Adaptive and evolvable analog electronics for space applications. In *Proceedings of the 7th International Conference on Evolvable Systems: From Biology to Hardware*, ICES'07, pages 379–390, Berlin, Heidelberg, 2007. Springer-Verlag.
- [3] Altera. *Nios II Processor Reference Handbook*, 2011.
- [4] E. Andersen. uclibc, <http://www.uclibc.org>.
- [5] G. Araujo, P. Centoducatte, R. Azevedo, and R. Pannain. Expression-tree-based algorithms for code compression on embedded risc architectures. *IEEE Trans. on VLSI Systems*, 8(5):530–533, Oct 2000.
- [6] ARM. *An Introduction to Thumb*. Advanced RISC Machines Ltd., March 1995.
- [7] ARM Limited. ARMv7-M Architecture Reference Manual. July 2012.
- [8] N. Aslam, M. Milward, I. Nousias, T. Arslan, and A. Erdogan. Code compression and decompression for instruction cell based reconfigurable systems. In *IPDPS*, pages 1–7, March 2007.
- [9] R. Azevedo. *Uma Arquitetura para Execução de Código Comprimido em Sistemas Dedicados*. PhD thesis, Instituto de Computação - UNICAMP, 2002.
- [10] W. Badawy and G. A. Julien. *System-on-Chip for Real-Time Applications*. Springer, 2003.
- [11] F. Barat, R. Lauwereins, and G. Deconinck. Reconfigurable instruction set processors from a hardware/software perspective. *IEEE Trans. Softw. Eng.*, 28(9):847–862, September 2002.

- [12] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanovic. Randomized instruction set emulation. *ACM Trans. Inf. Syst. Secur.*, 8(1):3–40, February 2005.
- [13] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.
- [14] M. Benes, S. M. Nowick, and A. Wolfe. A fast asynchronous Huffman decoder for compressed-code embedded processors. In *Proceedings of the 4th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, September 1998.
- [15] M. Benes, A. Wolfe, and S. M. Nowick. A high-speed asynchronous decompression circuit for embedded processors. In *Proceedings of the 17th Conference on Advanced Research in VLSI (ARVLSI '97)*, September 1997.
- [16] L. Benini, A. Macii, and A. Nannarelli. Code compression for cache energy minimization in embedded systems. In *IEEE Proceedings on Computers and Digital Techniques*, 149(4), pages 157–163, 2002.
- [17] A. Beszèdes, R. Ferenc, T. Gyimothy, A. Dolenc, and K. Karsisto. Survey of code-size reduction methods. *ACM Comput. Surv.*, 35(3):223–267, 2003.
- [18] Á. Beszédes, R. Ferenc, T. Gergely, T. Gyimóthy, G. Lóki, and L. Vidács. Csibe benchmark: One year perspective and plans. Technical report, University of Szeged, Hungary, 2004.
- [19] E. Billo, R. Azevedo, G. Araujo, P. Centoducatte, and E. W. Netto. Design of a decompressor engine on a sparc processor. In *SBCCI '05: Proceedings of the 18th annual symposium on Integrated circuits and system design*, pages 110–114, New York, NY, USA, 2005. ACM.
- [20] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In *Proceedings of the 2013 IEEE 19th HPCA*, HPCA '13, pages 1–12, Washington, DC, USA, 2013. IEEE Computer Society.
- [21] T. Bonny and J. Henkel. Efficient code density through look-up table compression. *Design, Automation and Test in Europe Conference and Exhibition*, 0:151, 2007.
- [22] T. Bonny and J. Henkel. Instruction re-encoding facilitating dense embedded code. *Design, Automation and Test in Europe Conference and Exhibition*, 0:770–775, 2008.

- [23] E. Borin, M. Breternitz, Y. Wu, and G. Araujo. Clustering-based microcode compression. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 189–196, oct. 2006.
- [24] J. Bunda, D. Fussell, W. C. Athas, and R. Jenevein. 16-bit vs. 32-bit instructions for pipelined microprocessors. *SIGARCH Comput. Archit. News*, 21(2):237–246, 1993.
- [25] P. C. Centoducatte. *Compressão de Programas Usando árvores de Expressão*. PhD thesis, Instituto de Computação, Universidade Estadual de Campinas, 1999.
- [26] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java bytecode compression for embedded systems. Technical Report RR-3578, Inria, Institut National de Recherche en Informatique et en Automatique, 1998.
- [27] M. Collin and M. Brorsson. Two-level dictionary code compression: A new scheme to improve instruction code density of embedded applications. *IEEE/ACM International Symposium on Code Generation and Optimization*, 0:231–242, 2009.
- [28] C/MSA Microprocessor Standards Committee. *1754-1994 - IEEE Standard for a 32-bit Microprocessor Architecture*. IEEE Computer Society, 1994.
- [29] Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded risc processors. *SIGPLAN Not.*, 34(5):139–149, May 1999.
- [30] M. L. Corliss, E. C. Lewis, and A. Roth. The implementation and evaluation of dynamic code decompression using dise. *ACM Trans. Embed. Comput. Syst.*, 4(1):38–72, 2005.
- [31] IBM Microelectronics Division. *The PowerPC 405 Core*. International Business Machines (IBM) Corporation, 1998.
- [32] L. L. Ecco. Sparc16: Uma nova visão de compressão para processadores sparc. Master’s thesis, University of Campinas, 2010.
- [33] L. L. Ecco, B. C. Lopes, E. C. Xavier, R. Pannain, P. Centoducatte, and R. Azevedo. Sparc16: A new compression approach for the sparc architecture. In *21st International Symposium on Computer Architecture and High Performance Computing, 2009. SBAC-PAD ’09.*, pages 169–176, 2009.
- [34] J. Edler and M.D. Hill. Dinero iv trace-driven uniprocessor cache simulator, 2003.
- [35] J. Ernst, C. W. Fraser, W. Evans, S. Lucco, and T. A. Proebsting. Code compression. pages 358–365, June 1997.

