

# **Second-Generation Stack Computer Architecture**

**Charles Eric LaForest**

A thesis  
presented to the Independent Studies Program  
of the University of Waterloo  
in fulfilment of the  
thesis requirements for the degree  
Bachelor of Independent Studies (BIS)



Independent Studies  
University of Waterloo  
Canada  
April 2007



# Declaration

I hereby declare that I am the sole author of this research paper.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Signature:

I further authorize the University of Waterloo to reproduce this research paper by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Signature:

The work in this research paper is based on research carried out in the Independent Studies Program at the University of Waterloo, Canada. No part of this thesis has been submitted elsewhere for any other degree or qualification and is all my own work unless referenced to the contrary in the text.

**Copyright © 2007 by Charles Eric LaForest.**

The copyright of this thesis rests with the author. Quotations and information derived from it must be acknowledged.

# **Second-Generation Stack Computer Architecture**

**Charles Eric LaForest**

**Submitted for the degree of Bachelor of Independent Studies  
April 2007**

## **Abstract**

It is commonly held in current computer architecture literature that stack-based computers were entirely superseded by the combination of pipelined, integrated microprocessors and improved compilers. While correct, the literature omits a second, new generation of stack computers that emerged at the same time. In this thesis, I develop historical, qualitative, and quantitative distinctions between the first and second generations of stack computers. I present a rebuttal of the main arguments against stack computers and show that they are not applicable to those of the second generation. I also present an example of a small, modern stack computer and compare it to the MIPS architecture. The results show that second-generation stack computers have much better performance for deeply nested or recursive code, but are correspondingly worse for iterative code. The results also show that even though the stack computer's zero-operand instruction format only moderately increases the code density, it significantly reduces instruction memory bandwidth.

# Acknowledgements

Firstly, thanks go to my family, immediate and extended, who have always given me the leeway and support I needed, who always believed in me.

Sometime in 2000, Ralph Siemsen and Andrew E. Mileski introduced me to the Forth programming language, which changed my view of programming. Soon after, I discovered the microprocessors of Chen-Hanson Ting, Jeff Fox, and Charles H. (Chuck) Moore, which did the same for my view of computer hardware. Aaron Holtzman suggested I play with FPGA simulations of these computers, and graciously bore all my grumblings about broken Verilog compilers. At the same time, I had stimulating email discussions with Myron Plichota and Jecel Mattos de Assumpcao Jr. which led to some of the new ideas in this thesis.

It was Sheryl Cronk who eventually gave me the arguments and reasons to return to University. Many friends bought my old junk and helped me move. For this kick-start and support, I am forever grateful.

Once at Waterloo, Professor Chrysanne DiMarco became my Adviser. Her thorough knowledge of the English language and of the customs of academia improved me greatly. Thus, I must atone by myself for any linguistic errors in this thesis. Professors Giuseppe Tenti and Barry Ferguson unrusted and expanded my mathematical skills. Professor Manoj Sachdev and his PhD students, Shahab Ardalan and Bhaskar Chatterjee, took much time both inside and outside of class to discuss the details of VLSI circuitry with me. Professors Mark Aagaard helped me gain a broader perspective on computer architecture and led me to the class of Professor Paul Dasiewicz who taught me more about the subject. PhD candidate Brad Lushman took time to help me with my exploration of programming languages. I also thank Professor Anne Innis Dagg of Independent Studies, whose course on independent research rounded me out well.

Outside of class, the denizens of the Computer Science Club provided both enthusiastic discussions and gave me a chance to make my work heard. Mike Jeays provided me with a useful and rare primary source for the KDF9, his favourite computer. Professor Steven M. Nowick of Columbia University helped me understand his MINIMALIST synthesis tool.

The wheels of Independent Studies were kept turning by Professors Bill Abbott and Richard Holmes and especially by Susan Gow, who provided endless enthusiasm, countless good examples, and sage advice.

The writing of this thesis was supervised by Dr. Andrew Morton here at Waterloo, and by Professor J. Gregory Steffan of the University of Toronto. I am very grateful for their feedback and guidance.

And finally, thanks to Joy, my fiancée. You brighten my life. You make me happy.

The years ahead with you glow with promise and adventure.

”But the speed was power, and the speed was joy, and the speed was pure beauty.”

– Richard Bach, *Johnathan Livingston Seagull*

“If my calculations are correct, when this baby hits eighty-eight miles per hour,  
you’re gonna see some serious shit.”

– Emmet “Doc” Brown, in *Back To The Future*

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Research Goals . . . . .	2
1.2	Thesis Outline . . . . .	3
1.2.1	Part I: Historical Review . . . . .	3
1.2.2	Part II: Qualitative Arguments . . . . .	3
1.2.3	Part III: Quantitative Arguments . . . . .	3
	<b>I Historical Review</b>	<b>5</b>
<b>2</b>	<b>History of the First Generation of Stack Computers</b>	<b>7</b>
2.1	Lukasiewicz and the First Generation . . . . .	7
2.1.1	Poland: Jan Lukasiewicz (1878-1956) . . . . .	7
2.1.2	Germany: Konrad Zuse (1910 - 1995) . . . . .	8
2.1.3	Germany: Friedrich Ludwig Bauer (1924-) . . . . .	8
2.1.4	Australia: Charles Leonard Hamblin (1922-1985) . . . . .	9
2.1.5	USA: Robert Stanley Barton . . . . .	10
2.2	The First Generation of Stack Computers . . . . .	11
2.2.1	Zuse Z4 . . . . .	11
2.2.2	English Electric Co. KDF9 . . . . .	12
2.2.3	Burroughs B5000 and later models . . . . .	14
2.2.4	International Computers Ltd. ICL2900 series . . . . .	16
2.2.5	Hewlett-Packard HP3000 . . . . .	17
2.3	Shortcomings and Disappearance of the First Generation . . . . .	18
2.3.1	Explicit High-Level Language Support . . . . .	18
2.3.2	The Rise of RISC . . . . .	18
2.3.3	Excessive Memory Traffic . . . . .	19
2.3.4	The Need for Index Registers . . . . .	20
<b>3</b>	<b>History of the Second Generation of Stack Computers</b>	<b>21</b>
3.1	Charles H. Moore and the Second Generation . . . . .	21
3.1.1	Charles Havice (Chuck) Moore II . . . . .	21
3.1.1.1	The Forth Programming Language Basis of Second-Generation Stack Computers . . . . .	21
3.1.2	Philip J. Koopman, Jr. . . . .	22

3.2	The Second Generation of Stack Computers . . . . .	23
3.2.1	NOVIX NC4016 . . . . .	23
3.2.2	Harris RTX-2000 . . . . .	23
3.2.3	Sh-BOOM (Patriot Scientific IGNITE I) . . . . .	23
3.2.4	MuP21 . . . . .	25
3.2.5	F21 . . . . .	26
3.2.6	c18 . . . . .	26
3.3	Recent Research . . . . .	26
3.4	Strengths and Weaknesses of the Second Generation . . . . .	28
3.4.1	The Need for Index Registers . . . . .	28
3.4.2	Stack Manipulation Overhead . . . . .	28
3.4.3	Poor Support of ALGOL-like Languages . . . . .	29
3.4.4	Reduced Instruction Memory Bandwidth and System Complexity . . . . .	29
3.4.5	Fast Subroutine Linkage and Interrupt Response . . . . .	29

## **II Qualitative Arguments 31**

<b>4</b>	<b>Distinguishing the First and Second Generations</b>	<b>33</b>
4.1	Location of Stacks: In-Memory vs. In-Processor . . . . .	34
4.2	Use of Stacks: Procedure Nesting vs. Expression Evaluation . . . . .	35
4.3	Operations with Stacks: High-Level Language Support vs. Primitive Operations	36
<b>5</b>	<b>Objections Cited by Hennessy &amp; Patterson</b>	<b>37</b>
5.1	The Enormous Influence of Hennessy & Patterson on Computer Architecture . . . . .	37
5.2	The Disappearance of Stack Computers (of the First Generation) . . . . .	38
5.3	Expression Evaluation on a Stack . . . . .	39
5.4	The Use of the Stack for Holding Variables . . . . .	40
5.5	Distorted Historical Arguments . . . . .	40

## **III Quantitative Arguments 45**

<b>6</b>	<b>A Stack-Based Counterpart to DLX: Gullwing</b>	<b>47</b>
6.1	Block Diagram . . . . .	47
6.1.1	Memory Subsystem . . . . .	48
6.1.1.1	Single Memory Bus . . . . .	48
6.1.1.2	Differentiating Loads, Stores, and Fetches . . . . .	48
6.1.2	Computation Subsystem . . . . .	48
6.1.3	Control Subsystem . . . . .	48
6.2	Instruction Set . . . . .	49
6.2.1	Instruction Packing . . . . .	49
6.2.2	Flow Control . . . . .	49
6.2.3	Load, Store, and Literal Fetch . . . . .	50
6.2.4	Arithmetic and Logic . . . . .	50

6.2.4.1	Synthesizing More Complex Operations . . . . .	51
6.2.5	Stack Manipulation . . . . .	52
6.2.6	No-Op and Undefined . . . . .	52
6.2.7	Instruction Format and Execution Example . . . . .	53
6.3	State Machine and Register Transfer Description . . . . .	54
6.3.1	Improvement: Instruction Fetch Overlap . . . . .	56
<b>7</b>	<b>Comparisons With DLX/MIPS</b>	<b>59</b>
7.1	Gullwing Benchmarks . . . . .	59
7.1.1	Flight Language Kernel (Bare) . . . . .	59
7.1.2	Flight Language Extensions (Ext.) . . . . .	60
7.1.3	Virtual Machine (VM) . . . . .	60
7.2	Comparison of Executed Benchmark Code . . . . .	61
7.2.1	Dynamic Instruction Mix . . . . .	61
7.2.2	Cycles Per Instruction . . . . .	64
7.2.3	Memory Accesses Per Cycle . . . . .	65
7.2.4	Instructions per Memory Word . . . . .	67
7.2.4.1	Basic Blocks and Instruction Fetch Overhead . . . . .	68
7.3	Behaviour of Iteration, Recursion, and Subroutine Calls . . . . .	68
7.3.1	Measured Properties . . . . .	68
7.3.2	Demonstrators . . . . .	69
7.3.3	Iterative Triangular Numbers . . . . .	70
7.3.4	Recursive Triangular Numbers . . . . .	72
7.3.5	Tail-recursive Triangular Numbers . . . . .	74
7.3.6	Subroutine Calls . . . . .	76
7.4	Pipelining . . . . .	79
7.4.1	Transforming the DLX Pipeline to Gullwing . . . . .	79
7.4.2	Altering the ISR to Deal with the Additional Latency . . . . .	81
7.4.3	The Effect of Pipelining on Calls, Jumps, and the CPI . . . . .	83
7.5	Summary and Performance Comparison . . . . .	84
<b>8</b>	<b>Improving Code Density</b>	<b>85</b>
8.1	Improving High-Level Code Density by Adding an Instruction Stack . . . . .	86
8.1.1	Side-Effects on Return Stack Manipulation . . . . .	87
8.2	Implementation . . . . .	88
8.3	Side-Effect on Code Size, Silicon Area, and Subroutine Overhead . . . . .	88
8.3.1	The Instruction Stack as an Instruction Cache . . . . .	89
<b>9</b>	<b>Conclusions, Contributions, and Further Work</b>	<b>91</b>
9.1	Contributions . . . . .	93
9.2	Further Work . . . . .	93
9.2.1	Reducing the DLX/MIPS Subroutine Call Overhead by Adding Stacks . . . . .	94
9.2.2	Reducing Gullwing's Instruction Count with Compound Stack Operations . . . . .	96

9.2.3	Reducing Gullwing's CPI by Executing Multiple Instructions using Generalized Instruction Folding . . . . .	97
<b>A</b>	<b>Gullwing Benchmarks Source</b>	<b>99</b>
A.1	Flight Language Kernel . . . . .	99
A.1.1	Internal Variables and Memory Map . . . . .	99
A.1.1.1	Counted Strings . . . . .	100
A.1.2	Utility Functions . . . . .	101
A.1.3	String Functions . . . . .	101
A.1.4	Input Functions . . . . .	102
A.1.5	Name Lookup . . . . .	103
A.1.6	Function Definition Functions . . . . .	104
A.1.7	Compilation Functions . . . . .	106
A.1.8	Inline Compilation . . . . .	108
A.1.9	Main Loop . . . . .	108
A.1.10	Decimal to Binary Conversion . . . . .	109
A.2	Flight Language Extensions . . . . .	109
A.2.1	Making the Flight Language More Tractable . . . . .	110
A.2.2	Interactively Usable Opcodes . . . . .	112
A.2.3	Basic Compiling Functions . . . . .	112
A.2.4	Terminal Control Characters . . . . .	113
A.2.5	Conditionals and Comparisons . . . . .	113
A.2.6	Code Memory Allocation . . . . .	115
A.2.7	String Copying and Printing . . . . .	115
A.2.8	De-Allocating Functions . . . . .	116
A.2.9	Unsigned Multiplication and Division . . . . .	117
A.2.10	Binary to Decimal Conversion . . . . .	118
A.2.11	Simple Fibonacci Examples . . . . .	119
A.2.12	Static Variables . . . . .	120
A.2.13	Accumulator Generator . . . . .	121
A.2.14	Fibonacci Generator . . . . .	121
A.2.15	Caesar Cipher Generator . . . . .	122
A.2.16	Higher-Order Function (Map) . . . . .	124
A.3	Virtual Machine . . . . .	125
A.3.1	VM . . . . .	125
A.3.2	Metacompiler . . . . .	131
A.3.3	Self-hosted Kernel . . . . .	135
A.3.4	Flight Language Extensions . . . . .	140
<b>B</b>	<b>Static and Dynamic Gullwing Code Analyses</b>	<b>141</b>
B.1	Static Analyses . . . . .	141
B.1.1	Memory Usage . . . . .	141
B.1.2	Range of Literals . . . . .	142
B.1.3	Range of Addresses . . . . .	142
B.1.4	Instructions per Instruction Word . . . . .	143

B.1.5	Instruction Density . . . . .	143
B.1.6	Compiled Instruction Counts . . . . .	144
B.2	Dynamic Analyses . . . . .	145
B.2.1	Overall Execution . . . . .	145
B.2.2	Executed Instruction Counts . . . . .	145
B.2.3	Average CPI . . . . .	147
B.2.4	Instruction Types . . . . .	147
B.2.5	Basic Block Length . . . . .	148
B.2.6	Data Stack Depth . . . . .	149
B.2.7	Return Stack Depth . . . . .	150

<b>Bibliography</b>	<b>151</b>
---------------------	------------



# List of Tables

5.1	Comparison of Citations of Computer Architecture Texts (as of Fall 2004)	38
6.1	Gullwing Flow Control Instructions	50
6.2	Gullwing Load and Store Instructions	50
6.3	Gullwing ALU Instructions	51
6.4	Gullwing Stack Manipulation Instructions	52
6.5	Gullwing No-Op and Undefined Instruction	52
7.1	Compilers Dynamic Instruction Mix	63
7.2	Interpreters Dynamic Instruction Mix	64
7.3	DLX CPI with Load and Branch Penalties	65
7.4	Gullwing CPI by Instruction Type	65
7.5	Gullwing Memory Accesses Per Cycle (Total)	66
7.6	DLX/MIPS Memory Accesses Per Cycle Caused by Loads and Stores	67
7.7	Triangular Iterative Code Comparison	71
7.8	Iterative Dynamic Instruction Mix	71
7.9	Triangular Recursive Code Comparison	73
7.10	Recursive Dynamic Instruction Mix	73
7.11	Triangular Tail-Recursive Code Comparison	75
7.12	Tail-Recursive Dynamic Instruction Mix	75
7.13	Add2 Code Comparison	76
7.14	Add2 Dynamic Instruction Mix	76
7.15	Add3 Dynamic Instruction Mix	77
7.16	Add3 Code Comparison	77
7.17	Add4 Dynamic Instruction Mix	78
7.18	Add4 Code Comparison	78
9.1	Synthesized Stack Operations on MIPS with Stacks	95
9.2	Recursive MIPS32 Instruction Distribution With and Without Stacks	95
9.3	Triangular Recursive MIPS32 Code Comparison With and Without Stacks	95
B.1	Compiled Flight Code Memory Usage	141
B.2	Range of Literals by Absolute Value	142
B.3	Range of Addresses by Absolute Value	142
B.4	Instructions per Instruction Word	143
B.5	Instruction Density	143

B.6	Compiled Instruction Counts . . . . .	144
B.7	Overall Execution . . . . .	145
B.8	Executed Instruction Counts . . . . .	146
B.9	Average CPI . . . . .	147
B.10	Instruction Types . . . . .	147
B.11	Basic Block Length . . . . .	148
B.12	Data Stack Depth . . . . .	149
B.13	Return Stack Depth . . . . .	150

# List of Figures

2.1	Evaluation of Polish Notation expression / + 5 5 2 . . . . .	8
2.2	Fig. 1 from Bauer and Samelson German Patent #1094019 . . . . .	9
2.3	Programming model for the Zuse Z4 . . . . .	11
2.4	KDF9 Q-Store Layout . . . . .	12
2.5	KDF9 Block Diagram . . . . .	13
2.6	B6900 Top-of-Stack and Stack Bounds Registers . . . . .	15
2.7	B7700 Stack Buffer and Stack Memory Area . . . . .	15
2.8	Comparison of B6700 and ICL 2900 stack mechanisms . . . . .	16
2.9	HP3000 Stack Registers . . . . .	17
3.1	NC4016 and RTX-2000 Block Diagrams . . . . .	24
3.2	IGNITE I Block Diagram . . . . .	25
4.1	First-Generation Stack Computer Block Diagram . . . . .	34
4.2	General-Purpose Register Computer Block Diagram . . . . .	34
4.3	Second-Generation Stack Computer Block Diagram . . . . .	35
6.1	Gullwing Block-Level Datapath . . . . .	47
6.2	Gullwing Instruction Shift Register Block Diagram . . . . .	47
6.3	Gullwing Instruction Format . . . . .	53
7.1	DLX Pipeline Block Diagram . . . . .	79
7.2	Gullwing Pipeline Block Diagram . . . . .	80
7.3	Gullwing Pipeline Operation . . . . .	81
7.4	Gullwing Load/Stores Pipeline Diagram . . . . .	81
7.5	Gullwing ISR Modified for Pipeline . . . . .	82
7.6	Gullwing Instruction Fetch (with Overlap) Pipeline Diagram . . . . .	82
7.7	Gullwing Instruction Fetch (without Overlap) Pipeline Diagram . . . . .	83
7.8	Gullwing Taken Jumps or Calls Pipeline Diagram . . . . .	83
8.1	Gullwing High-Level Code with Unavailable Slots . . . . .	85
8.2	Instruction Stack During Call and Return . . . . .	86
8.3	Gullwing High-Level Code with Available Slots . . . . .	86
8.4	Instruction Stack During >R and R> . . . . .	87
9.1	MIPS Register File with Stacks . . . . .	94

A.1	Flight Language Kernel Memory Map . . . . .	100
A.2	Counted String Format . . . . .	100

# List of Algorithms

1	Gullwing Synthesis of Subtraction and Bitwise OR . . . . .	51
2	Gullwing Synthesis of Multiplication (4x4) . . . . .	51
3	Gullwing Flow Control Instructions . . . . .	54
4	Gullwing No-Op and Undefined Instructions . . . . .	54
5	Gullwing ALU Instructions . . . . .	55
6	Gullwing Load and Store Instructions . . . . .	55
7	Gullwing Stack Instructions . . . . .	55
8	Gullwing ALU Instructions with Instruction Fetch Overlap . . . . .	57
9	Gullwing Stack Instructions with Instruction Fetch Overlap . . . . .	57
10	Gullwing No-Op and Undefined Instructions with Instruction Fetch Overlap . . . . .	57
11	Triangular Iterative C Source . . . . .	71
12	Triangular Iterative MIPS32 Assembly . . . . .	71
13	Triangular Iterative Gullwing Assembly . . . . .	71
14	Triangular Recursive C Source . . . . .	72
15	Triangular Recursive MIPS32 Assembly . . . . .	73
16	Triangular Recursive Gullwing Assembly . . . . .	73
17	Triangular Tail-Recursive C Source . . . . .	74
18	Triangular Tail-Recursive MIPS32 Assembly . . . . .	75
19	Triangular Tail-Recursive Gullwing Assembly . . . . .	75
20	Add2 C Source . . . . .	76
21	Add2 MIPS32 Assembly . . . . .	76
22	Add2 Gullwing Assembly . . . . .	76
23	Add3 C Source . . . . .	77
24	Add3 MIPS32 Assembly . . . . .	77
25	Add3 Gullwing Assembly . . . . .	77
26	Add4 C Source . . . . .	78
27	Add4 MIPS32 Assembly . . . . .	78
28	Add4 Gullwing Assembly . . . . .	78
29	Alterations to Gullwing to Support an Instruction Stack . . . . .	88
30	Triangular Recursive MIPS32 Assembly with Stacks Added . . . . .	95
31	Gullwing Compound Stack Operations . . . . .	96
32	Example Gullwing Instruction Sequence Using Generalized Folding . . . . .	97



# Chapter 1

## Introduction

I first learnt about stack computers in 2000 while working at a computer manufacturer where co-workers introduced me to the Forth programming language, a stack-based programming environment. Soon after, while looking for a suitable processor for a homebrew computer system, I came across a mention of the MuP21 [MT95] in the Usenet *Embedded Processor and Microcontroller Primer and FAQ*<sup>1</sup>:

The MuP21 was designed by Chuck Moore, the inventor of Forth. With the MuP21, Forth can compile into machine code and still be Forth, because the machine code IS Forth. The MuP21 freaks out at 100 MIPS while consuming only 50 milliwatts. Not only that, the chip includes a video generator, has only about 7000 transistors (that's right, 7000 and not 7,000,000), and costs about \$20.

The assembler on this chip is a sort of dialect of Forth, as the CPU is modeled after the Forth virtual machine. MuP21 is a MINIMAL Forth engine. [...] The CPU programs the video generator and then just manipulates the video buffer. It is composite video out, so it only needs one pin. MuP21 is only a 40 pin chip.

I'd never heard of anything like it. It was smaller and faster, and its machine code was a structured language! I was hooked. Understanding this type of hardware and software became a hobby that ultimately led me to pursue a University degree on the topic. However, I couldn't simply take a Computer Engineering degree since this kind of computer is virtually non-existent in the mainstream literature and totally absent from the curriculum. Therefore, I had to create one under the aegis of the Independent Studies (IS) program.

The IS program is a self-directed course of study guided and vetted by a Faculty Adviser and composed of a combination of Independent Study Units and regular courses. After two years of study (typically), a student petitions to enter Thesis Phase and if approved, spends a year developing a thesis on a selected topic. A successfully completed thesis grants the degree of Bachelor of Independent Studies (BIS). Overall, IS bears more resemblance to graduate studies than undergraduate ones.

The structure of this thesis reflects the directions I have taken throughout the IS program. I began with broad historical explorations of stack architecture and programming languages, complemented by regular engineering courses on digital systems, computer architecture, and

---

<sup>1</sup>Copyright (c) 1997 by Russ Hersch, all rights reserved. <http://www.faqs.org/faqs/microcontroller-faq/primer/>

integrated circuits. These efforts eventually concentrated on defining, simulating, programming, and partially implementing a particular stack computer design. In this thesis, I leave aside the issues of programming language design and VLSI implementation to focus on the architecture of the computer itself.

## 1.1 Research Goals

A stack computer performs its operations not upon a randomly accessible set of registers, but upon a simpler, linear list of such. This list is conveniently viewed as a pushdown stack with the visible registers at the top. Since virtually all arithmetic and logical operations are either unary or binary, at a minimum the top two elements of a stack need to be accessible. The operations implicitly access these locations for operands and return values. The stack can be used for evaluating expressions in the manner of Reverse Polish Notation and also for storing a chain of activation records (stack frames) of nested subroutines.

The main problem in reasoning about stack computers is that those usually mentioned in the computer architecture literature, the first generation, have been superseded. They were popular due to their efficient use of hardware, their natural application towards algebraic computation, and the ease with which they could be programmed. Although sophisticated, they were eventually flatly outperformed by the new VLSI microprocessors that came out of the Berkeley RISC [PS81] and Stanford MIPS [HJP<sup>+</sup>82] projects. The first generation of stack computers can be defined by its support for High-Level Language, mainly ALGOL. This required in-memory stacks primarily used to allocate storage for nested procedures, and encouraged a large instruction set which attempted to match the semantics of ALGOL closely. The goal was to make programming easier in the absence of good compilers.

The second generation of stack computers arose just as the first faded away. These computers had simple hardware, high code density, fast subroutine linkage and interrupt response, but garnered little attention since they were aimed at embedded systems instead of general-purpose computing. This separated the second generation from the mainstream of processor design and caused it to become confused with the first generation, further discouraging work. The second generation of stack computers can be defined by its support for the Forth programming language. It defines two stacks, Data and Return, which are separate from main memory and not randomly addressable. The Data Stack is used to evaluate expressions and pass values between procedures in a functional manner. The Return Stack holds the return addresses of subroutines and can also act as temporary storage. The small instruction set is mostly composed of Forth primitives, which are simple stack manipulations, loads and stores, and basic arithmetic and logical functions similar to those found in conventional register-based computers.

The purpose of this thesis is to argue for a distinction of stack computers into first and second generations. I do this by recapitulating the evolution of stack computers, revisiting old arguments against them, and comparing the design of a model second-generation stack computer to a modern computer architecture. Given this refreshed view, I hope to fill the gap in the literature about stack computers and uncover some interesting avenues in computer architecture.

## 1.2 Thesis Outline

This thesis is divided into three major parts: a Historical Review, Qualitative Arguments, and Quantitative Arguments. The first and third may be read independently. However, the second part depends on the background provided by the first and is supported by data from the third.

### 1.2.1 Part I: Historical Review

Current computer literature only briefly touches upon stack architecture, always from the first generation, and usually as an introductory contrast to register-based computers. Chapter 2 provides a more detailed summary of the history of the people and machines that make up the first generation of stack computers, starting with their conceptual origins and ending with the main reasons for their downfall. It uncovers two different fundamental approaches to the design of stack computers: support for the ALGOL programming language, and composition of functions. This difference turns out to be the main criterion for distinguishing first-generation stack computers from second-generation ones.

Chapter 3 contains an overview of the second generation of stack computers. It focuses on the latest wave of such machines which originated with the work of Charles H. Moore and were extensively studied by Philip J. Koopman. It gathers together the scattered publications on the subject and also much information that was never formally published.

### 1.2.2 Part II: Qualitative Arguments

Before any comparison can be made between second-generation stack computers and current register-based computers, the confusion about stack computers in the mainstream literature must first be addressed. Chapter 4 proposes a set of three criteria to divide stack computers into a first and a second generation. They concern the location of the stacks, their purpose, and the operations done upon them. Stacks in a second-generation computer resemble registers used for calculations and parameter-passing, while the stacks of a first-generation machine are effectively call stacks holding procedure activation records.

With these criteria and the historical data in mind, Chapter 5 addresses the arguments against stack architectures cited by Hennessy & Patterson. These arguments are found to rely on outdated assumptions about compiler and hardware technology, and have also been distorted through secondhand citations. The original arguments are cited, and found to be much less critical of stack architectures than suggested by Hennessy & Patterson.

### 1.2.3 Part III: Quantitative Arguments

Given that past arguments have been found lacking, the comparison between second-generation stack computers and current register-based computers needs to be revisited. Chapter 6 describes in detail the design of a small, modern stack computer architecture, named 'Gullwing', along with a simple optimization to its instruction fetch mechanism which makes practical the use of a single memory bus for both instructions and data.

Chapter 7 compares Gullwing to the DLX and MIPS processors used as demonstrators by Hennessy & Patterson. The processors are compared with aggregate benchmarks and with

low-level analyses of how they execute iterative, recursive, tail-recursive, and nested subroutine code. The issue of pipelining Gullwing is explored as a transformation of the DLX pipeline. Gullwing is found to have a definite advantage at subroutine calls and memory bandwidth, but is unfortunately architecturally equivalent to a DLX processor without load or branch delay slots, with the same penalty to performance.

Chapter 8 addresses Gullwing's inefficient usage of memory for holding compiled code by adding a third stack to temporarily hold instructions during subroutine calls. This new architectural feature will increase the density of code to the maximum possible value and accelerate returns from subroutines.

Finally, Section 9.2 outlines the addition of stacks to a MIPS processor, without altering the pipeline or instruction set, in order to give it the efficient subroutine call mechanism of a stack computer. This section also introduces the addition of two forms of parallelism to Gullwing: one which reduces its instruction count with compound stack operations, and the other which reduces its CPI by overlapping the execution of instructions.

Appendix A provides the source to the Flight language kernel and the software used to benchmark Gullwing. Appendix B contains the tabulated raw data from the analyses of the dynamic and static properties of Gullwing machine code.

**Part I**  
**Historical Review**



## Chapter 2

# History of the First Generation of Stack Computers

I present here the first generation of stack computers in the context of the pioneers of the field and of the machines that followed their insights. I discuss the organization and design goals of these computers, their shortcomings, and ultimately their replacement by RISC designs.

### 2.1 Lukasiewicz and the First Generation

The idea of using stacks for computation seems to have occurred independently, in slightly different forms, to several people worldwide within an interval of about a decade. It is difficult to tell if they were aware of each other's work at the time. Nonetheless, there seems to be a chronological order to the discoveries.

#### 2.1.1 Poland: Jan Lukasiewicz (1878-1956)

In 1929, while a professor at Warsaw University, Lukasiewicz wrote *Elements of Mathematical Logic* [Luk29]. In it he introduced a parenthesis-free notation for arithmetic and logic which eventually became known as Polish Notation or Prefix Notation. Its main feature is that it makes the order of operations explicit, contrary to the usual algebraic notation (correspondingly called Infix Notation) which depends on a knowledge of operator precedence and the use of parentheses to override it where necessary.

For example, the expression  $(5 + 5)/2$  requires the use of parentheses to specify that the result should be 5 and not 7.5 due to the higher precedence of the division operator. The equivalent Prefix expression  $/ + 5 5 2$  is unambiguous and can be evaluated left-to-right by leaving the application of an operator pending until enough operands are available. The alternative interpretation of the infix expression would be written in prefix notation as  $+ / 5 2 5$  or  $+ 5 / 5 2$ . Figure 2.1 shows how the expression is evaluated one symbol at a time. It is easy to see how the operators and operands could each reside in separate stacks until they are respectively executed or consumed.

$$\begin{array}{r}
 / \\
 / + \\
 / + 5 \\
 / + 5 5 \\
 / 10 \\
 / 10 2 \\
 5
 \end{array}$$

Figure 2.1: Evaluation of Polish Notation expression  $/ + 5 5 2$

### 2.1.2 Germany: Konrad Zuse (1910 - 1995)

The case of Konrad Zuse is unusual. He did his work privately, outside of academia or industry, and it was destroyed multiple times during the World War II air raids on Berlin. He also did not base his work on Lukasiewicz, but appeared to have come to the use of a stack out of simple engineering need. He constructed a series of computers of increasing capability, arriving at the stack-based Z4 in 1945, predating all other stack computers by at least 15 years [BFPB97, 10.3]. Unfortunately, except for the various machines produced up to 1969 by the Zuse KG company, there are no architectural descendants of the Z4 in Germany or abroad.

### 2.1.3 Germany: Friedrich Ludwig Bauer (1924-)

The earliest known mechanical realization of Lukasiewicz's idea was Bauer's STANISLAUS relay calculator [Bau60], first conceived in 1950/1951. It emerged out of the desire to mechanically test the well-formedness of formulae. The publication of this achievement was delayed by the need for secrecy while patents for the evaluation method were filed in Germany, the United States, France, the United Kingdom, and Sweden [BSc][BSd][BSa][BSb]. Figure 2.2 shows Fig. 1 from the original German patent, clearly showing an 'OperationsKeller' (Operations Cellar) and a 'ZahlKeller' (Number Cellar) used to evaluate Polish Notation expressions. The method is also discussed in a paper published after the patents were filed in 1957/1958 [SB60].

This 'cellar principle', now referred to as the stack principle, made its way into a proposal for an International Algebraic Language [Car58, BBR58] as the natural method for the block structure of storage allocation for subroutines [Bau90]. This language evolved into ALGOL 60 [BBG<sup>+</sup>60]. Its use of a dynamic stack to support subroutine nesting and recursion has since become the dominant organizing principle of programming languages. It is important to note that the support of this structure is one of the hallmarks of first-generation stack computers (Figure 4.1).

In a recent talk [Bau02, BD02], Bauer mentioned some other appearances of the stack principle:

*Hardware* cellars, stacks, and pushdown stores have been discussed elsewhere, possibly as early as 1947 by Alan Turing, certainly in 1949 by Harry D. Huskey

in connection with the ZEPHYR (SWAC) computer and in 1956 by Willem L. van der Poel in connection with the design of the MINIMA computer; in all cases presumably for the treatment of return jumps in subroutines. [...]

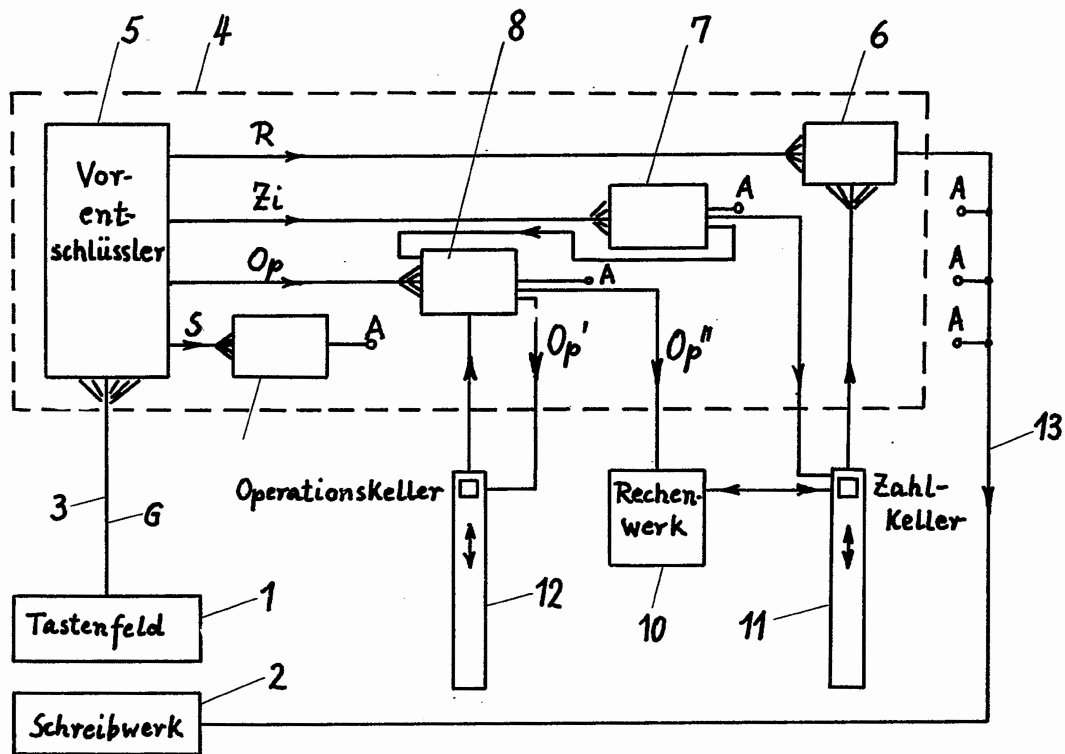


Fig.1

Figure 2.2: Fig. 1 from Bauer and Samelson German Patent #1094019

### 2.1.4 Australia: Charles Leonard Hamblin (1922-1985)

Facing the tedium of the programming systems of the time, Hamblin independently discovered the importance of Lukasiewicz's work for expressing formulae but took it into a slightly different direction. Also, because of the secrecy during the pre-filing period of Bauer and Samelson's patents, he could not have known of their work.