- [36] A. Fog. *Instructions for objconv*, 2011. Version 2.11.
- [37] C. W. Fraser and T. A. Proebsting. Custom instruction sets for code compression. *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, 1995.
- [38] J. Gaisler. The leon3 processor. online, 2008. <http://www.gaisler.com>.
- [39] C. Galuzzi and K. Bertels. The Instruction-Set Extension Problem: A Survey. *ACM Trans. Reconfigurable Technol. Syst.*, 4(2):18:1–18:28, May 2011.
- [40] M. Game and A. Booker. *CodePack: Code Compression for PowerPC Processors*. International Business Machines (IBM) Corporation, 1998.
- [41] S. K. Gaurav, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS '03*, pages 272–280, 2003.
- [42] GNU. Gnu binutils, <http://www.gnu.org/software/binutils>.
- [43] M.R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. *IEEE International Workshop on Workload Characterization, WWC-4, 2001*, pages 3–14, Dec. 2001.
- [44] S. I. Haider and L. Nazhandali. A hybrid code compression technique using bitmask and prefix encoding with enhanced dictionary selection. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 58–62, New York, NY, USA, 2007. ACM.
- [45] T. R. Halfhill. Intel's Tiny Atom: New Low-Power Microarchitecture Rejuvenates the Embedded x86. Microprocessor Report, 2008.
- [46] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2 edition, 1990.
- [47] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.
- [48] D. A. Huffman. A method for construction of minimum redundancy codes. In *Proceedings of the IRE*, volume 40, pages 1098–1101, 1952.

- [49] R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani. Mao – an extensible micro-architectural optimizer. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.
- [50] IBM. *CodePack: PowerPC Code Compression Utility User's Manual. Version 3.0*. International Business Machines (IBM) Corporation, 1998.
- [51] Intel. *Galileo Datasheet*, 2013.
- [52] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, Volume 2: Instruction Set Reference edition.
- [53] Sparc International. *SYSTEM V APPLICATION BINARY INTERFACE, SPARC Processor Supplement*. 1996.
- [54] L. Jowiak, N. Nedjah, and M. Figueroa. Modern development methods and tools for embedded reconfigurable systems: A survey. *Integr. VLSI J.*, 43(1):1–33, January 2010.
- [55] T.M. Kemp, R.M. Montoye, J.D. Harper, J.D. Palmer, and D.J. Auerbach. A de-compression core for powerpc. *IBM Journal of Research and Development*, 42(6), Nov 1998.
- [56] D. Kirovski, J. Kin, and W. H. Mangione-Smith. Procedure based program compression. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 204–213, Washington, DC, USA, 1997. IEEE Computer Society.
- [57] K. Kissell. *MIPS16: High-density MIPS for the Embedded Market*. Silicon Graphics MIPS Group, 1997.
- [58] T. J. K. E. Koch, I. Boehm, and B. Franke. Integrated instruction selection and register allocation for compact code generation exploiting freeform mixing of 16 and 32-bit instructions. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 180–189. ACM, 2010.
- [59] Franck Koebel and Jean-Francois Coldefy. Scoc3: A space computer on a chip: An example of successful development of a highly integrated innovative asic. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1345–1348. European Design and Automation Association, 2010.

- [60] A. Krishnaswamy and R. Gupta. Profile guided selection of arm and thumb instructions. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, LCTES/S-COPES '02, pages 56–64, New York, NY, USA, 2002. ACM.
- [61] A. Krishnaswamy and R. Gupta. Dynamic coalescing for 16-bit instructions. *ACM Trans. Embed. Comput. Syst.*, 4(1):3–37, 2005.
- [62] R. Kumar and D. Das. Code compression for performance enhancement of variable-length embedded processors. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–36, 2008.
- [63] R. Kumar, A. Gupta, B. S. Pankaj, M. Ghosh, and P. P. Chakrabarti. Post-compilation optimization for multiple gains with pattern matching. *SIGPLAN Not.*, 40(12):14–23, 2005.
- [64] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.
- [65] K. P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux J.*, 1996(29es), September 1996.
- [66] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.
- [67] C. Lefurgy. *Efficient Execution of Compressed Programs*. PhD thesis, University of Michigan, June 2000.
- [68] C. Lefurgy, P. Bird, I. Chen, and T. Mudge. Improving code density using compression techniques. Technical Report CSE-TR-342-97, EECS Department, University of Michigan, 1997.
- [69] H. Lekatsas, J. Henkel, and V. Jakkula. Design of an one-cycle decompression hardware for performance increase in embedded systems. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 34–39, New York, NY, USA, 2002. ACM.
- [70] H. Lekatsas, J. Henkel, and W. Wolf. Code compression for low power embedded system design. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 294–299, New York, NY, USA, 2000. ACM.