The key change Hamblin made was reversing the order of the notation, placing the operands before the operator. This 'reverse Polish' notation kept the operators in the same order as in the original infix notation and removed the need for delaying the application of an operator since it would arrive only after its operands. This made straightforward the translation of an expression

into a sequence of machine instructions. For example, the expression  $(5 + 5)/2$  is expressed unambiguously as  $5\ 5\ +\ 2\ /$ , while the alternative interpretations (assuming no parentheses) would be written as  $5\ 2\ / 5\ +$  or  $5\ 5\ 2\ /\ +$ . Furthermore, only a single stack is required since the operators are never waiting for their operands.

Hamblin expanded upon this insight in a pair of 1957 papers [Ham57a][Ham57b]<sup>1</sup>(reprinted [Ham85]). In summary:

It is now possible to visualize the general shape a machine designed to use such code might take. It would have a 'running accumulator' and 'nesting register' as described, and a number-store arranged on something like the pattern indicated, [...]

The running accumulator is a stack and is equivalent to Bauer's Number Cellar. The nesting register is of the same structure but holds the return addresses of subroutines. This separation of evaluation and flow-control into two stacks, which are also separate from main memory, is the main architectural feature of second-generation stack computers (Figure 4.3).

Some employees of the English Electric Co. were present when Hamblin delivered his first paper [All85][Dun77]. They integrated his ideas into their next computer, the KDF9.

### **2.1.5 USA: Robert Stanley Barton**

Just as the Bauer and Samelson patents were being granted, Barton also independently came to the same conclusions about the application of Lukasiewicz's work [Bul77]. In 1959, he proposed the design of a stack-based computer to be programmed entirely in ALGOL [Bar61a] (reprinted [Bar87]) [Bar61b][Bar61c]. The proposal took form as the Burroughs B5000, which became the archetypal first-generation stack computer design.

Barton acknowledged the work of Bauer and Samelson, but seems to have not known about Hamblin's work at the time. This, and the focus on directly supporting ALGOL, might explain the use of a single stack (per process) in the B5000.

---

<sup>1</sup>This is a slightly abridged form of the first paper.

## 2.2 The First Generation of Stack Computers

There were many more machines of this type than those I've enumerated here. I've mentioned the ones that are directly linked to the people of the previous section or which have been notable in industry. A much larger list can be found in Koopman's book [Koo89, App.A].

### 2.2.1 Zuse Z4

The Z4 is too simple to fit into either the first or second generation of stack computers, not being a stored-program machine, but it is the earliest one known and thus deserves mention. Originally built in 1945, it was damaged during World War II, and later rebuilt in 1950. It currently resides in the Deutsches Museum in Munich, Germany.

Like many of Zuse's computers, the Z4 was designed to perform engineering calculations. Its program was read from a punched plastic tape<sup>2</sup>, and it included a second tape reader as a form of subroutine call. Its ingenious mechanical main memory held 64 32-bit floating-point numbers which could be loaded into a stack of 2 elements, operated upon, and then stored back. It supported a full complement of basic arithmetic operations, including square root and some built-in constants such as  $\pi$ . Its 8-bit instruction format was zero-operand, with one-operand loads and stores for direct addressing of memory. No address calculations or access to the stack pointer were possible. It supported conditional stop, skip, and call instructions. Figure 2.3 shows the programming model for the Z4. It is a simplified reproduction from Blaauw and Brooks' book [BFPB97, fig. 10-29].

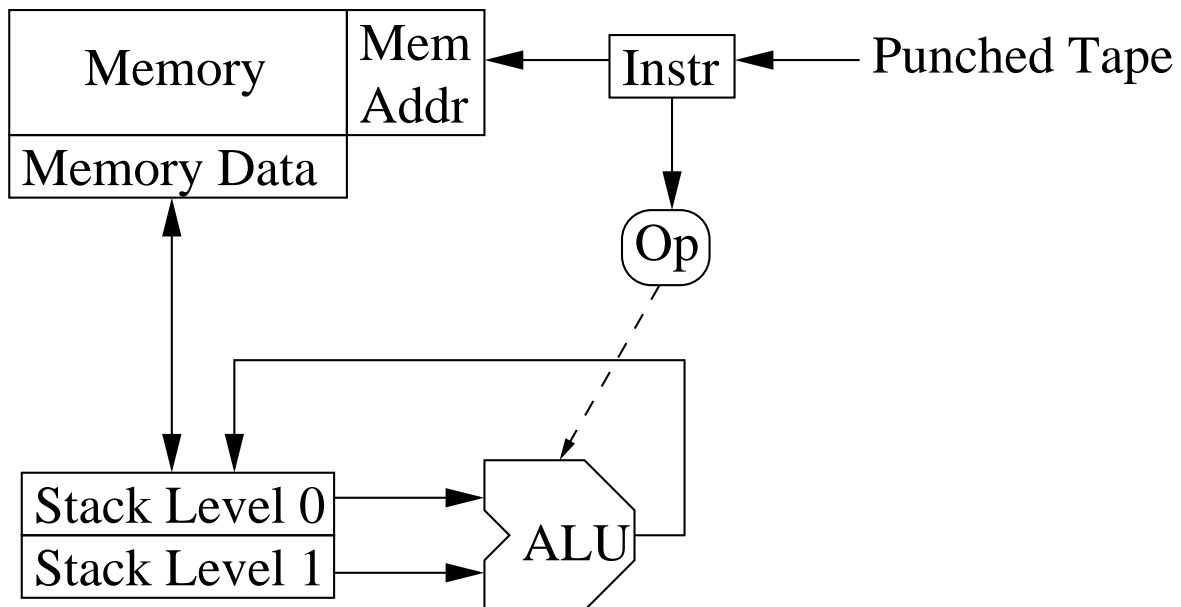


Figure 2.3: Programming model for the Zuse Z4

---

<sup>2</sup>used movie film, in fact!

## 2.2.2 English Electric Co. KDF9<sup>3</sup>

The design of the KDF9 (Figure 2.5) [Eng63, fig. 2] was inspired by Hamblin's first paper on stack-based computing [Ham57a] and thus uses a pair of stacks for its operation. The Nesting Store was a 19-deep hardware stack, with the top two elements visible to the Arithmetic Unit, upon which expressions were evaluated. The Sub-Routine Jump Nesting Store was similar, but only 16-deep with only the top-most element visible. Neither of these stacks extended into main memory. All storage locations were 48 bits wide.

A third set of 16 stores, named 'Q -Stores' (Figure 2.4) [KDF61, pg. 9], were used for random access storage, address modification, loop counting, and I/O operations. The first store, Q0, was a read-only zero register. The remainder were either used as single 48-bit registers, or triads of 16-bit registers, with direct or accumulative storage. The 16-bit sub-registers could also be used respectively as modifier, increment, and counter. An access to main memory could have its address augmented by the modifier. Afterwards, the modifier could then be incremented by the increment and the counter decremented by one. With a jump instruction to test the counter, this made for efficient loops and array processing. The counter of a Q-Store could also hold the amount of positive or negative shift for shift instructions. Finally, a Q-Store could hold a device number and the start and end addresses of an area in memory in preparation for an automated I/O operation.

The English Electric Co. went through a series of acquisitions and mergers, eventually forming International Computers Ltd. in 1968 [Lav80]. However, by then their focus seemed to have changed to competing with Burroughs' B5000 series and IBM's System/360 [Dun77] and so the dual-stack approach, and the KDF9, was dropped entirely.

The KDF9 is an oddity. Historically, it is a first-generation stack computer. However, based on the distinguishing criteria for first and second-generation stack computers (Chapter 4), it falls squarely into the second. Had it not been discontinued, the first and second generations might have existed in parallel.

### **Q-STORE: STORAGE OF 48-BITS**

(TOTAL NUMBER OF Q-STORES: 15)

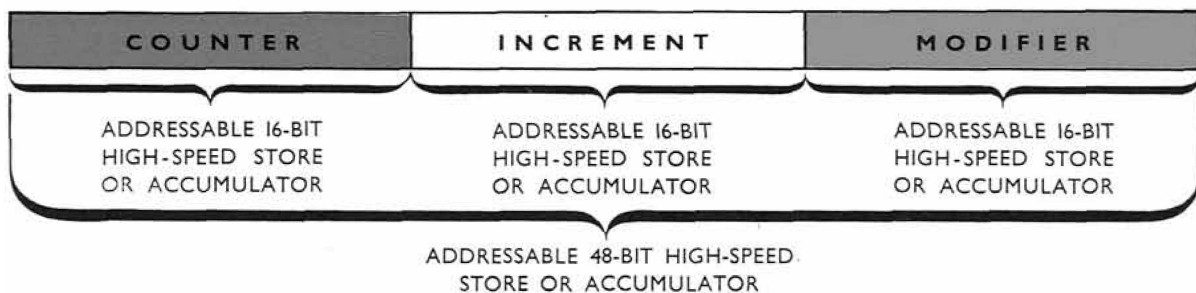


Figure 2.4: KDF9 Q-Store Layout

<sup>3</sup>The variations 'KDF.9' and 'KDF-9' are also used.

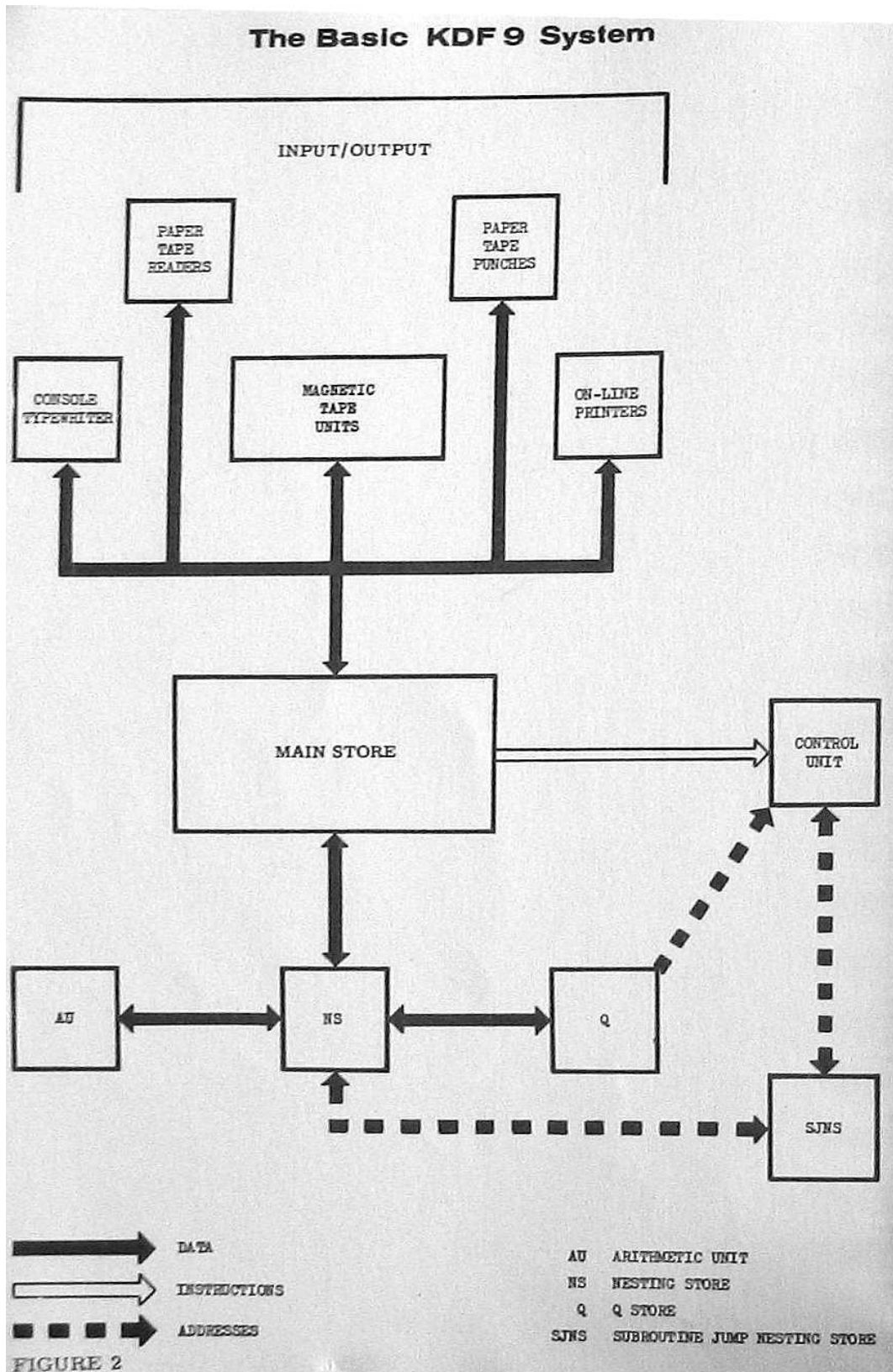


Figure 2.5: KDF9 Block Diagram

### 2.2.3 Burroughs B5000 and later models

The B5000 spawned an entire series of stack computers, all aimed at the direct and efficient execution of the ALGOL language. They were complex multiprocessing systems with tagged memory and descriptors for primitive data types and automatic management of subroutine parameters. I will concentrate here on the design and use of the single, in-memory stack that governed the execution of a program. This feature is essentially unchanged across the entire series.

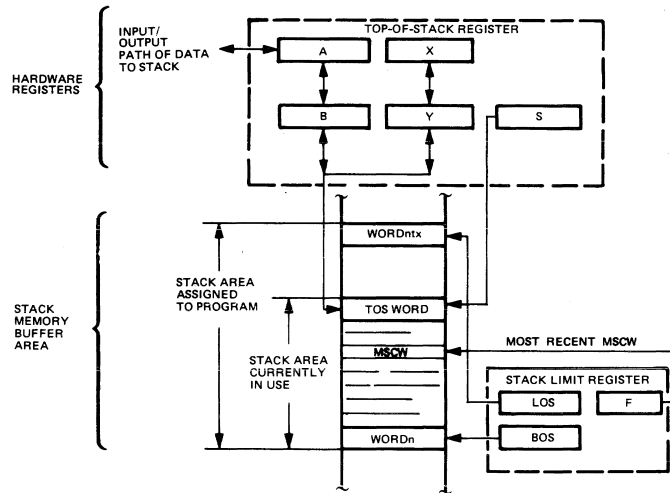
Figure 2.6 shows the implementation of the stack in the B6900 [Bur81, Sec.3]. The stack memory area is delimited by the contents of the Bottom Of Stack (BOS) and Limit Of Stack (LOS) registers. The current subroutine area is indicated by the F register which points to a Mark Stack Control Word (MSCW). This word contains the context information necessary to return to the subroutine's caller. The topmost stack element in use is pointed to by the S register.

The A and B registers are a working cache for the top of the stack and are connected to the ALU. They are extended by the X and Y registers for double-precision calculations. Their contents are loaded and unloaded as required by each operation in progress and so their entire operation is transparent to the program. They are not part of the stack proper since they are flushed whenever the top of the stack is altered by some operation, such as a subroutine call, and so cannot be used to pass parameters.

The B7700 added a 32-entry circular stack buffer between main memory and the A and B registers (Figure 2.7) [Bur73, Sec.2]. This is a genuine buffer that is transparent to the program, and is only flushed if the processor registers (including the S, F, LOSR, and BOSR registers) are altered with a SPRR (Set Processor Register) or a MVST (Move To Stack) operation, or in the case of an atomic memory exchange with the top of the stack using RDLK (Read With Lock).

Both computation and subroutine linkage were done on the same stack in a manner specifically designed to support the structure of the ALGOL programming language. When a subroutine or nested block of code was to be entered, a MSCW was placed on the stack, followed by the parameters to the subroutine, followed by a Return Control Word (RCW) which saved the condition flags, amongst other things. The local variables and temporary values were then allocated above all this. Only at this point could the Enter (ENTR) operator be executed to enter the subroutine.

In 1986, the Burroughs Corporation merged with the Sperry Corporation into the Unisys Corporation [Ros87]. The B5000 series of computers continues in the company's ClearPath line of mainframes.



MV 1593

Figure 3-1. Top-of-Stack and Stack Bounds Registers

Figure 2.6: B6900 Top-of-Stack and Stack Bounds Registers

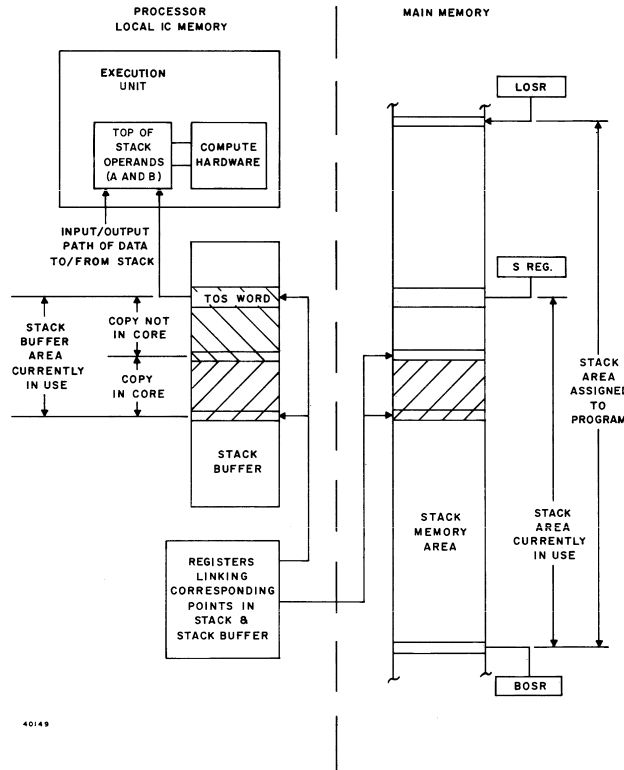


Figure III-1-12. Stack Buffer and Stack Memory Area

3-39

Figure 2.7: B7700 Stack Buffer and Stack Memory Area

## 2.2.4 International Computers Ltd. ICL2900 series

The ICL2900 series, introduced in 1974, was fairly similar to the Burroughs computers save for a lack of tagged memory and a different approach to the use of the stack. Its design is derived from the Manchester MU5 [IC78]. The ICL computers were accumulator-based with a stack that was explicitly referenced by the programmer. Figure 2.8 shows a comparison between the stacks of the B6700 and the ICL2900 [Dor75a]. The top three stack elements were buffered in registers [Dor75b, Chu75].