- [71] Haris Lekatsas, Jörg Henkel, and Wayne Wolf. Design and simulation of a pipelined decompression architecture for embedded systems. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, pages 63–68, New York, NY, USA, 2001. ACM.
- [72] S. Y. Liao. *Code generation and optimization for embedded digital signal processors*. PhD thesis, 1996. Supervisor-Srinivas Devadas.
- [73] S. Y. Liao, S. Devadas, and K. Keutzer. Code density optimization for embedded dsp processors using data compression techniques. In *ARVLSI '95: Proceedings of the 16th Conference on Advanced Research in VLSI (ARVLSI'95)*, page 272, Washington, DC, USA, 1995. IEEE Computer Society.
- [74] B. Enoksson M. Ramström, J. Höglund and R. Svenningsson. *Final Report - 32-bit Microprocessor and Computer Development Programme*. Saab Ericsson Space AB, 1997.
- [75] R. Marks, F. Araujo, R. Santos, F. Yonehara, and R. Santos. Design and implementation of the pbiw instruction decoder in a softcore embedded processor. *2012 13th Symposium on Computer Systems*, 0:110–117, 2012.
- [76] Inc. MIPS Technologies. *MIPS32 Architecture For Programmer, Volume II: The MIPS32 Instruction Set*. MIPS Technologies, 2001.
- [77] Inc. MIPS Technologies. micromips instruction set architecture, uncompromised performance, minimum system cost. Technical report, MIPS Technologies, Inc., MIPS Technologies, Inc. 955 East Arques Avenue Sunnyvale, CA 94085 (408) 530-5000, 2009.
- [78] E. W. Netto, R. Azevedo, P. Centoducatte, and G. Araujo. Multi-profile based code compression. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 244–249, New York, NY, USA, 2004. ACM.
- [79] R. Pannain. *Compressão de Código de Programa Usando Fatoração de Operandos*. PhD thesis, Faculdade de Engenharia Elétrica e Computação, Universidade Estadual de Campinas, 1999.
- [80] D. Patterson and D. Sequin. RISC I: A Reduced Instruction Set VLSI Computer. In *Proceedings of the 8th Annual International Symposium on Computer Architecture, ISCA '81*. IEEE Computer Society Press, 1981.
- [81] D. A. Patterson and J. L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

- [82] X. Qin and P. Mishra. Efficient placement of compressed code for parallel decompression. *International Conference on VLSI Design*, 0:335–340, 2009.
- [83] Jan S. Rellermeyer, Seong-Won Lee, and Michael Kistler. Cloud platforms and embedded computing: The operating systems of the future. In *Proceedings of the 50th Annual Design Automation Conference, DAC '13*, pages 75:1–75:6, New York, NY, USA, 2013. ACM.
- [84] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. Archc: a systemc-based architecture description language. pages 66–73, Oct. 2004.
- [85] S. Seong and P. Mishra. An efficient code compression technique using application-aware bitmask and dictionary selection methods. *Design, Automation and Test in Europe Conference and Exhibition*, 0:112, 2007.
- [86] Shibu. *Introduction To Embedded Systems*. Tata McGraw-Hill Education, 2009.
- [87] Inc. SPARC International. *The SPARC architecture manual: version 8*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.
- [88] R. Stallman. *Using GCC: the GNU compiler collection reference manual*. Free Software Foundation, 2003.
- [89] B. D. Sutter, B. D. Bus, and K. D. Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Trans. Program. Lang. Syst.*, 27(5):882–945, 2005.
- [90] B. D. Sutter, L. V. Put, D. Chanet, B. D. Bus, and K. D. Bosschere. Link-time compaction and optimization of arm executables. *ACM Trans. Embed. Comput. Syst.*, 6(1):5, 2007.
- [91] A. S. Tanenbaum. *Structured Computer Organization*. Prentice Hall, 5th edition edition, 2005.
- [92] W. T. Wilner. Burroughs b1700 memory utilization. In *AFIPS '72 (Fall, part I): Proceedings of the December 5-7, 1972, fall joint computer conference, part I*, pages 579–586, New York, NY, USA, 1972. ACM.
- [93] A. Wolfe and A. Chanin. Executing compressed programs on an embedded risc architecture. *SIGMICRO Newsl.*, 23(1-2):81–91, 1992.
- [94] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. 1992.

- [95] L. Xianhua, Z. Jiyu, and C. Xu. Efficient code size reduction without performance loss. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 666–672, New York, NY, USA, 2007. ACM.
- [96] X. H. Xu, C. T. Clarke, and S. R. Jones. High performance code compression architecture for the embedded arm/thumb processor. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, pages 451–456, New York, NY, USA, 2004. ACM.

Appendix A

Static Analysis

A.1 Instruction Usage By Group

Usage	Group	Instructions			
22.42%	alu_arith	(29.59%)add_imm (28.49%)subcc_imm (15.93%)add_reg (12.28%)subcc_reg (05.68%)sub_reg (02.37%)smul_reg	(01.45%)addcc_imm (01.28%)subx_imm (01.22%)addx_imm (0.33%)umul_reg (0.32%)smul_imm (0.28%)udiv_reg	(0.20%)subx_reg (0.17%)addcc_reg (0.16%)sdiv_reg (0.14%)addx_reg (0.04%)udiv_imm (0.04%)sub_imm	(0.02%)smulcc_reg (0.01%)udivcc_reg (0.00%)sdivcc_reg
15.00%	load	(61.16%)ld_imm (16.69%)ld_reg (05.59%)ldub_reg	(04.46%)lduh_imm (03.54%)ldd_imm (01.85%)ldub_imm	(01.71%)ldsb_reg (01.70%)lduh_reg (01.41%)ldsh_imm	(0.71%)ldsb_imm (0.65%)ldsh_reg (0.54%)ldd_reg
12.14%	move	(71.93%)or_reg	(28.07%)or_imm		
11.76%	branch	(32.66%)ba (22.71%)be (13.58%)bne (07.86%)ble	(06.06%)bl (04.79%)bleu (04.27%)bg (02.91%)bgu	(02.76%)bge (0.80%)bcs (0.67%)bcc (0.58%)bpos	(0.35%)bneg
08.69%	store	(64.02%)st_imm (15.20%)st_reg	(06.62%)stb_reg (05.06%)sth_imm	(03.14%)stb_imm (03.02%)std_imm	(02.52%)sth_reg (0.42%)std_reg
08.68%	alu_logic	(31.58%)or_imm (29.42%)or_reg (10.92%)and_imm (09.04%)andcc_imm	(04.51%)xor_reg (03.10%)andcc_reg (02.67%)and_reg (02.65%)orcc_reg	(02.51%)xor_imm (01.56%)orcc_imm (01.18%)andn_reg (0.72%)xnor_reg	(0.15%)andncc_reg
04.77%	call	(87.38%)call	(12.62%)jmpl_reg		
04.36%	sethi	(10.00%)sethi			
04.33%	alu_shift	(41.15%)sll_imm (25.29%)sra_imm	(25.03%)srl_imm (04.81%)sll_reg	(02.76%)srl_reg (0.95%)sra_reg	
02.06%	branch annul	(35.45%)be (26.54%)bne (08.54%)ba (04.92%)bg	(04.42%)bl (04.30%)ble (04.15%)bgu (03.74%)bge	(03.67%)bleu (01.67%)bcs (01.18%)bcc (0.75%)bneg	(0.68%)bpos
01.64%	registerwindow	(49.17%)save_imm	(48.23%)restore_reg	(02.36%)restore_imm	(0.24%)save_reg
01.50%	nop	(10.00%)nop			
01.47%	unimp	(10.00%)unimp			
0.75%	ret from sub	(10.00%)jmpl_imm			
0.17%	os	(67.96%)wr_reg	(32.04%)rd_reg		
0.16%	ret from leaf sub	(10.00%)jmpl_imm			
0.09%	jump	(70.65%)jmpl_imm	(29.35%)jmpl_reg		

Table A.1: Instruction usage by groups – mediabench

Usage	Group	Instructions			
22.63%	alu_arith	(31.66%)subcc_imm (30.29%)add_imm (12.38%)subcc_reg (11.07%)add_reg (06.93%)sub_reg	(01.91%)subx_imm (01.39%)ad- dcc_imm (01.32%)addx_imm (0.98%)smul_reg (0.54%)umul_reg	(0.51%)udiv_reg (0.35%)subx_reg (0.27%)addcc_reg (0.24%)addx_reg (0.07%)udiv_imm	(0.03%)sub_imm (0.03%)smulcc_reg (0.03%)smul_imm
15.00%	move	(80.84%)or_reg	(19.16%)or_imm		
13.42%	branch	(32.11%)ba (22.54%)be (15.41%)bne (08.29%)ble	(05.30%)bleu (05.07%)bg (04.39%)bl (03.36%)bgu	(01.71%)bge (0.80%)bcs (0.57%)bcc (0.28%)bneg	(0.17%)bpos
11.50%	load	(57.69%)ld_imm (13.76%)ld_reg (09.34%)ldd_imm	(05.25%)ldub_reg (05.12%)lduh_imm (04.65%)ldsb_reg	(01.46%)ldd_reg (01.20%)ldsb_imm (01.00%)ldsh_imm	(0.40%)ldub_imm (0.13%)ldsh_reg
09.06%	alu_logic	(29.44%)or_reg (23.62%)or_imm (15.69%)andcc_imm (09.83%)and_imm	(04.51%)and_reg (04.47%)orcc_reg (04.09%)andcc_reg (03.21%)xor_imm	(01.90%)andn_reg (01.69%)orcc_imm (01.10%)xor_reg (0.30%)xnor_reg	(0.17%)andncc_reg
07.94%	store	(60.07%)st_imm (19.32%)st_reg	(06.74%)std_imm (06.17%)stb_reg	(04.34%)sth_imm (02.02%)stb_imm	(0.87%)sth_reg (0.48%)std_reg
04.64%	call	(93.25%)call	(06.75%)jmpl_reg		
03.25%	sethi	(10.00%)sethi			
03.17%	unimp	(10.00%)unimp			
02.86%	alu_shift	(36.14%)srl_imm (32.66%)sll_imm	(15.26%)sra_imm (08.43%)sll_reg	(06.29%)srl_reg (01.20%)sra_reg	
02.15%	branch annul	(43.59%)be (24.73%)bne (05.87%)ba (05.34%)bgu	(04.98%)bleu (03.91%)bg (02.85%)bge (02.67%)bl	(02.14%)ble (01.42%)bcs (01.42%)bcc (0.71%)bpos	(0.36%)bneg
01.56%	registerwindow	(49.39%)restore_reg	(48.65%)save_imm	(01.47%)restore_imm	(0.49%)save_reg
01.56%	nop	(10.00%)nop			
0.72%	ret from sub	(10.00%)jmpl_imm			
0.21%	os	(62.96%)wr_reg	(37.04%)rd_reg		
0.19%	ret from leaf sub	(10.00%)jmpl_imm			
0.15%	jump	(65.00%)jmpl_imm	(35.00%)jmpl_reg		