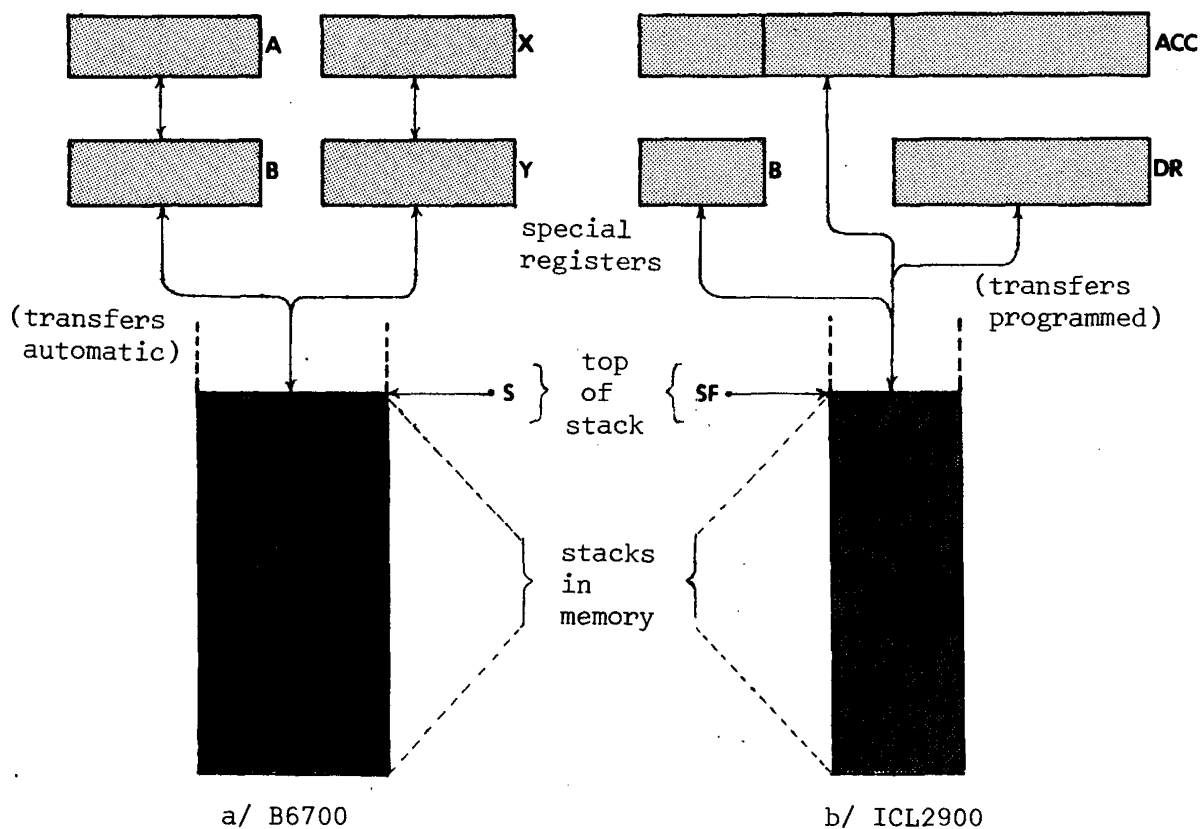
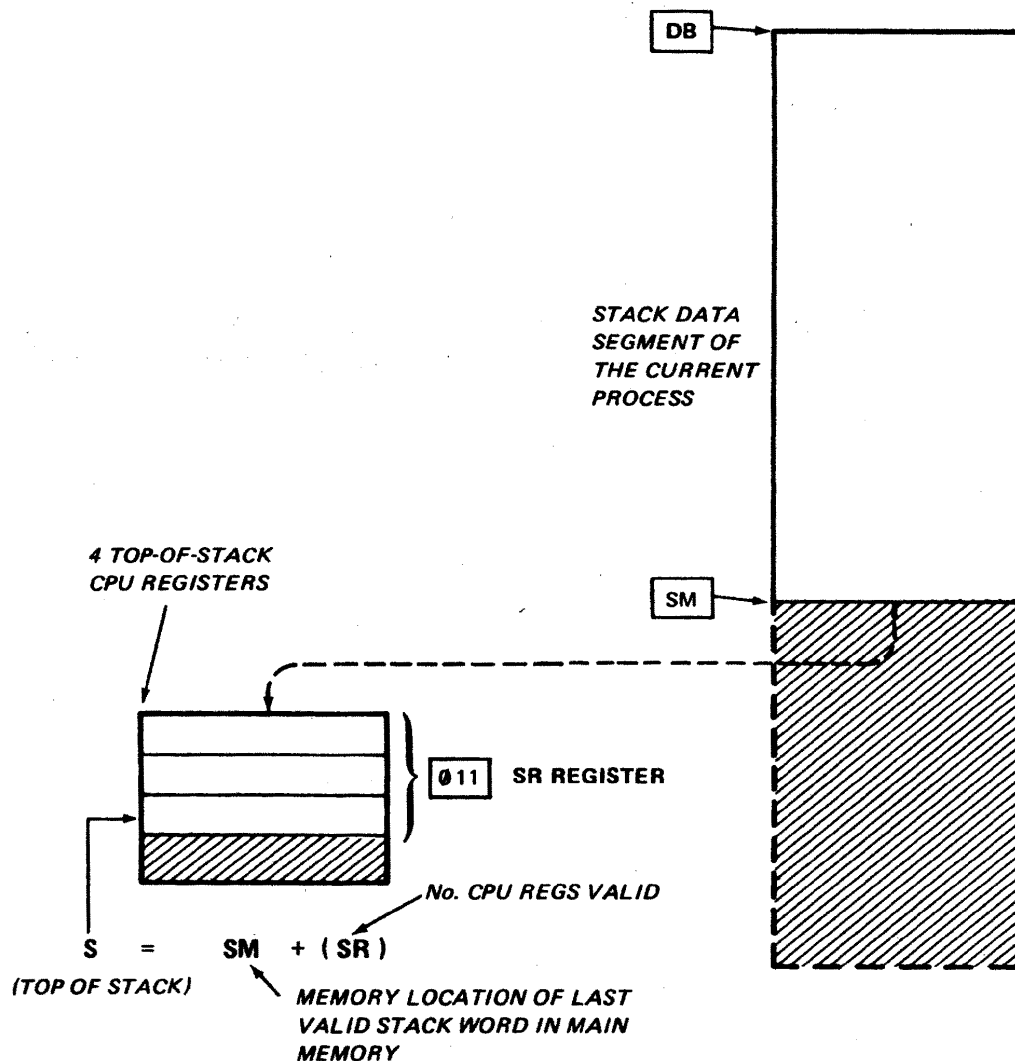


fig.3 Arithmetic stack mechanisms

Figure 2.8: Comparison of B6700 and ICL 2900 stack mechanisms

## 2.2.5 Hewlett-Packard HP3000

The HP 3000 series was originally introduced in 1972. It is similar to the Burroughs computers, but has been simplified to support real-time response [McK80, Sto80]. Figure 2.9 shows the structure of its stack. The main difference lies in the use of the four-element circular stack buffer. Unlike the B7700, the current top two elements of the stack buffer feed the ALU directly and the S register points to the current head of the buffer instead of main memory. Like the Burroughs computers, the buffer is managed automatically and flushed on subroutine calls [Bla77]. In later models (Series 68), the stack buffer was expanded to eight elements [Hew84, pg.86]. This series of computers was being sold by Hewlett-Packard, under the name 'e3000', up until November 2001.



**Figure 7. Stack registers extend the stack in memory.**

Figure 2.9: HP3000 Stack Registers

## 2.3 Shortcomings and Disappearance of the First Generation

During their heyday of about 20 years, stack computers were quite possibly the most sophisticated general-purpose computers available. But in retrospect, they had several glaring shortcomings which were endemic in machines of the time.

### 2.3.1 Explicit High-Level Language Support

The idea of directly supporting a high-level language in hardware seems downright baroque today. In hindsight however, there were some constraints then that have since vanished:

- Compilers were primitive, and took up a lot of the available memory.
- The machines were slow, leading to long compilation times, only to end up with sub-optimal code!
- Since code was written mostly by hand, and programs were getting larger and harder to write (including compilers), supporting a high-level language helped the programmer.

These led to two major features: hardware support for the execution models of structured languages such as ALGOL, and the integration of complex functions in the instruction set, implemented as microcode, making it easier to program the computer directly.

These features became weaknesses over time. A computer designed to execute one language well would perform poorly with another [Org73, ch.8]. As compilers improved they generated simpler subroutine linkages that did not match the full-featured built-in ones [HP02, 2.14]. The compilers also could not use the complex instructions provided. Finally, the microcode for these computers had itself grown to the point of unmanageability [Pat85] (reprinted [Pat86, FL86]).

Eventually, compilers became able to effectively reduce high-level languages features into series of simple operations, and the RISC computers that followed were designed in that light.

### 2.3.2 The Rise of RISC

The first generation of stack computers began to fade away in the early 1980's with the advent of the Reduced Instruction Set Computer (RISC) designs. The combination of advances in compilers, hardware speed, and integration forced a revision of the approaches used to improve the performance and reduce the costs of the hardware and the software. The combined end-results flatly outperformed the first generation of stack computers while also efficiently supporting high-level languages.

- Ditzel and Patterson criticized the original arguments for High-Level Language Computer Systems (HLLCS) [DP80] (reprinted [DP86, FL86], and [DP98b] with updated comments [DP98a]), and conclude that “. . . *almost any system can be a HLLCS through the appropriate software. . .*”. They also wrote an overview of the arguments for reduced instruction sets [PD80].
- Patterson later wrote an extremely broad article on the features and successes of the early RISC experiments, including software measurements and compiler techniques [Pat85] (reprinted [Pat86, FL86]), and advocates taking implementation as a factor in computer architecture.
- At Berkeley, Patterson and Sequin headed the RISC I and RISC II projects as one approach to RISC designs [PS81] (reprinted [PS98a] with updated comments [PS98b]). The fine details of their implementation were presented in the PhD thesis of one of their students, Manolis Katevenis [Kat85].
- One of the premises of RISC design is that the hardware and the software must be considered together. Hennessy and Jouppi measured the low-level features of software and proposed some architectural guidelines to support them without the pitfalls of past high-level language support. These included “. . . *the use load/store architecture and the absence of condition codes. . .*”. [HJB<sup>+</sup>82]. These data guided the Stanford MIPS project [HJP<sup>+</sup>82].
- The major technological change of the time was the emergence of Very Large Scale Integrated (VLSI) circuits which made possible the implementation of an entire processor on a single chip. The various approaches to integrated architecture are discussed by Hennessy [Hen84] (reprinted [Hen86, FL86]).

### 2.3.3 Excessive Memory Traffic

Without an optimizing compiler, a requirement for the explicit support of a structured, high-level language was the use of an execution stack in main memory instead of registers in the processor. This increased traffic to main memory, which was much slower, and further drove the development of complex microcoded instructions to avoid accessing it.

For example, passing parameters to a subroutine required copying values from a location in the caller’s stack frame to one in the callee’s frame, necessitating a memory read and a write for each parameter. Local variables and temporary values were also on the stack since there was no location in the processor to store them, further increasing memory traffic. In the case of the Burroughs computers, the use of a single stack meant that subroutine parameters were buried under the return address and other subroutine linkage information. This meant that they could not be used directly for computation without explicit loads into the top of the stack.

In later stack computers, some registers were used to buffer the top of the stack, but their small number (four or less) limited their usefulness to holding intermediate results of algebraic expressions. The subroutine linkage conventions required the registers to be flushed whenever a subroutine was called and so they could not be used to pass parameters. The Burroughs

B7700 was likely the only first-generation stack computer to address this problem by including a genuine 32-entry buffer (Figure 2.7) for the top of the stack [Bur73, pg.3-36].

### 2.3.4 The Need for Index Registers

Stack computers execute iterative code poorly compared to general-purpose register computers. Random-access registers can hold the loop indices and intermediate results for immediate access. On the other hand, a stack computer must temporarily move values off the top of the stack to access any index or result that isn't the most immediate. It is the source of enormous overhead whether or not this generates memory traffic and special-purpose index registers have to be used to reduce it. All first-generation stack computers included some form of index register:

- The KDF9 was the first to do so by including the Q-Stores [Hal62] (Figure 2.4). They were abundant (16) and could be also used as general-purpose registers.
- The B5000 series encoded loop counts in special instructions, such as BEGIN LOOP (BLP), END LOOP (ELP), and JUMP OUT LOOP CONDITIONAL (JLC), which reused some internal registers to hold addresses while in Character Mode<sup>4</sup> [Bur63] [Bur67].
- The B7700 added a vector mode of operations in which the index for one loop and the addresses and increments for up to three arrays were stored in separate internal registers so as to free the stack for computations [Bur73, pg.3-112].
- One of the three top-of-stack registers of the ICL 2900 could be used as an index register [Dor75a].
- The HP 3000 series had a single index register (X) to support loops [Hew84].

---

<sup>4</sup>Character Mode processed 6-bit Binary-Coded Decimal numbers, while Word Mode processed binary 48-bit numbers.

## Chapter 3

# History of the Second Generation of Stack Computers

In this chapter, I present the second generation of stack computers in the context of the pioneers of the field and of the machines that followed their insights. At the same time that the first generation of stack computers was fading away in the light of RISC, the second generation began to emerge and found a niche in embedded control systems instead of general-purpose computing.

### 3.1 Charles H. Moore and the Second Generation

The latest wave of second-generation stack computers was almost entirely initiated by Charles H. Moore and documented by Philip J. Koopman, Jr., with some additional unpublished material made available online by Jeff Fox [Fox04].

#### 3.1.1 Charles Havice (Chuck) Moore II

Chuck Moore studied physics at MIT (BS, 1960) and mathematics at Princeton. He became a freelance programmer during the 1960s and the software toolkit he created for himself gradually evolved into the Forth programming language [ML70]<sup>1</sup>. Along with Elizabeth Rather and Ned Conklin, he co-founded Forth Inc. in 1973 [RCM93] [RCM96]. In 1981, Moore began to pursue hardware implementations of the Forth virtual machine. This work was the basis for the second generation of stack computers and continues to this day.

##### 3.1.1.1 The Forth Programming Language Basis of Second-Generation Stack Computers

Much as stack computers from the first generation were based on ALGOL, those from the second generation are derived from the Forth programming language. Surprisingly, there seems to be no historical connection at all between the design of Forth and the early work of Charles Hamblin (Section 2.1.4) or the design of the KDF9 computer (Section 2.2.2). However, the

---

<sup>1</sup>Online as of March 2007 at [http://www.ultratechnology.com/4th\\_1970.pdf](http://www.ultratechnology.com/4th_1970.pdf) and [/4th\\_1970.html](http://www.ultratechnology.com/4th_1970.html)

Burroughs B5500 computer was the influence for the use of a stack for expression evaluation [Moo91]. The best introduction to Forth and the methodologies it favours are a pair of books by Leo Brodie [BI86] [Bro84].

A Forth system is divided into two interpreters. The outer interpreter receives source input and looks up each word in a dictionary. If found, it calls the inner interpreter to process the word's definition. In the most basic case a word definition is a series of addresses of other words, themselves defined in the same manner, ending with primitives which are written in machine code<sup>2</sup>. The inner interpreter is a small virtual machine which walks through these definitions and executes the primitives it encounters. The inner interpreter keeps track of the nesting of these definitions on a stack in memory, commonly referred to as the Return Stack.

The Forth primitives do their operations on another such stack, the Data Stack, where they take their arguments and place their results. The primitives are thus simple function applications which can be composed by executing them in sequence. Higher-level words are functional compositions of these primitives. These new words interact with the stack and compose in the same manner as the primitives.

A second-generation stack computer is basically a physical realization of the inner interpreter, the Forth primitives, and the stacks. The primitives become the instruction set which operates on a hardware Data Stack. The inner interpreter reduces to simple call and return instructions which use a Return Stack to store the return addresses of subroutines.

### **3.1.2 Philip J. Koopman, Jr.**

From 1986 to about 1995, Philip J. Koopman, Jr. did the most well-know applied and theoretical research on stack computers while at WISC Technologies, Harris Semiconductor (now Intersil), the United Technologies Research Centre, and Carnegie Mellon University (where he is now part of the faculty). His book on stack computers is still the single best reference on the subject [Koo89]. His paper on modern stack computer architecture [Koo90] contains the essential insights and comparisons to the CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer) designs of the time. He also did the initial work on efficiently compiling the C language to such machines [Koo94]. He touches upon the problem of pipelining a stack computer in a set of slides [Koo91]. Finally, he co-authored some comparative performance studies [KKC92b] [KKC92a]. Although he has left stack computers as a research field, his academic work remains the most visible one known.

---

<sup>2</sup>This is known as 'indirect-threaded code'. There exists also direct-threaded, string-threaded, token-threaded, and subroutine-threaded versions, each with different size/speed trade-offs. Second-generation stack computers are subroutine-threaded systems.

## 3.2 The Second Generation of Stack Computers

I'm concentrating here on the computers primarily designed by Chuck Moore. There are many more machines than the ones listed here (see Koopman's book [Koo89, App.A]), but Moore's work was by far the most ground-breaking and influential. Much less was published about his machines than those of the first generation. Therefore, the descriptions here are mostly based on information found in Koopman's book, reference manuals, and unpublished documentation.

### 3.2.1 NOVIX NC4016

Formed in 1983, NOVIX produced the first prototypes of the NC4016 (initially called the NC4000) in 1985. The NC4016 was a 16-bit processor, designed by Chuck Moore, which ran the Forth programming language natively. It was a remarkably small device implemented in about 4000 gates, amounting to about 16000 transistors [Mur86] [Koo89, 4.4]. Figure 3.1a shows a block diagram [Koo89, fig.4.6].