Table A.2: Instruction usage by groups – MiBench

Usage	Group	Instructions			
17.21%	alu_arith	(33.87%)subcc_imm (25.75%)add_imm (18.27%)subcc_reg (09.67%)add_reg (04.45%)sub_reg (01.76%)addx_imm	(01.59%)subx_imm (01.28%)mulsc_reg (01.13%)addcc_imm (0.66%)addcc_reg (0.63%)addxc_reg (0.44%)addx_reg	(0.28%)sub_imm (0.14%)subx_reg (0.04%)mulsc_imm (0.02%)subxc_reg (0.00%)umul_reg (0.00%)udivcc_imm	(0.00%)udiv_reg (0.00%)tad- dcctv_imm (0.00%)smul_reg (0.00%)sdivcc_reg (0.00%)addxc_imm
16.08%	load	(73.05%)ld_imm (11.75%)ld_reg (05.22%)ldub_imm (04.48%)lduh_imm	(02.48%)ldd_imm (01.12%)ldub_reg (0.54%)lduh_reg (0.52%)ldsb_imm	(0.38%)ldsb_reg (0.28%)ldd_reg (0.17%)ldsh_imm (0.01%)ldsh_reg	(0.00%)ldstub_imm
13.15%	move	(66.27%)or_reg	(33.73%)or_imm		
10.89%	branch	(32.64%)be (28.60%)ba (21.24%)bne (02.72%)bl	(02.72%)bgu (02.43%)ble (02.22%)bcs (02.12%)bleu	(01.76%)bcc (01.65%)bg (01.11%)bge (0.40%)bneg	(0.37%)bpos
10.24%	alu_logic	(30.07%)or_imm (28.29%)or_reg (09.72%)andcc_imm (08.53%)and_imm (06.34%)and_reg (05.47%)orcc_imm	(03.71%)andcc_reg (02.15%)orcc_reg (02.07%)xor_reg (01.36%)andn_reg (01.31%)xor_imm (0.52%)xnor_reg	(0.29%)andncc_reg (0.05%)andn_imm (0.04%)xnor_imm (0.03%)andncc_imm (0.02%)orn_reg (0.01%)xorcc_reg	(0.01%)xnorcc_reg (0.00%)xorcc_imm (0.00%)orncc_reg
07.47%	store	(70.38%)st_imm (10.62%)st_reg	(06.21%)stb_imm (06.16%)std_imm	(04.41%)sth_imm (01.22%)stb_reg	(0.56%)sth_reg (0.44%)std_reg
06.85%	call	(94.37%)call	(05.55%)jmpl_reg	(0.09%)jmpl_imm	
04.65%	sethi	(10.00%)sethi			
03.07%	nop	(10.00%)nop			
02.88%	alu_shift	(51.45%)sll_imm (32.53%)srl_imm	(08.82%)sra_imm (04.14%)sll_reg	(02.23%)srl_reg (0.83%)sra_reg	
02.81%	branch annul	(32.16%)be (30.20%)bne (18.86%)ba (04.11%)bgu	(02.76%)bcs (02.45%)bg (02.20%)bleu (01.89%)bcc	(01.86%)bl (01.29%)ble (01.13%)bge (0.61%)bneg	(0.47%)bpos (0.02%)bvs
02.60%	registerwindow	(48.34%)restore_reg	(46.70%)save_imm	(04.82%)restore_imm	(0.14%)save_reg
01.04%	ret from sub	(10.00%)jmpl_imm			
0.36%	os	(36.47%)rd_reg (19.57%)tne_imm (17.67%)te_imm (09.82%)ta_imm (06.31%)wr_reg	(02.02%)tcs_imm (01.96%)wr_imm (01.25%)tle_imm (01.07%)tgu_imm (01.07%)tcc_imm	(0.65%)rett_imm (0.54%)tl_imm (0.48%)rett (0.36%)tge_imm (0.36%)flush_reg	(0.18%)flush_imm (0.12%)tleu_imm (0.12%)tg_imm
0.31%	ret from leaf sub	(10.00%)jmpl_imm			
0.23%	unimp	(10.00%)unimp			
0.12%	alternate	(24.74%)sta (23.68%)sta (18.25%)lda	(14.56%)lda (09.47%)stba (04.74%)lduba	(02.11%)stda (01.05%)stha (01.05%)lduha	(0.18%)ldsha (0.18%)ldda
0.03%	jump	(82.05%)jmpl_reg	(17.95%)jmpl_imm		

Table A.3: Instruction usage by groups – Linux Kernel

Appendix B

SPARC16 ISA

B.1 List of SPARC16 Instructions

B.1.1 ADDCCri, ADDCCri_ext, ADDCCrr, ADDri, ADDri_ext, ADDrr

ADDCCri, Format: RRI, Assembly syntax: `add16 $rs, $imm, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	imm					rs			rd		

ADDCCri_ext, Format: EXTEND RRI, Assembly syntax: `eadd16 $rs, $imm, $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	imm								0	0	0	1	0	imm				rs		rd				

ADDCCrr, Format: RRR, Assembly syntax: `add16 $rs1, $rs2, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	0	0	rs2			rs1			rd		

ADDri, Format: RRI, Assembly syntax: `add16 $rs, $imm, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	imm					rs			rd		

ADDri_ext, Format: EXTEND RRI, Assembly syntax: `eadd16 $rs, $imm, $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	imm								0	0	0	1	0	imm				rs		rd				

ADDrr, Format: RRR, Assembly syntax: `add16 $rs1, $rs2, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	0	0	rs2			rs1			rd		

B.1.2 ADDFP, ADDFP_ext

ADDFP, Format: RI2, Assembly syntax: `addfp $addr, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	0	0	addr					rd		