Since the NC4016 was a hardware realization of the Forth programming language, it supported an expression evaluation stack and a subroutine linkage stack both separate from main memory and accessed via separate external buses. It also used an unencoded instruction format, similar to microcode, which allowed simultaneous control of the ALU, the stacks, and the memory. A clever compiler could combine two to five primitive Forth operations into a single instruction. In ideal conditions, the NC4016 could manipulate both stacks, fetch from main memory, execute an ALU operation, and perform a subroutine return all in the same cycle.

The NC4016 led to the NC6016, which was licenced to Harris Semiconductors in 1987 and renamed the RTX-2000 [RCM96]. NOVIX ceased operations by 1989.

### 3.2.2 Harris<sup>3</sup> RTX-2000

The RTX-2000 is derived from the NC4016. The the main changes include the addition of byte-swapped memory access, some counter/timers, an interrupt controller, and a hardware 16x16 multiplier. The stacks are now on-chip [Koo89, 4.5] and can be subdivided into smaller stacks to support fast task switching. The RTX-2000 has mainly seen application in aerospace systems. Versions of the processor manufactured in radiation-resistant ('rad-hard') processes [Int00] have flown (and are still flying) on several NASA missions [Fre98] [Fre01] [Ras03]. Figure 3.1b shows a block diagram [Koo89, fig.4.8].

### 3.2.3 Sh-BOOM (Patriot Scientific IGNITE I)

In 1988, Russell Fish proposed a new low-cost microprocessor targeted at embedded systems, the Sh-BOOM, which Chuck Moore designed. It contained a 32-bit dual-stack microprocessor which shared the single DRAM memory bus with a smaller dedicated processor for deterministic transfers to peripherals and for dynamic memory refresh. The stacks, for expression evaluation and subroutine linkage, were on-chip, about 16 cells deep each, and would spill/fill

---

<sup>3</sup>Now Intersil.

to/from memory as required. The implementation used about 9000 gates. Figure 3.2 shows a block diagram of the main processor [Sha02, fig.1].

Contrary to the NC4016 or the RTX-2000 the Sh-BOOM did not use an unencoded instruction format, but packed four 8-bit, Forth-like instructions into each memory word. This formed a simple instruction cache that allowed instructions to be executed while the next memory fetch was in progress. This also allowed very small loops to execute from within the cache without requiring an instruction fetch. Another interesting feature was the use of conditional SKIP instructions which, if the condition was met, would skip over the remainder of the instructions in the memory word. Conditional jumps and calls were implemented this way [GWS91].

The Sh-BOOM broke away from a pure stack architecture by including 16 general-purpose registers (g0-g15), and by making most of the on-chip return stack addressable as registers (r0, r1, etc. . . ). The general-purpose registers were used for temporary storage and for I/O operations, much like the KDF9. To support the stack frames of ALGOL-like languages, instead of simply pushing values on the return stack, a number of empty locations could be allocated in one step and then later filled from the general-purpose registers.

The Sh-BOOM design is currently being marketed by Patriot Scientific<sup>4</sup> as the IGNITE I [Sha02] (previously PSC1000 [Sha99]) processor core, targeted at embedded Java applications. It is the most sophisticated second-generation stack computer currently available.

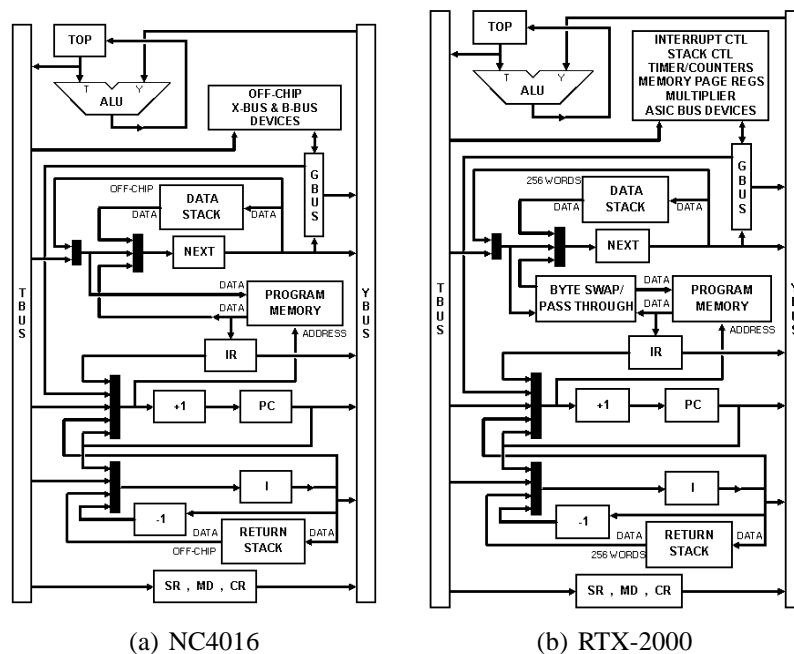


Figure 3.1: NC4016 and RTX-2000 Block Diagrams

<sup>4</sup><http://www.ptsc.com/>



### 3.2.5 F21

Jeff Fox had formed UltraTechnology in 1990 in Berkeley, to develop a custom computer in collaboration with Chuck Moore. The result was the F21 microprocessor [Fox98], which was an extension of the MuP21. The instruction set added post-incrementing loads and stores from the address register and the top of the return stack, and some extra arithmetic operations. The stacks were expanded to about 17 cells each. Like the MuP21, the memory interface was 20-bits wide and values were stored internally as 21 bits.

Like the MuP21, the F21 contained a video coprocessor, and added similar coprocessors for analog I/O and serial networks. Some common routines were included in on-chip ROM.

In a 0.8u process, the F21 was implemented in about 15,000 transistors, and had a typical execution rate of 100 MIPS (peaking internally at 500 MIPS), depending on memory access time. Ultratechnology ceased to exist in 2002 with nothing formally published about the design and only some prototype chips made. The website for the company<sup>5</sup> contains some fairly detailed documentation. For a reconstruction of what the block-level design might have been like, see Figure 6.1.

### 3.2.6 c18

Around 2001, Moore took the F21 design in a new direction and produced the c18 [Moo01b, Moo01a]. Architecturally, it is virtually identical to the F21, but adds a second address register for on-chip communication. Its width was also reduced to 18 bits to match the fast cache memory chips available at the time. This leaves room to pack only 3 instructions per memory word.

The coprocessors were eliminated and replaced by a watchdog timer. There is no external memory bus. External memory must be accessed via the parallel I/O pins, and programs must be loaded into a few hundred words of on-chip RAM to be executed.

The c18 was simulated in a modern 1.8V 0.18u process. It had a predicted \*sustained\* execution rate of 2400 MIPS, while dissipating about 20 mW. It was an aggressive, fully asynchronous design.

The c18 was targeted at multiprocessing. A 5x5 array of c18's, connected by horizontal and vertical buses, would fit in 7mm<sup>2</sup>. This eventually became realized as the SEAforth-24 multiprocessor currently entering production at Intellasis<sup>6</sup> (Intelligent Array Systems).

## 3.3 Recent Research

Most of the research in the last decade has been outside of academia, and/or of little visibility due to the mistaken lumping of these second-generation designs with the previous generation. This section will overview the most salient academic, commercial, and amateur research on the subject.

- Between 1994 and 2004, Christopher Bailey co-authored a number of papers on various enhancements to stack computers for High-Level Language support, interrupt per-

---

<sup>5</sup>As of March 2007: <http://www.ultratechnology.com/f21.html>

<sup>6</sup><http://intellasis.net/>

formance, and instruction-level parallelism [Bai94] [BS94] [Bai00, DBL00] [Bai04] [SB04]. His 1996 PhD thesis presented an improvement to stack spill/fill algorithms so as to further reduce memory traffic [Bai96].

- During his Master's studies at the University of Alberta, Robert James Chapman explored the synthesis of stack computers using VHDL [Cha97] [Cha98] and wrote a paper which decomposed the usual stack permutation operations into smaller primitives [Cha95].
- Myron Plichota is a freelance consultant in Hamilton, Ontario who designed in 2001 the Steamer 16 processor<sup>7</sup> as a VHDL design implemented on a Cypress CY37128P84-125JC CPLD (Complex Programmable Logic Device). It is notable for fitting in very little hardware, having only eight, 3-bit instructions, and a bare minimum 3-deep on-chip stack. It runs Forth-like software with C-like stack frames in main memory. It is remarkable in its speed/size trade-off while still achieving 20 MIPS. It has been used in an industrial machine vision application.
- While at the Technical University of Munich, Bernd Paysan wrote his 1996 Diploma thesis on the 4stack processor<sup>8</sup> [Pay96], a 4-way superscalar VLIW (Very Long Instruction Word) design specified in Verilog. It is meant for embedded DSP (Digital Signal Processing) applications, but has a supervisor mode and virtual memory for desktop use. He also designed the much smaller b16 microprocessor<sup>9</sup> [Pay02], a 12-bit version of which is used internally at Mikron AG.
- Chung Kwong Yuen<sup>10</sup> at the National University of Singapore has an unpublished paper [Yue] on how to implement a reorder buffer to obtain superscalar execution in stack computers<sup>11</sup>.
- Chen-Hanson Ting currently runs the eForth Academy<sup>12</sup> in Taiwan, which provides design classes for embedded systems. He created the P series of microprocessors<sup>13</sup> derived from the works of Chuck Moore. They are described in [Tin97a, Tin97b]<sup>1415</sup>.
- A number of students at the Chinese University of Hong Kong designed and implemented two versions of a derivative of the MuP21 microprocessor : The MSL16 was first implemented on an FPGA (Field-Programmable Gate Array) [LTL98] and later re-implemented in silicon using asynchronous logic as the MSL16A [TCCLL99].

---

<sup>7</sup>The only documentation was at <http://www.stringtuner.com/myron.plichota/steamer1.htm> which is now defunct, but archived in the Internet Archive Wayback Machine at <http://www.archive.org/web/web.php>

<sup>8</sup><http://www.jwtd.com/~paysan/4stack.html>

<sup>9</sup><http://www.jwtd.com/~paysan/b16.html>

<sup>10</sup><http://www.comp.nus.edu.sg/~yuenck/>

<sup>11</sup>One of two papers available at <http://www.comp.nus.edu.sg/~yuenck/stack>

<sup>12</sup><http://www.eforth.com.tw/>

<sup>13</sup><http://www.eforth.com.tw/academy-n/Chips.htm>

<sup>14</sup>Publication list: [http://www.eforth.com.tw/academy-n/Bookstore/bookstore\\_4.htm](http://www.eforth.com.tw/academy-n/Bookstore/bookstore_4.htm)

<sup>15</sup>Published by Offete Enterprises: <http://www.ultratechnology.com/offete.html>

## 3.4 Strengths and Weaknesses of the Second Generation

The second generation of stack computers still has some of the drawbacks of the first: a need for index registers, stack manipulation overhead, and it additionally supports ALGOL-like languages poorly. However, the second generation also has some distinct advantages: reduced memory bandwidth and system complexity, and faster subroutine linkage and interrupt response.

### 3.4.1 The Need for Index Registers

As in computers from the first generation (Section 2.3.4), the access of loop indices or intermediate results which are not immediately on top of the stack requires significant stack manipulation overhead. All second-generation stack computers, except the very smallest, mitigate this problem with index registers:

- The NC4016 buffered the topmost Return Stack element on-chip so it could be used as an index register (I). A loop-on-index instruction would decrement I, and then conditionally jump to the beginning of the loop. The RTX-2000 used the same mechanism.
- The Sh-Boom used a count (ct) and an index (x) register. The count register was used by decrement-and-branch-on-non-zero instructions, and the index register was used for direct, post-incrementing, and pre-decrementing memory accesses.
- The MuP21 used the A register to hold memory addresses which could then be moved to the data stack, modified, and the returned to A for the next access. The alternative would have needed a deeper stack and more stack manipulation opcodes.
- The F21 could do post-incrementing memory accesses from A register and from the top of the Return Stack. They were primarily meant for fast memory-to-memory transfers. The c18 has the same mechanism.

### 3.4.2 Stack Manipulation Overhead

Stack computers from the first generation suffered from excessive memory traffic (Section 2.3.3) since their stacks, except for a few working registers, were entirely in main memory. However, this had the advantage of allowing random access to the entire stack.

Computers from the second generation keep their stacks separate from main memory and usually on-chip. This has the advantage of causing no memory traffic, but typically limits access to the topmost two to four elements since the stack is no longer addressable. This limitation requires that the topmost elements be permuted to bring a value to the top of the stack (Section 7.3.3).

The original overhead of memory traffic is thus transformed into the overhead of extra operations to manipulate the stack. This problem can be mitigated by using more deeply-accessible stacks or more complex stack manipulation operations. This random access comes at the price of increased system complexity, culminating in a conventional multiparty register file.

### **3.4.3 Poor Support of ALGOL-like Languages**

Languages which are derived from ALGOL, such as C, use the stack as a means of allocating space for, amongst others, the local variables of a procedure. This entails pushing entire structures onto the stack and then accessing them randomly, often through a pointer. Thus a local variable must be present in main memory since the stack in a second-generation computer is not addressable.

Solutions to this problem include more sophisticated compilers [Koo94] [ME97] [ME98], the addition of some form of frame pointer register which can support indexed addressing, or making the return stack addressable such as in the PSC1000/IGNITE I [Sha02] [Sha99].

### **3.4.4 Reduced Instruction Memory Bandwidth and System Complexity**

The compact encoding of stack instructions stems from the implicit nature of their operands: It is always the top of the stack. Thus, contrary to a register machine, several such operations can fit into a single memory word. This correspondingly reduces the frequency of instruction fetches. The memory access cycles between instruction fetches can then be used for loads, stores, and fetches, eliminating the need for separate instruction and data memory buses (Section 7.2.4). This greatly reduces system complexity and allows for a higher internal operating frequency than the memory would normally allow.

The case of unencoded instructions, as in the NC4016 and the RTX-2000, does not provide the same memory bandwidth advantage, but while still operating on a single memory bus, increases the performance through the multiple simultaneous operations that can be contained in such an instruction word.

### **3.4.5 Fast Subroutine Linkage and Interrupt Response**

A stack computer does not need to save the contents of registers upon entering a subroutine. Its parameters are already on top of the data stack, which become its working values throughout the computation, and ultimately remain as one or more return values upon exiting the subroutine. The call and return instructions automatically use the return stack for the return address. Subroutine linkage thus requires no additional memory traffic and takes only a cycle or two (Section 7.3.6).

An interrupt service routine (ISR) is effectively a hardware-invoked subroutine, and so benefits from the fast linkage. The ISR can simply do its work on top of the stacks so long as it leaves them unaltered upon exit.



**Part II**

**Qualitative Arguments**



## Chapter 4

# Distinguishing the First and Second Generations

The existing computer architecture literature considers all stack computers to be of the same kind<sup>1</sup>. This view seems correct when contrasted against modern register-based machines. However, it conflates the first and second generations of stack computers, which makes difficult a clear discussion of their respective properties. Distinguishing the generations is important since the second resembles current register-based designs much more than it does the first. Without this distinction, an entire branch of computer architecture is missed solely due to a categorical error.

The differences between the generations stem from the two historical approaches to stack computation. The first generation is based on Bauer’s “stack principle” for subroutine storage allocation (Section 2.1.3) and is exemplified by the Burroughs B5000 series (Section 2.2.3). The second generation originates in Hamblin’s method for evaluating and composing expressions (Section 2.1.4), first seen in the English Electric KDF9 (Section 2.2.2), and later independently rediscovered as the Forth programming language (Section 3.1.1.1).

The only significant exception to this conflation I’ve found is a section in Feldman and Retter’s text [FR93, pp.599-604] which lists some of the same “first generation stack machines” as I do in Section 2.2, then proceeds to explain in a nutshell the origin and features of the second generation of stack computers, although they do not refer to them by that name.

In this chapter, I expand on Feldman and Retter’s statements and propose a codification based on some properties of the stacks: their location, purpose, and the operations done upon them.

---

<sup>1</sup>Except for a passing mention in the preface of Koopman’s book [Koo89] and a short summary in Bailey’s PhD Thesis [Bai96].

## 4.1 Location of Stacks: In-Memory vs. In-Processor

“[...] It is this explicit coupling of all ALU operations to a hardware stack which sets these machines apart.”

[FR93, pg.600]

The first distinguishing feature is the location of the stacks.

First-generation stack computers (Figure 4.1) kept their stacks as randomly accessible data structures in main memory, located by an automatically managed stack pointer register. Each operation implicitly loaded and stored the required data for expression evaluation to an internal stack of two to four registers. The number and size of the stacks in memory were variable and usually each process had its own. Later machines used circular buffers between the registers and the memory to accelerate access to items recently put on the stack. Unfortunately, first-generation stack computers were overtaken by register-based machines before this feature became widespread.

General-purpose register computers (Figure 4.2) use the same kinds of stacks, also kept in memory. However, the stack pointer is now an ordinary register, selected by convention, and is manually controlled by software. Software manually loads and stores values from the stack into an arbitrarily accessible register file. Register machines usually place a cache buffer between the registers and memory.

Contrary to both first-generation and general-purpose register computers, second-generation machines (Figure 4.3) keep their stacks in the processor. The stacks are not randomly accessible data structures and only make visible the top two to four elements. The stacks are separate from main memory and only interact with it via load, store, and flow control instructions. The number and size of the stacks are fixed, although they may spill to memory via a pointer, depending on the size of the system, and thus behave as their own caches. There are typically only two stacks, shared by all processes, which can internally exchange data amongst themselves.

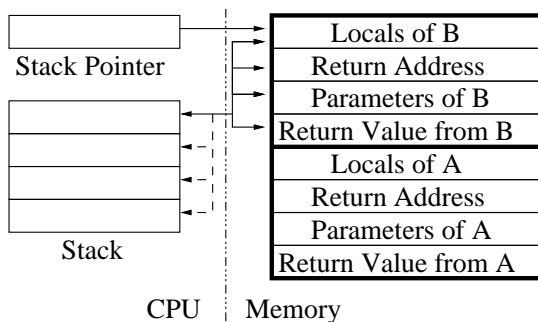


Figure 4.1: First-Generation Stack Computer Block Diagram

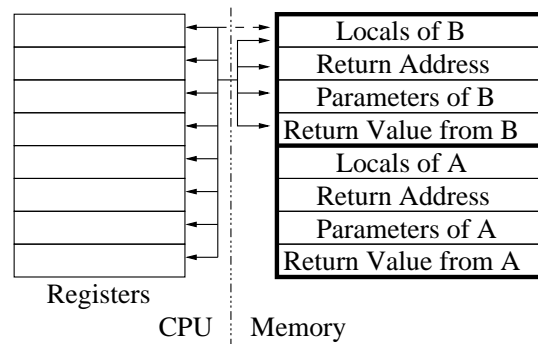


Figure 4.2: General-Purpose Register Computer Block Diagram

## 4.2 Use of Stacks: Procedure Nesting vs. Expression Evaluation

“[...] While recursion is easy to accomplish with a stack in memory, this is not what we mean by *stack machine*. [...]”

“[...] Unlike most earlier stack machines, these Forth processors have two stacks, one to hold temporary data and pass subroutine parameters, and the other to save subroutine return addresses and loop counters. [...]”

[FR93, pg.600]

The second distinguishing feature is the use of the stacks.

First-generation stack computers (Figure 4.1) used stacks as structured temporary storage for program procedures. Each procedure invocation would automatically cause the allocation of an amount of space on the stack to contain (typically) the parameters of the procedure, its return value, its local variables, and the return address of its caller. This area is referred-to as a procedure activation record. Values from the record were loaded and stored into the small internal stack as needed for expression evaluation. The internal stack only held the intermediate results of computations within a given procedure. All linkage between procedures was done solely on the stack in memory.

General-purpose register computers (Figure 4.2) use the same kind of stacks, in the same manner, except that the procedure activation records are manually managed by software and procedure linkage can occur through the registers if the parameters, locals, and return values fit within them.

Second-generation stack computers (Figure 4.3) use separate stacks to control procedure nesting and to perform expression evaluation. The return addresses of procedure calls are stored on a stack dedicated to that purpose (Return Stack). Storing them separately helps to eliminate the division of the other stack (Data Stack) into procedure activation records. Thus, a called procedure finds its parameters (P) on top of the Data Stack, manipulates them directly as locals (L) during expression evaluation, and leaves on top either its return value (R) upon exit or the parameters for calling another procedure. The Data Stack is used for an effectively single, large, and complex expression evaluation whose parts are tracked by the contents of the Return Stack.

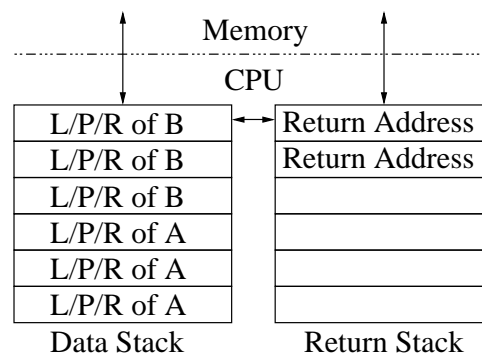


Figure 4.3: Second-Generation Stack Computer Block Diagram

### 4.3 Operations with Stacks: High-Level Language Support vs. Primitive Operations

“First generation stack machines, such as the Burroughs B5000, B5500, B6700, B7700, Hewlett-Packard HP3000, and ICL2900, were designed to execute languages like Algol-60 [Dor75b]. Most of these had a single stack which was used for holding temporary data, passing subroutine parameters, and saving subroutine addresses. [...]”

[FR93, pg.600]

The third distinguishing feature is the operations performed upon the stacks.

First-generation stack computers (Figure 4.1) had built-in hardware and microcode support for high-level language (typically ALGOL) and operating system features. The procedure call and return instructions would automatically allocate and deallocate activation records on the stack. The arithmetic instructions would determine the data type of their operands from special tags. Indirect reference words and data descriptors would allow for lexical scoping across activation records and resolve call-by-name references. Multiple stacks could be maintained and cross-referenced to enable multitasking and inter-process communication. Descriptors could point to data on disk to support code and data paging-on-demand. The end result was powerful, but extremely complex. These features are well-described in Organick’s book [Org73].

General-purpose register computers (Figure 4.2) have none of these language-specific features, although they were designed with the efficient support of ALGOL-like languages in mind. A large register file supports fast, simple procedure linkage when possible, and can hold multiple intermediate values during expression evaluation. The instruction set is simple but can use any registers as source and target. High-level language and system features are managed in software by the compiler and the operating system.

Second-generation stack computers (Figure 4.3) have more in common with general-purpose register machines than with first-generation stack computers. The call and return instructions only save and restore the calling procedure’s return address, leaving more complex procedure linkage to software. Arithmetic instructions operate solely on the top few elements of an internal stack and their operands must be loaded from and stored to memory explicitly. Only simple direct and indirect addressing modes are supported, although post-incrementing/decrementing versions are common. Other than the implicit use of stacks for basic procedure linkage and expression evaluation, all high-level language and operating system features are implemented in software in the same manner as in register-based machines.

# Chapter 5

## Objections Cited by Hennessy & Patterson

Hennessy and Patterson’s magnum opus “*Computer Architecture: A Quantitative Approach*” [PH90, HP96, HP02] has a tremendous influence on the field of computer architecture. I support this claim by comparing the number of citations it has received compared to other textbooks on computer architecture.

Most arguments against stack computer architecture are drawn from a few statements found in this book. I present counterarguments which support the claim that the statements are valid for the first generation of stack computers, but not the second.

### 5.1 The Enormous Influence of Hennessy & Patterson on Computer Architecture

Some data collected in Fall 2004 provides a view of this excellent book’s status (Table 5.1) based on the number of citations it has received compared with that of other books on computer architecture. These books were all found in the University of Waterloo Davis Centre Library. The number of citations was obtained from the ACM Digital Portal’s Guide To Computing Literature<sup>1</sup>.

It’s easy to see that the influence of the first two editions of Hennessy & Patterson’s work completely dwarfs that of the remainder of the sample. As of January 2006, the updated cumulative number of citations provided by the ACM Guide are:

- First Edition [PH90]: 424
- Second Edition [HP96]: 454
- Third Edition [HP02]: 125

Additionally, the CiteSeer<sup>2</sup> scientific literature library show a total of 1525 citations for all editions combined, as of January 2006.

---

<sup>1</sup><http://portal.acm.org/>

<sup>2</sup><http://citeseer.ist.psu.edu/>

Incidentally, the main text on second-generation stack computers [Koo89] had, according to the ACM Guide, seven citations in the Fall of 2004 and twelve as of January 2006. It is not listed in CiteSeer.

Book	Citations	Book	Citations
[PH90]	324	[FR93]	1
[HP96]	310	[MK97]	1
[HP02]	19	[HVZ95]	0
[PH98]	21	[Sta02]	0
[Hwa92]	50	[GL03]	0
[Kog90]	24	[Omo99]	0
[Omo94]	6	[Bur98]	0
[SSK97]	6	[Sto92]	0
[Sta93]	2	[Wil91]	0
[Wil01]	1	[Sta90]	0
[Hay97]	1	[Mur90]	0
[Wil96]	1	[Bla90]	0
[MP95]	1		

Table 5.1: Comparison of Citations of Computer Architecture Texts (as of Fall 2004)

## 5.2 The Disappearance of Stack Computers (of the First Generation)

One of the views expressed by Hennessy & Patterson is that “*stack based machines fell out of favor in the late 1970s and, except for the Intel 80x86 floating-point architecture, essentially disappeared.*” [HP96, pg. 113][HP02, pg. 149]. This statement can only refer to the first generation of stack computers since the second generation did not really begin until 1985 (Section 3.2.1), “*roughly concurrent with the emergence of RISC as a design philosophy*” [FR93, pg. 600].

These new machines went unnoticed due to being in the niches of embedded real-time control and aerospace applications (Section 3.2.2) instead of general-purpose computing. The latter instances of the second generation were developed in the 1990s, with little visibility in academia (Section 3.3), since by then the term ‘stack computer’ had become synonymous with designs from the first generation.

## 5.3 Expression Evaluation on a Stack<sup>3</sup>

Hennessy & Patterson state:

Although most early machines used stack or accumulator-style architectures, virtually every machine designed after 1980 uses a load-store register architecture. The major reason for the emergence of general-purpose register (GPR) machines are twofold. First, registers—like other forms of storage internal to the CPU—are faster than memory. Second, registers are easier for a compiler to use and can be used more effectively than other forms of internal storage. For example, on a register machine the expression  $(A * B) - (C * D) - (E * F)$  may be evaluated by doing the multiplications in any order, which may be more efficient because of the location of the operands or because of pipelining concerns (see Chapter 3). But on a stack machine the expression must be evaluated left to right, unless special operations or swaps of stack positions are done. [HP96, pg. 71]

The third edition has a different final sentence:

Nevertheless, on a stack computer the hardware must evaluate the expression in only one order, since operands are hidden on the stack, and it may have to load an operand multiple times. [HP02, pg. 93]

The first point, which implies that the stack is in memory, is no longer valid. Second-generation stack computers keep their stacks internal to the CPU (Section 4.1). Furthermore, the access to a stack is faster than to registers since no addressing is required. The inputs to the ALU are hardwired to the top two elements of the stack. This is put to advantageous use when pipelining (Section 7.4).

Secondly, the claim that compilers can use registers more effectively is true only because much research has been done on register allocation in modern compilers. Prior to the advent of modern compiler techniques like graph colouring, registers were seen as difficult to use and stacks were favoured. For example, this is the reason the SPARC architecture uses register windows which effectively form a stack of activation records [PS98a, pg. 2]. There has been promising work showing that it is possible to cache virtually all local variable memory references onto the stack [Koo94]<sup>4</sup>[ME97]<sup>5</sup>[ME98]<sup>6</sup>. Also, it could be possible to evaluate expressions in an out-of-order fashion on a stack computer (Section 3.3).

Lastly, the final point raised is true. Operands are hidden on the stack and even with the aforementioned compiler techniques this fact makes for poor performance on iterative code due to the stack manipulations required (Section 7.3.3). However, a register-based computer has the same kind of repeated memory accesses and register-to-register copying overhead when

---

<sup>3</sup>The arguments in this section suggest an interesting way of thinking qualitatively about stacks: that they are the 'reciprocal' (or the 'inverse') of registers. For example, reading a register does not destroy its contents, but writing does. Conversely, reading from a stack pops information from it, but writing to it simply pushes the existing information down. Up to its maximum capacity, no information is lost. This is a tantalizing symmetry.

<sup>4</sup>[http://www.ece.cmu.edu/~koopman/stack\\_compiler/index.html](http://www.ece.cmu.edu/~koopman/stack_compiler/index.html)

<sup>5</sup><http://www.complang.tuwien.ac.at/papers/maierhofer%26ertl97.ps.gz>

<sup>6</sup><http://www.complang.tuwien.ac.at/papers/maierhofer%26ertl98.ps.gz>

entering or exiting a subroutine (Section 7.3.6). This overhead is virtually nil in stack computers. Finally, there are hints that a redesigned stack computer instruction set could combine stack manipulations with arithmetic operations 'for free', without adding any new datapath or control lines (Sections 9.2.2 and 9.2.3).

## 5.4 The Use of the Stack for Holding Variables

Immediately after the previous quote they state:

More importantly, registers can be used to hold variables. When variables are allocated to registers, the memory traffic is reduced, the program is sped up (since registers are faster than memory), and the code density improves (since a register can be named with fewer bits than a memory location). [HP96, pg. 71][HP02, pg. 93]

As stated in the previous section, a stack can also be used to hold variables. Since second-generation stack computers keep their stack in the CPU, the memory traffic is reduced (Section 7.2.3) and the program sped up in the same proportion. The code density is improved to an even greater extent since a stack does not need to be named (Section 7.2.4). No addressing bits are required since operations implicitly use the top of the stack. Overall, these features are no different than in register-based computers.

## 5.5 Distorted Historical Arguments

Lastly, Hennessy & Patterson raise points from past research against the use of (first-generation) stacks:

The authors of both the original IBM 360 paper [ABB64] and the original PDP-11 paper [BCM<sup>+</sup>70] argue against the stack organization. They cite three major points in their arguments against stacks:

1. Performance is derived from fast registers, not the way they are used.
2. The stack organization is too limiting and requires many swap and copy operations.
3. The stack has a bottom, and when placed in slower memory there is a performance loss.

[HP96, pg. 113][HP02, pg. 149]

At first glance, these points are correct when referring to first-generation stack computers: an abundance of fast registers which can be randomly and non-destructively accessed will increase performance by reducing memory traffic and re-ordering operations. For second-generation stack computers however, these points are moot:

1. On-chip stacks are really a linear collection of registers.

2. The limitations are partially a matter of compiler technology (Section 5.3) and on the other hand, stacks avoid the subroutine call overhead of register-based computers (Section 7.3.6).
3. This is a straw-man: All stacks have a bottom. Past experiments have shown that an on-chip stack buffer that is 16 to 32 elements deep eliminates virtually all in-memory stack accesses for both expression evaluation and subroutine nesting and that the number of accesses to memory decreases *exponentially* for a linear increase in hardware stack depth [Koo89, 6.4] [Bai96, 4.2.1] (Appendices B.2.6 and B.2.7). Current Intel processors use exactly such a stack, 16 elements deep, to cache return addresses on-chip [Int06, 2.1.2.1, 3.4.1.4], as does the Alpha AXP 21064 processor [McL93]. Hennessy & Patterson themselves show data supporting this feature [HP96, pg. 277] [HP02, pg. 214]. The SPARC architecture accomplishes a similar results with its register windows.

The preceding quote is however an abridged version. The original statements by Bell *et al.* were:

The System/360 designers also claim that a stack organized machine such as the English Electric KDF 9 (Allmark and Lucking, 1962) or the Burroughs B5000 (Lonergan and King, 1961) has the following disadvantages:

1. Performance is derived from fast registers, not the way they are used.
2. Stack organization is too limiting, and requires many copy and swap operations.
3. The overall storage of general register and stack machines are the same, considering point #2.
4. The stack has a bottom, and when placed in slower memory there is a performance loss.
5. Subroutine transparency is not easily realized with one stack.
6. Variable length data is awkward with a stack.

We generally concur with points 1, 2, and 4. Point 5 is an erroneous conclusion, and point 6 is irrelevant (that is, general register machines have the same problem). [BCM<sup>+</sup>70]

Hennessy & Patterson are simply repeating the points which are supported by the authors of this quote. In retrospect, it is peculiar that the authors lump both the KDF9 and the B5000 together, despite being very different machines (see Sections 2.2.2, 2.2.3, and Chapter 4). The arguments should not apply equally to both. It turns out that the quote is an extremely abridged version of the original statements by the System/360 designers:

Serious consideration was given to a design based on a pushdown accumulator or stack. This plan was abandoned in favor of several registers, each explicitly addressed. Since the advantages of the pushdown organization are discussed in the literature, it suffices here to enumerate the disadvantages which prompted the decision to use an addressed-register organization:

1. The performance advantage of a pushdown stack organization is derived principally from the presence of several fast registers, not from the way they are used or specified.
2. The fraction of “surfacing” of data in the stack which are “profitable”, i.e., what was needed next, is about one-half in general use, because of the occurrence of repeated operands (both constants and common factors). This suggests the use of operations such as TOP and SWAP, which respectively copy submerged data to the active positions and assist in clearing submerged data when the information is not longer required.
3. With TOP’s and SWAP’s counted, the substantial instruction density gained by the widespread use of implicit addresses is about equalled by that of the same instructions with explicit, but truncated, addresses which specify only the fast registers.
4. In any practical implementation, the depth of the stack has a limit. The register housekeeping eliminated by the pushdown organization reappears as management of a finite-depth stack and as specification of locations of submerged data for TOP’s and SWAP’s. Further, when part of a full stack must be dumped to make room for new data, it is the *bottom* part, not the active part, which should be dumped.
5. Subroutine transparency, i.e., the ability to use a subroutine recursively, is one of the apparent advantages of the stack. However, the disadvantage is that the transparency does not materialize unless additional independent stacks are introduced for addressing purposes.
6. Fitting variable-length fields into a fixed-width stack is awkward.

In the final analysis, the stack organisation would have been about break-even for a system intended principally for scientific computing. Here the general-purpose objective weighed heavily in favor of the more flexible addressed-register organization. [ABB64]

I’ll address these points individually:

1. As previously mentioned, on-chip stacks are really a linear collection of registers. Additionally, the unabridged statement supports the use of stacks when either fully on-chip, as in second-generation stack computers (which the KDF9 was), or sufficiently buffered as in the B7700 (Sections 2.2.3 and 2.3.3) which was not introduced until 1973.
2. While stack permutations are inevitable, how often they are required is a strong function of how the code is arranged by the programmer or compiler. It also depends on whether the code is primarily iterative, recursive, or composed of nested procedures (Section 7.3.1). The choice of stack instruction set affects the overhead significantly (Section 9.2.2).
3. The static code analyses for a basic second-generation stack computer (Appendix B.1.5) support this statement, but also suggest that the instruction density can be increased much further (Chapter 8). This is also dependent on the nature and arrangement of the code.

4. Experiments done since show that a stack needs to be only 16 to 32 elements deep to virtually eliminate stack overflow to main memory [Koo89, 6.4.1] [Bai96, 4.2]. What memory traffic remains can be mitigated by the hardware stack management algorithms that have been discovered since [Koo89, 6.4.2] [Bai96, 6.2]. The abridged versions of this statement omitted the trade-off between the register housekeeping required for subroutine calls on register-based computers versus the stack housekeeping required for iterative code on stack-based computers.
5. I suspect the authors were thinking of the KDF9 since first-generation stack computers exhibit subroutine transparency using a single stack. This is possible because the activation record of an instance of a subroutine is stored in main memory. For second-generation stack computers like the KDF9, whose stack is not randomly accessible, a single stack is insufficient since at the very least the arguments being passed recursively would be buried under the return address (Section 4.2). Thus at the minimum a second stack is needed to hold return addresses.  
The statement is correct for both generations when index registers are used (Sections 2.3.4 and 3.4.1) since unless they are also stacks themselves, nested subroutines which use them would have to do housekeeping to preserve and access their values, effectively implementing in software the stack of a first-generation stack computer at a great performance penalty. Finally, given integrated circuits, the cost of an extra stack is minimal.
6. Bell *et al.* are correct when saying that this point is irrelevant. The problem, if present, is orthogonal to the internal organization of a fixed-width machine.