ADDFP_ext, Format: EXTEND RI2, Assembly syntax: `eaddfp $addr, $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	addr								1	0	0	1	1	0	0	0	addr				rd			

B.1.3 ADDSP, ADDSP_ext

ADDSP, Format: RI2, Assembly syntax: `addsp $addr, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	1	addr					rd		

ADDSP_ext, Format: EXTEND RI2, Assembly syntax: `eaddsp $addr, $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	addr								1	0	0	1	1	0	1	1	addr				rd			

B.1.4 ADDXri, ADDXri_ext, ADDXrr

ADDXri, Format: RRI2, Assembly syntax: `addx16 $rs, $imm, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	0	0	0	imm			rs			rd	

ADDXri_ext, Format: EXTEND RRI2, Assembly syntax: `eaddx16 $rs, $imm, $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	imm											1	1	0	0	1	0	0	0	imm	rs			rd			

ADDXrr, Format: RR, Assembly syntax: `addx16 $rs, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	1	0	0	rs				rd	

B.1.5 ANDri, ANDri_ext, ANDrr

ANDri, Format: RRI, Assembly syntax: `and16 $rs, $imm, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	imm					rs			rd		

ANDri_ext, Format: EXTEND RRI, Assembly syntax: `eand16 $rs, $imm, $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	imm								0	0	0	1	1	imm				rs			rd			

ANDrr, Format: RRR, Assembly syntax: `and16 $rs1, $rs2, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	1	0	rs2			rs1			rd		

B.1.6 ANDNrr

Format: RR, Assembly syntax: `andn16 $rs, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	1	1	0	1	rs				rd	

B.1.10 BNE, BNE_ext

BNE, Format: I, Assembly syntax: `bne16 $brtarget`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	brtarget									

BNE_ext, Format: EXTEND I, Assembly syntax: `ebne16 $brtarget`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	brtarget											0	1	0	0	1	brtarget										

B.1.11 CALL, CALL_ext

CALL, Format: I, Assembly syntax: `call16 $calltarget`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	calltarget										

CALL_ext, Format: EXTEND I, Assembly syntax: `ecall16 $calltarget`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	calltarget											0	0	0	0	0	calltarget										

B.1.12 CALLR

Format: RR, Assembly syntax: `callr $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	1	1	1	0	0	0	rd		

B.1.13 CALLRX

Format: RR, Assembly syntax: `callrx $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	1	1	1	1	1	1	rd		

B.1.14 CALLX, CALLX_ext

CALLX, Format: I, Assembly syntax: `callx $calltarget`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	calltarget										

CALLX_ext, Format: EXTEND I, Assembly syntax: `ecallx $calltarget`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	calltarget										1	0	0	0	0	calltarget											

B.1.15 CMPri, CMPri_ext, CMPrr

CMPri, Format: RI, Assembly syntax: `cmp16 $rd, $imm`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	imm							rd			

CMPri_ext, Format: EXTEND RI, Assembly syntax: `ecmp16 $rd, $imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	0	0	0	imm				0	1	1	0	1	imm						rd					

CMPrr, Format: RR, Assembly syntax: `cmp16 $rs, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	1	0	0	rs			rd		

B.1.16 JMPR

Format: RR, Assembly syntax: `jmp $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	1	1	1	0	0	1	rd		

B.1.17 JMPRX

Format: RR, Assembly syntax: `jmprx $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	1	1	1	1	1	0	rd		

B.1.18 LDri, LDri_ext, LDrr

LDri, Format: RRI, Assembly syntax: `ld16 [$addr], $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	addr								rd		

LDri_ext, Format: EXTEND RRI, Assembly syntax: `eld16 [$addr], $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	addr								1	1	0	0	0	addr								rd		

LDrr, Format: RRR, Assembly syntax: `ld16 [$addr], $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	addr						rd		

B.1.19 LDFP, LDFP_ext

LDFP, Format: RI2, Assembly syntax: `ldfp [$imm], $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	0	imm						rd	

LDFP_ext, Format: EXTEND RI2, Assembly syntax: `eldfp [$imm], $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	0	1	1	0	0	0	imm								1	0	0	1	1	0	1	0	imm								rd	

B.1.20 LDSBri, LDSBri_ext, LDSBrr

LDSBri, Format: LDST, Assembly syntax: `ldsb16 [$addr], $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	addr							rd		

LDSBri_ext, Format: EXTEND LDST, Assembly syntax: `eldsb16 [$addr], $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	addr									1	0	1	0	0	0	addr							rd		

LDSBrr, Format: EXTEND RR, Assembly syntax: `eldsb16 [$addr], $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	0	0	0	0	0	addr			0	1	0	1	0	0	1	0	0	1	addr			rd		

B.1.21 LDShri, LDShri_ext, LDShrr

LDShri, Format: LDST, Assembly syntax: `ldsh16 [$addr], $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	addr							rd		

LDShri_ext, Format: EXTEND LDST, Assembly syntax: `eldsh16 [$addr], $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	addr									0	0	1	0	0	0	addr							rd		

LDShrr, Format: EXTEND RR, Assembly syntax: `eldsh16 [$addr], $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	0	0	0	0	0	addr			0	1	0	1	0	0	1	0	1	0	addr			rd		

B.1.22 LDSP, LDSP_ext

LDSP, Format: RI2, Assembly syntax: `ldsp [$imm], $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	1	0	1	imm						rd	

LDSP_ext, Format: EXTEND RI2, Assembly syntax: `eldsp [$imm], $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	1	1	0	0	0	imm								1	0	0	1	1	1	0	1	imm								rd