In summary, the arguments quoted by Hennessy & Patterson were oversimplified and referred to old hardware and compiler technology. In fact, the original source of the arguments is far less critical of stacks than the version Bell *et al.* present and most of the shortcomings have been overcome since then.



**Part III**

**Quantitative Arguments**



# Chapter 6

## A Stack-Based Counterpart to DLX: Gullwing

I present the Gullwing<sup>1</sup> processor architecture as an unpipelined, stack-based analogue of Hennessy and Patterson's DLX [HP02]: a small design that is a simplified reflection of the current state of the art. Gullwing is closely based on the available descriptions of the MuP21, F21, and c18 processors (Chapter 3).

### 6.1 Block Diagram

Figure 6.1 shows the datapath components of the processor. The description is not entirely abstract or free from implementation details. Some have been included for clarity and some are simply too compelling to ignore. Unless noted otherwise, the registers and memory are 32 bits wide and contain integers. The depth of the stacks is arbitrary. See Section 5.5 for a discussion of useful depths.

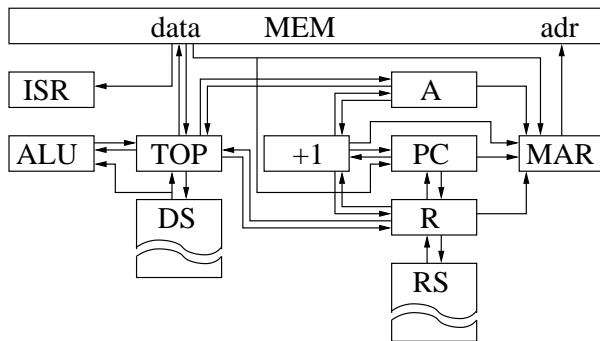


Figure 6.1: Gullwing Block-Level Datapath

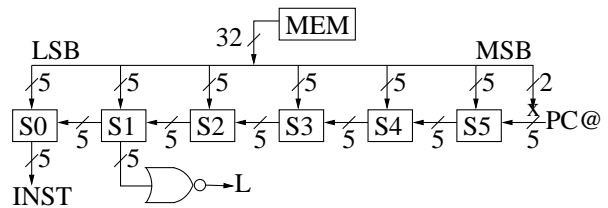


Figure 6.2: Gullwing Instruction Shift Register Block Diagram

<sup>1</sup>The name 'Gullwing' was inspired by Richard Bach's book, "Johnathan Livingston Seagull", and also by the DeLorean DMC-12 made famous in the "Back To The Future" movie trilogy by Robert Zemeckis and Bob Gale.

## 6.1.1 Memory Subsystem

At the top is the main memory (MEM), whose address port is continually driven by the Memory Address Register (MAR). The MAR is itself loaded by one of the Address Register (A), the Program Counter (PC), or the Return Register (R). The contents of each of these can be modified by a dedicated incremter (+1). The MAR can also be loaded directly from memory output in the case of jumps and calls, or from the incremented version of the PC in the case of subroutine returns.

The PC holds the address of the next memory word to be read for fetching instructions or literals. Upon a subroutine call, it is stored into R, which is itself saved onto the Return Stack (RS). The reverse process occurs when a subroutine returns.

The A register is used under program control to hold the addresses of loads and stores, with optional post-incrementing. The R register can be used in the same manner.

### 6.1.1.1 Single Memory Bus

In order to concentrate on the architecture of Gullwing, the main memory is abstracted away to an idealized form. It has a single, bidirectional data bus and a single, unidirectional address bus<sup>2</sup>. One read or one write is possible per cycle. By default, the output of the memory is always available for reading.

### 6.1.1.2 Differentiating Loads, Stores, and Fetches

If a read from memory simply reads the current memory output, it is termed a 'fetch'. There is no write equivalent to a fetch. If an explicit address is used to read or write to memory, it is termed a 'load' or a 'store', respectively. For example, in the next two sections it will be shown that calls and jumps fetch their target address, but load the instructions from that location.

## 6.1.2 Computation Subsystem

At the far left, the Arithmetic and Logic Unit (ALU) always takes its inputs from the Top Register (TOP) and the Data Stack (DS) and returns its output to TOP, whose original contents are optionally pushed onto the DS. The TOP register is the central point of gullwing. It is the source and target for memory operations and can exchange data with the A and R registers.

## 6.1.3 Control Subsystem

Figure 6.2 shows the details of the Instruction Shift Register (ISR). The 32 bits of an instruction word are loaded into six 5-bit registers (S0 to S5), each holding one instruction, with the two most significant bits (MSB) left unused<sup>3</sup>. The instruction in the least significant (LSB) position is output as the current one being executed (INST). When INST finishes executing, the contents

---

<sup>2</sup>This is not the instruction fetch bottleneck it seems to be. See Section 6.3.1 for why a second bus would remain idle over 90% of the time.

<sup>3</sup>These two bits could be used as a seventh instruction taken from the set of instructions whose three MSB are always zero, or as subroutine return and call flags as in the NC4016 and RTX-2000. They are left unused for simplicity.

of the instruction registers are shifted over by one instruction and filled-in at the far end with a Program Counter Fetch (PC@) instruction (see Section 6.2). After the last loaded instruction is executed, the PC@ instruction fetches the next instruction group into the ISR.

If the instruction after INST is a PC@, then INST is the last one of this group. This is signalled by the Last line (L) being high during the execution of INST. If the encoding of PC@ is conveniently chosen to be all zeroes, then L is simply the NOR of all the bits of S1. This line is used to overlap the execution of INST and PC@ whenever possible (Section 6.3.1).

## 6.2 Instruction Set

The Gullwing instruction set is identical to that of the F21 (Section 3.2.5), with some minor implementation simplifications which are beneficial to pipelining (Section 7.4):

- To fetch instructions, the F21 uses an internal memory access delay triggered by a lack of memory-accessing instructions in its ISR. Gullwing instead uses a Program Counter Fetch (PC@) instruction as the instruction fetching mechanism. It was the simplest choice to implement and the most deterministic in behaviour. This also allows skipping of unused instruction slots, which would otherwise have to be filled with no-ops.
- The F21 call and jump instructions use the remainder of the ISR to hold the low-order bits of the target address. The number of these bits depends on where in the ISR the instruction is placed. These bits would replace the corresponding ones in the PC, providing a limited branch distance relative to the current value of the PC. In Gullwing, the call and jump instructions take their target address from a memory address which follows the instruction, usually the very next one. This allows uniform, absolute addressing over the entire memory range.
- Like the calls and jumps, literal fetches on the F21 also place their argument in the remaining low-order bits. In Gullwing, the literal fetch instruction takes its literal value from a memory address which follows the instruction, usually the very next one. This makes for uniform and general storage of literals.

### 6.2.1 Instruction Packing

All instructions fit into 5-bit opcodes and have no operand fields or extended formats of any kind. Thus, up to six instructions can be packed into each 32-bit memory word (Section 6.2.7). This packing of instructions greatly reduces the number of instruction fetches for sequential code (Section 7.2.4) and makes possible a much higher code density (Chapter 8).

### 6.2.2 Flow Control

These instructions (Table 6.1) control the flow of execution of a program. They all access memory and use full-word absolute addresses either explicitly or implicitly. The jumps and calls fetch their target address from the next word in memory, while the PC@ and RET instructions

respectively take theirs from the Program Counter (PC) and the Return Register (R). They all execute in two cycles, except for PC@ which requires only one.

Mnemonic	Name	Operation
PC@	PC Fetch	Fetch next group of instructions into ISR
JMP	Jump	Unconditional Jump
JMP0	Jump Zero	Jump if top of data stack is zero. Pop stack.
JMP+	Jump Plus	Jump if top of data stack is positive. Pop stack.
CALL	Call	Call subroutine
RET	Return	Return from subroutine

Table 6.1: Gullwing Flow Control Instructions

### 6.2.3 Load, Store, and Literal Fetch

Table 6.1 shows the instructions used to access main memory. All accesses are word-wide. There are no half-word, byte or double-word formats. The Fetch Literal (LIT) instruction uses the next word of memory to contain its value. The other instructions use addresses stored in A and R to access main memory, and can post-increment them by one. The TOP register is always the source of data for stores and the target of loads and fetches<sup>4</sup>. All of these instructions execute in two cycles, except for LIT which requires only one.

Mnemonic	Name	Operation
LIT	Fetch Literal	Push in-line literal onto data stack
@A+	Load A Plus	Push MEM[A++] onto data stack
@R+	Load R Plus	Push MEM[R++] onto data stack
@A	Load A	Push MEM[A] onto data stack
!A+	Store A Plus	Pop data stack into MEM[A++]
!R+	Store R Plus	Pop data stack into MEM[R++]
!A	Store A	Pop data stack into MEM[A]

Table 6.2: Gullwing Load and Store Instructions

### 6.2.4 Arithmetic and Logic

The Gullwing ALU supports a small number of operations (Table 6.3). Other operations such as subtraction and bitwise OR must be synthesized by small code sequences. Binary operations

<sup>4</sup>Except for PC@, which loads the ISR.

are destructive: they pop two values from the DS and push one result back. Unary operations simply replace the contents of the top of the DS. All these instructions execute in a single cycle.

The  $+*$  operation is unusual in that it is conditional and non-destructive. It replaces the contents of TOP with the sum of TOP and of the first element of DS, only if the original value of TOP was odd (least significant bit is set). Combined with shift operations, it provides a means of synthesizing a multiplication operation.

Mnemonic	Name	Operation
NOT	Not	Bitwise complement of top of data stack
AND	And	Bitwise AND of top two elements of data stack
XOR	Xor	Bitwise XOR of top two element of data stack
+	Plus	Sum of top two elements of data stack
2*	Two Star	Logical shift left of top of data stack
2/	Two Slash	Arithmetic shift right of top of data stack
+*	Plus Star	Multiply step

Table 6.3: Gullwing ALU Instructions

### 6.2.4.1 Synthesizing More Complex Operations

The ALU instructions included, given the small amount of available opcodes, are the most frequent and useful: Addition is much more common than subtraction and XOR is frequently used as an equality test. Other primitive instructions can be synthesized with short sequences that fit inside a single memory word. Algorithm 1 shows how subtraction is implemented as a two's-complement operation followed by an addition, while the implementation of the bitwise OR operator  $A \vee B$  is expressed as  $A \oplus (B \wedge \bar{A})$ .

---

#### Algorithm 1 Gullwing Synthesis of Subtraction and Bitwise OR

---

Subtraction:  
 NOT LIT 1 + +  
 Bitwise OR:  
 OVER NOT AND XOR

---



---

#### Algorithm 2 Gullwing Synthesis of Multiplication (4x4)

---

>R 2\* 2\* 2\* 2\* R>  
 +\* 2/ +\* 2/ +\* 2/ +\* 2/  
 >R DROP R>

---

Algorithm 2 shows how the multiplication of two 4-bit numbers is implemented. The topmost element of the Data Stack contains the multiplier, while the second contains the multiplicand.

The multiplicand is first shifted right four times to align it with the empty space in front of the multiplier. It is then added to that empty space if the least significant bit (LSB) of the multiplier is set. The partial product and multiplier are shifted left by one, which both prepares the LSB with the next bit of the multiplier and has the effect of effectively shifting the multiplicand right by one relative to the partial product. The process is repeated four times,

shifting out the multiplier and leaving the completed 8-bit product on the top of the stack. The multiplicand is then discarded.

This method produces one bit of product approximately every two cycles and is a compromise between full hardware and software implementations. Its disadvantage is that it is limited to single-word products, and thus to half-word factors. However, it can also be optimized when the multiplier is much smaller than the multiplicand.

The multiplication of full-word factors, with a double-word product, could be accomplished in the same way by shifting across two registers. For example, TOP could hold the multiplicand and A the multiplier. By having `+*` test the LSB of A, and `2/` shifting from TOP into A, the process would leave the lower half of the product in A and the upper half in TOP. Executing `A>` would then place both halves on the Data Stack.

### 6.2.5 Stack Manipulation

Table 6.4 shows the instructions which manipulate the stacks and move data internally. They execute in one cycle.

Mnemonic	Name	Operation
<code>A&gt;</code>	A From	Push A onto data stack
<code>&gt;A</code>	To A	Pop data stack into A
<code>DUP</code>	Dup	Duplicate top of data stack
<code>DROP</code>	Drop	Pop data stack
<code>OVER</code>	Over	Push copy of second element onto data stack
<code>&gt;R</code>	To R	Pop data stack and push onto return stack
<code>R&gt;</code>	R From	Pop return stack and push onto data stack

Table 6.4: Gullwing Stack Manipulation Instructions

### 6.2.6 No-Op and Undefined

Finally, one instruction is defined as no operation (Table 6.5). It does nothing but fill an instruction slot and use a machine cycle. The remaining four undefined instructions are mapped to NOP also.

Mnemonic	Name	Operation
<code>NOP</code>	Nop	Do Nothing
<code>UND[0-3]</code>	Undefined	Do Nothing

Table 6.5: Gullwing No-Op and Undefined Instruction

## 6.2.7 Instruction Format and Execution Example

Instruction opcodes are 5-bit numbers. Six of these can be packed into a 32-bit memory word. The last two bits are unused (see Section 6.1.3). Literals and addresses always occupy an entire word. Each instruction position is termed a slot and the slots in a word constitute an instruction group. After a group is fetched, the instructions in the slots are executed in turn until they are exhausted. Figure 6.3 shows the memory layout of a random snippet of code tailored to show the layout of the instructions. They are laid out in left-to-right order for convenience only.

In the initial state of this example, the group at address 1 has just been fetched, the instruction in Slot 0 (S0) is about to be executed, and the Program Counter (PC) has been incremented to point to address 2. The first instruction (>R) executes without comment and shifts the contents of the ISR to the left by one instruction, placing JMP0 as the current instruction.

In the case of the JMP0 conditional jump being not taken, the PC is incremented by one to point to address 3, and the execution continues with DUP. When the LIT instruction is reached it fetches the literal pointed-to by the PC (at address 3) and increments the PC by one. The addition instruction is then executed.

When the PC@ instruction is reached, it fetches a new group from the location pointed to by the PC, which is currently address 4, and then increments the PC. Eventually the CALL instruction is reached. It loads the instruction group from the target address in the memory word pointed to by PC (at address 5), and loads the PC with the target address plus one. The remaining slots are never executed and are filled by convention with PC@.

Had the JMP0 conditional jump been taken, it would have executed similarly to the CALL instruction, loading the group at address 4 and leaving the PC pointing to address 5. The strictly word-wide access to memory is why the R> in Slot 0 at address 4 could not be placed in Slot 5 at address 1. Jumping or calling to an address always begins executing at Slot 0.

	S0	S1	S2	S3	S4	S5
1	>R	JMP0	DUP	LIT	+	PC@
2	Address for JMP0 (4)					
3	Number for LIT					
4	R>	XOR	CALL	PC@	PC@	PC@
5	Address for CALL					

Figure 6.3: Gullwing Instruction Format

## 6.3 State Machine and Register Transfer Description

The operation of the unpipelined Gullwing processor is very simple. There are no exceptions or interrupts. Algorithms 3 through 4 show the state transitions and register transfers. The inputs of its state machine consist of the opcode of the current instruction (INST), the current state (S), and whether the contents of the TOP register is all zeroes (=0), has the most significant bit clear (MSB0), or has the least significant bit set (LSB1). These three flags have been collapsed into a single input column (TOP) for brevity. 'Don't Care' values are represented by X. The outputs are the next state (N) and the various enable and select signals needed to steer the datapath, which are represented symbolically.

All actions on a line occur concurrently. For example, the operation of PC@ can be understood as: leave the state at zero, route the PC through the incrementer (+1) and store its output into both PC and MAR, and load the ISR with the current output of MEM. All instructions end with either loading or shifting the ISR (ISR<<), which changes INST to the next instruction to execute (see Figure 6.2). Instructions that take two cycles use the state bit to select which phase to execute.

Instructions which load instructions from memory, such as CALL and JMP, execute the same sequence of steps as PC@ in their second phase. For brevity, these steps have been replaced with the opcode.