B.1.23 LDUBri, LDUBri_ext, LDUBrr

LDUBri, Format: LDST, Assembly syntax: `ldub16 [$addr], $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	addr								rd	

LDUBri_ext, Format: EXTEND LDST, Assembly syntax: `eldub16 [$addr], $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	addr								1	0	1	0	0	1	addr								rd		

LDUBrr, Format: EXTEND RR, Assembly syntax: `eldub16 [$addr], $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	0	0	0	0	0	addr			0	1	0	1	0	0	0	0	0	1	addr			rd		

B.1.24 LDUHri, LDUHri_ext, LDUHrr

LDUHri, Format: LDST, Assembly syntax: `lduh16 [$addr], $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	addr								rd	

LDUHri_ext, Format: EXTEND LDST, Assembly syntax: `elduh16 [$addr], $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	addr								0	0	1	0	0	1	addr								rd		

LDUHrr, Format: EXTEND RR, Assembly syntax: `elduh16 [$addr], $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	0	0	0	0	0	addr			0	1	0	1	0	0	0	0	1	0	addr			rd		

B.1.25 MOV, MOV_ext

MOV, Format: RI, Assembly syntax: `mov16 $imm, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	imm								rd		

MOV_ext, Format: EXTEND RI, Assembly syntax: `emov16 $imm, $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	0	0	0	imm					0	1	1	1	0	imm						rd				

B.1.26 MOV8to32, MOVrr

MOV8to32, Format: I2, Assembly syntax: `movra $reg8, $reg32`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	reg32						reg8	

MOVrr, Format: I2, Assembly syntax: `movra $reg8, $reg32`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	reg32						reg8	

B.1.27 MOV32to8

Format: I2, Assembly syntax: `movrb $reg32, $reg8`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	reg32						reg8	

B.1.28 NEGrr

Format: RR, Assembly syntax: `neg16 $rs, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	1	0	1	rs			rd		

B.1.29 NOPFormat: RR, Assembly syntax: `nop`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	1	1	1	1	0	1	0	0	0

B.1.30 ORri, ORri_ext, ORrr*ORri*, Format: RRI2, Assembly syntax: `or16 $rs, $imm, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	imm		rs		rd			

ORri_ext, Format: EXTEND RRI2, Assembly syntax: `eor16 $rs, $imm, $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	imm											1	1	0	1	0	1	0	0	imm		rs		rd			

ORrr, Format: RRR, Assembly syntax: `or16 $rs1, $rs2, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	1	1	rs2		rs1		rd				

B.1.31 ORNri, ORNri_ext, ORNrr*ORNri*, Format: RRI2, Assembly syntax: `orn16 $rs, $imm, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	imm		rs		rd			

ORNri_ext, Format: EXTEND RRI2, Assembly syntax: `eorn16 $rs, $imm, $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	imm											1	1	0	1	0	1	1	0	imm		rs		rd			

ORNrr, Format: RR, Assembly syntax: `orn16 $rs, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	1	0	0	rs		rd			

B.1.32 RDY

Format: RR, Assembly syntax: `rd16 %y, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	1	1	1	1	0	0	0	rd		

B.1.33 RESTORErr, RESTORErr_ext

RESTORErr, Format: RR, Assembly syntax: `restore16 $rs, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	0	0	0	rs			rd		

RESTORErr_ext, Format: EXTEND RR, Assembly syntax: `erestore16 $rs, $rsext, $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	0	0	0	0	0	rsext			0	1	0	1	0	0	0	0	0	0	rs			rd		

B.1.34 RET

Format: RR, Assembly syntax: `ret`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	1	1	1	0	1	0	0	0	0

B.1.35 RETL

Format: RR, Assembly syntax: `retl`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	1	1	1	0	1	1	0	0	0

B.1.36 SAVEri, SAVEri_ext

SAVEri, Format: I2, Assembly syntax: `savesp $imm`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	imm								

SAVERi_ext, Format: EXTEND I2, Assembly syntax: **esavesp \$imm**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	0	0	0	0	imm				1	1	1	0	0	0	1	imm								

B.1.37 SDIVri, SDIVri_ext, SDIVrr

SDIVri, Format: RRI2, Assembly syntax: **sdiv16 \$rs, \$imm, \$rd**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	1	imm		rs			rd		

SDIVri_ext, Format: EXTEND RRI2, Assembly syntax: **esdiv16 \$rs, \$imm, \$rd**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	imm										1	1	0	1	0	0	0	1	imm		rs		rd				

SDIVrr, Format: RR, Assembly syntax: **sdiv16 \$rs, \$rd**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	1	1	1	rs			rd		

B.1.38 SETHi

Format: EXTEND Sethi, Assembly syntax: **sethi32 \$imm, \$rd**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	imm																rd								

B.1.39 SLLri, SLLrr

SLLri, Format: RRI, Assembly syntax: **sll16 \$rs, \$imm, \$rd**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	imm						rs		rd		

SLLrr, Format: RR, Assembly syntax: `sll16 $rs, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	1	0	1	rs			rd		

B.1.40 SMULri, SMULri_ext, SMULrr

SMULri, Format: RRI2, Assembly syntax: `smul16 $rs, $imm, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	imm			rs			rd	

SMULri_ext, Format: EXTEND RRI2, Assembly syntax: `esmul16 $rs, $imm, $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	imm											1	1	0	1	0	0	0	0	imm		rs		rd			

SMULrr, Format: RR, Assembly syntax: `smul16 $rs, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	0	1	1	rs			rd		

B.1.41 SRARI, SRARR

SRARI, Format: RRI, Assembly syntax: `sra16 $rs, $imm, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	imm					rs			rd		

SRARR, Format: RR, Assembly syntax: `sra16 $rs, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	1	1	1	rs			rd		

B.1.42 SRLri, SRLrr

SRLri, Format: RRI, Assembly syntax: `srl16 $rs, $imm, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	imm					rs			rd		

SRLrr, Format: RR, Assembly syntax: `srl16 $rs, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	1	1	0	rs			rd		

B.1.43 STri, STri_ext, STrr

STri, Format: RRI, Assembly syntax: `st16 $rd, [$addr]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	addr							rd			

STri_ext, Format: EXTEND RRI, Assembly syntax: `est16 $rd, [$addr]`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	addr								0	1	1	0	0	addr						rd				

STrr, Format: RRR, Assembly syntax: `st16 $rd, [$addr]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	0	addr						rd		

B.1.44 STBri, STBri_ext, STBrr

STBri, Format: LDST, Assembly syntax: `stb16 $rd, [$addr]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	addr							rd		

STBri_ext, Format: EXTEND LDST, Assembly syntax: `estb16 $rd, [$addr]`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	addr									0	1	1	1	1	0	addr						rd			

STBrr, Format: EXTEND RR, Assembly syntax: `estb16 $rd, [$addr]`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	0	0	0	0	0	addr			0	1	0	1	0	0	0	1	0	1	addr			rd		

B.1.45 STFP, STFP_ext

STFP, Format: RI2, Assembly syntax: `stfp $rd, [$imm]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	0	1	imm					rd		

STFP_ext, Format: EXTEND RI2, Assembly syntax: `estfp $rd, [$imm]`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	imm								1	0	0	1	1	0	0	1	imm						rd	

B.1.46 STHri, STHri_ext, STHrr

STHri, Format: LDST, Assembly syntax: `sth16 $rd, [$addr]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	addr							rd		

STHri_ext, Format: EXTEND LDST, Assembly syntax: `esth16 $rd, [$addr]`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	addr									0	1	1	1	1	1	addr						rd			

STHrr, Format: EXTEND RR, Assembly syntax: `esth16 $rd, [$addr]`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	0	0	0	0	0	addr			0	1	0	1	0	0	0	1	1	0	addr			rd		

B.1.47 STSP, STSP_ext

STSP, Format: RI2, Assembly syntax: `stsp $rd, [$imm]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	1	0	0	imm					rd		

STSP_ext, Format: EXTEND RI2, Assembly syntax: `estsp $rd, [$imm]`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	imm								1	0	0	1	1	1	0	0	imm					rd		

B.1.48 SUBrr

Format: RRR, Assembly syntax: `sub16 $rs1, $rs2, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	0	1	rs2			rs1			rd		

B.1.49 SUBXri, SUBXri_ext, SUBXrr

SUBXri, Format: RRI2, Assembly syntax: `subx16 $rs, $imm, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	0	0	1	imm			rs			rd	

SUBXri_ext, Format: EXTEND RRI2, Assembly syntax: `esubx16 $rs, $imm, $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	imm											1	1	0	0	1	0	0	1	imm		rs		rd			

SUBXrr, Format: RR, Assembly syntax: `subx16 $rs, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	1	1	0	0	rs			rd		

B.1.50 tRESTORE

Format: RR, Assembly syntax: `trestore`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	1	1	1	1	0	0	0	0	0

B.1.51 UDIVri, UDIVri_ext, UDIVrr

UDIVri, Format: RRI2, Assembly syntax: `udiv16 $rs, $imm, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	1	imm		rs			rd		

UDIVri_ext, Format: EXTEND RRI2, Assembly syntax: `eudiv16 $rs, $imm, $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1													1	1	0	1	0	0	1	1	imm		rs			rd	

UDIVrr, Format: RR, Assembly syntax: `udiv16 $rs, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	0	1	0				rs		rd

B.1.52 UMULri, UMULri_ext, UMULrr

UMULri, Format: RRI2, Assembly syntax: `umul16 $rs, $imm, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	imm		rs			rd		

UMULri_ext, Format: EXTEND RRI2, Assembly syntax: `eumul16 $rs, $imm, $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1													1	1	0	1	0	0	1	0	imm		rs			rd	

UMULrr, Format: RR, Assembly syntax: `umul16 $rs, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	0	0	1				rs		rd

B.1.53 WRY

Format: RR, Assembly syntax: `wr16 %y, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	1	1	1	0	0	0	0	rd		

B.1.54 XNORri, XNORri_ext, XNORrr

XNORri, Format: RRI2, Assembly syntax: `xnor16 $rs, $imm, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	1	imm		rs			rd		

XNORri_ext, Format: EXTEND RRI2, Assembly syntax: `exnor16 $rs, $imm, $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	imm											1	1	0	1	0	1	1	1	imm		rs		rd			

XNORrr, Format: RR, Assembly syntax: `xnor16 $rs, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	0	1	1	rs			rd		

B.1.55 XORri, XORri_ext, XORrr

XORri, Format: RRI2, Assembly syntax: `xor16 $rs, $imm, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	imm		rs			rd		

XORri_ext, Format: EXTEND RRI2, Assembly syntax: `exor16 $rs, $imm, $rd`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	imm											1	1	0	1	0	1	0	1	imm		rs		rd			

XORrr, Format: RR, Assembly syntax: `xor16 $rs, $rd`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	0	1	1	rs			rd		