---

**Algorithm 3** Gullwing Flow Control Instructions

---

	Inputs		Outputs		
	INST	TOP	S	N	Control
PC@	X		0	0	PC→(+1)→PC, MAR, MEM→ISR
JMP	X		0	1	MEM→PC, MAR
JMP	X		1	0	PC@
JMP0 =0			0	1	DS→TOP, DS(POP), MEM→PC, MAR
JMP0 !=0			0	0	DS→TOP, DS(POP), PC→(+1)→PC, MAR, ISR<<
JMP0 X			1	0	PC@
JMP+ MSB0			0	1	DS→TOP, DS(POP), MEM→PC, MAR
JMP+ MSB1			0	0	DS→TOP, DS(POP), PC→(+1)→PC, MAR, ISR<<
JMP+ X			1	0	PC@
CALL	X		0	1	PC→(+1)→R, R→RS, RS(PUSH), MEM→MAR, PC
CALL	X		1	0	PC@
RET	X		0	1	RS(POP), RS→R, R→PC, MAR
RET	X		1	0	PC@

---



---

**Algorithm 4** Gullwing No-Op and Undefined Instructions

---

	Inputs		Outputs				Inputs		Outputs		
	INST	TOP	S	N	Control		INST	TOP	S	N	Control
NOP	X		0	0	ISR<<	UND2	X		0	0	ISR<<
UND0	X		0	0	ISR<<	UND3	X		0	0	ISR<<
UND1	X		0	0	ISR<<						

---

---

**Algorithm 5** Gullwing ALU Instructions

---

Inputs		Outputs		
INST	TOP	S	N	Control
COM	X	0	0	TOP→ALU(NOT)→TOP, ISR<<
AND	X	0	0	TOP,DS→ALU(AND)→TOP, DS(POP), ISR<<
XOR	X	0	0	TOP,DS→ALU(XOR)→TOP, DS(POP), ISR<<
+	X	0	0	TOP,DS→ALU(+ )→TOP, DS(POP), ISR<<
2*	X	0	0	TOP→ALU(2*)→TOP, ISR<<
2/	X	0	0	TOP→ALU(2/)→TOP, ISR<<
++	LSB0	0	0	ISR<<
++	LSB1	0	0	DS,TOP→ALU(+ )→TOP, ISR<<

---

---

**Algorithm 6** Gullwing Load and Store Instructions

---

Inputs		Outputs		
INST	TOP	S	N	Control
LIT	X	0	0	MEM→TOP, TOP→DS, DS(PUSH), PC→(+1)→PC,MAR, ISR<<
@A+	X	0	1	A→MAR, (+1)→A
@A+	X	1	0	MEM→TOP, TOP→DS, DS(PUSH), PC→MAR, ISR<<
@R+	X	0	1	R→MAR, (+1)→R
@R+	X	1	0	MEM→TOP, TOP→DS, DS(PUSH), PC→MAR, ISR<<
@A	X	0	1	A→MAR
@A	X	1	0	MEM→TOP, TOP→DS, DS(PUSH), PC→MAR, ISR<<
!A+	X	0	1	A→MAR, (+1)→A
!A+	X	1	0	DS(POP), DS→TOP, TOP→MEM, PC→MAR, ISR<<
!R+	X	0	1	R→MAR, (+1)→R
!R+	X	1	0	DS(POP), DS→TOP, TOP→MEM, PC→MAR, ISR<<
!A	X	0	1	A→MAR
!A	X	1	0	DS(POP), DS→TOP, TOP→MEM, PC→MAR, ISR<<

---

---

**Algorithm 7** Gullwing Stack Instructions

---

Inputs		Outputs		
INST	TOP	S	N	Control
DUP	X	0	0	TOP→DS, DS(PUSH), ISR<<
DROP	X	0	0	DS→TOP, DS(POP), ISR<<
OVER	X	0	0	DS→TOP, TOP→DS, DS(PUSH), ISR<<
R>	X	0	0	RS(POP), RS→R, R→TOP, TOP→DS, DS(PUSH), ISR<<
>R	X	0	0	DS(POP), DS→TOP, TOP→R, R→RS, RS(PUSH), ISR<<
A>	X	0	0	A→TOP, TOP→DS, DS(PUSH), ISR<<
>A	X	0	0	TOP→A, DS→TOP, DS(POP), ISR<<

---

### 6.3.1 Improvement: Instruction Fetch Overlap

Using an instruction (PC@) to fetch the next group of instructions eliminates the need for dedicated instruction-fetching hardware. The fact that it is also shifted into the ISR for free is also very elegant. However, this means that at least one out of every seven instructions executed will be a PC@, or about 14%<sup>5</sup>. This overhead could be reduced by overlapping fetching with execution while the memory bus is free.

A priori, it is unclear how much benefit would come from overlapping the instruction fetch. Koopman provides a table of the dynamic instruction execution frequencies of Forth primitives averaged over a set of benchmarks [Koo89, 6.3]. These primitives map almost directly to the Gullwing instruction set and the benchmarks from Section 7.1 are also written in a Forth-like language. The data predicts that approximately half of the executed primitives are either flow control or load/store instructions and thus access memory. Assuming that this probability is evenly distributed, the fetching of instructions could be overlapped half the time, reducing the overhead to about 7% for straight-line code.

Accomplishing this overlap depends on this fact: If the the last instruction before a PC@ does not access memory, then both instructions can be executed simultaneously without conflict. Figure 6.2 shows how the ISR makes this possible. If the next instruction to be executed is a PC@<sup>6</sup>, then the current instruction being executed is the last one of this group. This is signalled by raising the Last flag (L). If the current instruction does not access memory<sup>7</sup>, then instead of shifting the ISR at the end of its execution, the instruction will fetch the next group of instructions from memory in the same manner as PC@, as if it had executed concurrently.

Implementing this optimization requires adding the L bit as an input to the state machine. Flow control and load/store instructions ignore this bit since they always access memory. The remaining instructions now have two versions, selected by L, which either shift or load the ISR. Algorithms 8, 9, and 10 show the necessary changes.

Analysis of code executed with an overlapping instruction fetch (Appendix B.2.2) confirms that the actual overhead of explicit PC@ instructions is reduced to 4.1 to 2.2% of the total executed instructions . It also shows that 4.2 to 7.5% of instructions are executed concurrently ('FOLDS') with a PC@. This demonstrates a 50.6 to 77.3% reduction in instruction fetch overhead.

---

<sup>5</sup>The actual overhead will be lower since other flow control instructions do their own instruction loading. Without instruction fetch overlapping the actual overhead of instruction fetching is 8.3 to 9.7%.

<sup>6</sup>PC@ is conveniently encoded as all zeroes.

<sup>7</sup>Since the opcodes are divided about equally between memory and non-memory instructions, a single bit (the MSB, for example) can be used to test if the instruction accesses memory. This should simplify the implementation of the state machine.

---

**Algorithm 8** Gullwing ALU Instructions with Instruction Fetch Overlap

---

Inputs		Outputs				
INST	TOP	L	S	N	Control	
COM	X	0	0	0	TOP→ALU(NOT)→TOP, ISR<<	
COM	X	1	0	0	TOP→ALU(NOT)→TOP, PC@	
AND	X	0	0	0	TOP,DS→ALU(AND)→TOP, DS(POP), ISR<<	
AND	X	1	0	0	TOP,DS→ALU(AND)→TOP, DS(POP), PC@	
XOR	X	0	0	0	TOP,DS→ALU(XOR)→TOP, DS(POP), ISR<<	
XOR	X	1	0	0	TOP,DS→ALU(XOR)→TOP, DS(POP), PC@	
+	X	0	0	0	TOP,DS→ALU(+)→TOP, DS(POP), ISR<<	
+	X	1	0	0	TOP,DS→ALU(+)→TOP, DS(POP), PC@	
2*	X	0	0	0	TOP→ALU(2*)→TOP, ISR<<	
2*	X	1	0	0	TOP→ALU(2*)→TOP, PC@	
2/	X	0	0	0	TOP→ALU(2/)→TOP, ISR<<	
2/	X	1	0	0	TOP→ALU(2/)→TOP, PC@	
++	LSB0	0	0	0	ISR<<	
++	LSB0	1	0	0	PC@	
++	LSB1	0	0	0	DS,TOP→ALU(+)→TOP, ISR<<	
++	LSB1	1	0	0	DS,TOP→ALU(+)→TOP, PC@	

---

---

**Algorithm 9** Gullwing Stack Instructions with Instruction Fetch Overlap

---

Inputs		Outputs				
INST	TOP	L	S	N	Control	
DUP	X	0	0	0	TOP→DS, DS(PUSH), ISR<<	
DUP	X	1	0	0	TOP→DS, DS(PUSH), PC@	
DROP	X	0	0	0	DS→TOP, DS(POP), ISR<<	
DROP	X	1	0	0	DS→TOP, DS(POP), PC@	
OVER	X	0	0	0	DS→TOP, TOP→DS, DS(PUSH), ISR<<	
OVER	X	1	0	0	DS→TOP, TOP→DS, DS(PUSH), PC@	
R>	X	0	0	0	RS(POP), RS→R, R→TOP, TOP→DS, DS(PUSH), ISR<<	
R>	X	1	0	0	RS(POP), RS→R, R→TOP, TOP→DS, DS(PUSH), PC@	
>R	X	0	0	0	DS(POP), DS→TOP, TOP→R, R→RS, RS(PUSH), ISR<<	
>R	X	1	0	0	DS(POP), DS→TOP, TOP→R, R→RS, RS(PUSH), PC@	
A>	X	0	0	0	A→TOP, TOP→DS, DS(PUSH), ISR<<	
A>	X	1	0	0	A→TOP, TOP→DS, DS(PUSH), PC@	
>A	X	0	0	0	TOP→A, DS→TOP, DS(POP), ISR<<	
>A	X	1	0	0	TOP→A, DS→TOP, DS(POP), PC@	

---

---

**Algorithm 10** Gullwing No-Op and Undefined Instructions with Instruction Fetch Overlap

---

Inputs		Outputs					Inputs		Outputs				
INST	TOP	L	S	N	Control	INST	TOP	L	S	N	Control		
NOP	X	0	0	0	ISR<<	UND1	X	1	0	0	PC@		
NOP	X	1	0	0	PC@	UND2	X	0	0	0	ISR<<		
UND0	X	0	0	0	ISR<<	UND2	X	1	0	0	PC@		
UND0	X	1	0	0	PC@	UND3	X	0	0	0	ISR<<		
UND1	X	0	0	0	ISR<<	UND3	X	1	0	0	PC@		

---



# Chapter 7

## Comparisons With DLX/MIPS

As stated in Section 1.1, this thesis aims to divide the family of stack-based computers into first and second generations. Part of this distinction consists of showing that the second generation resembles the register-based machines which replaced the first.

This chapter supports this argument by comparing the Gullwing processor from Chapter 6 with the well-known MIPS and DLX processors used as demonstrators by Hennessy & Patterson. This comparison is based on statistics derived from the organization and execution of programs and further based on a comparison of the pipeline structure of each microprocessor. The characteristics compared include: cycle time, cycle count, cycles per instruction, instruction count, dynamic instruction mix, memory accesses per cycle, code size, and code density.

### 7.1 Gullwing Benchmarks

Unfortunately, C compilers for second-generation stack computers are rare and, when available, are proprietary or experimental. Additionally, there are no existing operating systems or peripherals for Gullwing. Thus, it is not possible at the time to compile the SPEC benchmarks for this platform. The software used to test Gullwing performs mostly symbol table searches and machine code compilation and interpretation. It could be loosely viewed as a tiny subset of the GCC, Lisp, and Perl components of the SPECint benchmarks.

The software was originally written to explore a Forth-like self-extensible language, named 'Flight'. It is composed of a language kernel used to compile extensions to itself, including a metacompiler and a Gullwing virtual machine, which are then used to re-create the Flight language kernel and all its extensions in a self-hosting manner. This process is described in the following subsections. The source for all the benchmarks is listed in Appendix A.

#### 7.1.1 Flight Language Kernel (Bare)

The Flight language kernel is a small (about 800 32-bit memory words) piece of machine code. Its main loop reads in a name, searches for it in a linear dictionary and if found, calls the function associated with that name. The kernel's other built-in functions include management of a linear input buffer, string comparison, creation of dictionary entries, conversion of decimal numbers to binary, and compilation of the Gullwing opcodes. Source code fed to the kernel

is executed as it is received and can contain the name of any built-in or previously defined function. The source to the kernel is listed in Appendix A.1.

### 7.1.2 Flight Language Extensions (Ext.)

The language defined by the Flight kernel is extremely spartan. The Flight extensions begin by creating convenience functions to simplify the definition, lookup, and compilation of other functions. These are used to create various functions for string copying, compilation, and printing, multiplication and division of integers, binary to decimal conversion, and a simple 'map' function generator. These new function are used to construct small demonstrations of Fibonacci numbers and Caesar ciphers. These demonstrations account for a negligible portion of the total execution time and are used mainly as regression tests for the underlying code. The Flight language extensions exercise the functions of the Flight kernel and are composed of 329 lines of code<sup>1</sup> containing 1710 names<sup>2</sup> and executing 5,018,756 instructions (4,462,579 without the demonstrations). The source is listed in Appendix A.2.

### 7.1.3 Virtual Machine (VM)

The virtual machine is built upon the Flight extensions<sup>3</sup> (without demonstration code). It defines a software emulation of all the Gullwing opcodes, some bounds-checking functions for a given area of memory, and an instruction extraction and interpretation loop that reads Gullwing machine code. Compiled code executes in the virtual machine with an overhead of about 31 emulator instructions per actual emulated instruction. The end result is a fully contained emulation of the Gullwing microprocessor. The compilation of the virtual machine itself requires 276 lines of code containing 769 names. This process exercises the functions of the kernel and its extensions and executes 4,288,157 instructions. The source is listed in Appendix A.3.1.

**Metacompiler** The metacompiler manipulates the Flight kernel to retarget its operations to the memory area used by the virtual machine. It saves and restores the internal state of the kernel, such as the location of the dictionary and input buffer, and defines a new main loop to replace the kernel's default one. The new loop first searches the new dictionary in the virtual machine memory and if there is no match, continues the search in the kernel's original dictionary. This allows the use of functions previously defined outside of the virtual machine to bootstrap code inside it. The compilation of the metacompiler requires 106 lines of code containing 491 names. This process exercises the functions of the kernel and its extensions and executes 4,069,393 instructions. The source is listed in Appendix A.3.2.

**Self-Hosted Kernel** The first thing compiled into the virtual machine is another instance of the Flight kernel. While the original kernel was written in assembly language, this new kernel

---

<sup>1</sup>Only non-blank lines are counted.

<sup>2</sup>Each name is either a function name, or a string to be processed.

<sup>3</sup>Although the Virtual Machine software includes the compilation of the Extensions, the latter contributes to only 2.2% of the total number of executed instructions.

is defined using the functions of the original kernel and all its extensions. The result is a higher-level description of the Flight kernel, written in itself. The new kernel binary residing in the virtual machine memory area is identical to the original<sup>4</sup>. The compilation of the self-hosted kernel requires 246 lines of code containing 681 names. This process exercises the functions of the kernel and its extensions via the indirection of the metacompiler and executes 6,511,976 instructions. The source is listed in Appendix A.3.3.

**Flight Language Kernel Extensions** Now that a Flight kernel resides in the virtual machine memory, is it possible to start the virtual machine and execute this new kernel. The Flight language extensions from Section 7.1.2 are fed to the new kernel, exercising the same code, but through the emulation layer of the virtual machine. This executes 184,993,301 instructions, taking about 90% of the total execution time of the VM test suite.

## 7.2 Comparison of Executed Benchmark Code

This section compares the properties of Gullwing and DLX/MIPS when executing integer code. The DLX/MIPS data is derived from published SPECint92 [HP96, fig. 2.26] and SPECint2000 [HP02, fig. 2.32] results. The Gullwing data (Appendix B) is compiled from the execution of software built upon the kernel of a low-level Forth-like language, named 'Flight', tailored to Gullwing (Appendix A). The properties compared are dynamic instruction mix, cycles per instruction (CPI), memory accesses per cycle, and instructions per memory word.

Note that while the DLX and MIPS processors include optimizations such as operand forwarding, the Gullwing processor has none except for the instruction fetch overlapping described in Section 6.3.1. Other optimizations have been left for future work (Section 9.2).

### 7.2.1 Dynamic Instruction Mix

Tables 7.1 and 7.2 compare the proportions of instructions executed during the benchmarks. The Gullwing data is taken from the Extensions and Virtual Machine executed instruction counts from Appendix B.2.2. Since the Extensions software deals mainly with code compilation, it is compared to the GCC component of the SPECint92 and SPECint2000 suites. The Virtual Machine software is compared to the other interpretive components: Lisp from SPECint92 and Perl from SPECint2000.

The Gullwing instructions are grouped together and their statistics summed in order to match the meaning of the equivalent DLX/MIPS instructions. For example, while MIPS has one load instruction for all purposes, Gullwing has different ones (@A, @A+, @R+) depending on the addressing mode and the load address register.

The statistics for the XOR and NOT instructions are combined since the SPECint92 data groups them together even though the SPECint2000 data lists them separately<sup>5</sup>. Statistics for

---

<sup>4</sup>Actually, there is a gap of one memory word between functions due to a quirk of the compilation process, but the actual code is the same.

<sup>5</sup>From SPECint2000 data:

Table 7.1: 2.8% XOR, plus 0.3% other logical ops.

Table 7.2: 2.1% XOR, plus 0.4% other logical ops.

instructions which do not exist in a given machine are represented by a dash ('-').

For both the compiler and interpreter test data, the dynamic instruction mix of the Gullwing processor differs from that of DLX and MIPS in some significant ways:

- The proportion of loads and stores is lower. I believe this originates from a lack of complex data structures in the Gullwing software, and the need for DLX and MIPS to use a stack in main memory to pass parameters to subroutines.
- There are many more immediate loads (fetches) since Gullwing cannot include small literal operands within most instructions as DLX and MIPS do.
- The proportion of calls and returns is much greater. The stack-based architecture of Gullwing results in extremely efficient procedure linkage, which makes the inlining of code unnecessary in most cases.
- About a quarter of all the instructions executed are stack manipulation instructions. Some are manipulations made explicit by the absence of operands in Gullwing instructions as a consequence of the lack of random access to a stack. DLX and MIPS include implicit move, copy, and delete operations within their instructions by using a three-operand instruction format which gives random access to their registers. A large portion of these stack manipulations are moves of addresses from the top of the Data Stack to the Address Register, as performed by the >A instruction. Section 9.2.3 discusses a mean to reduce the overhead of these moves.

Additionally, the interpreter dynamic instruction mix (Table 7.2) has some further differences:

- The proportion of conditional jumps is lower on Gullwing. This is likely because of the lack of conditionals in the main VM loop, which uses instead a table of function addresses.
- The large incidence of shifts is due to the use of the shift-left (2/) instruction in the VM to extract individual instructions out of a memory word.

Benchmark	GCC (92)	GCC (2000)	Extensions	
DLX/MIPS Instr.	DLX	MIPS	Gullwing	Gullwing Instr.
load	22.8%	25.1%	15.2%	@A, @A+, @R+
store	14.3%	13.2%	0.4%	!A, !A+, !R+
add	14.6%	19.0%	10.3%	+
sub	0.5%	2.2%	-	
mul	0.1%	0.1%	-	
div	0.0%	-	-	
compare	12.4%	6.1%	-	
load imm	6.8%	2.5%	16.8%	LIT, PC@
cond branch	11.5%	12.1%	6.6%	JMPO, JMP+ (incl. TAKEN jumps)
cond move	-	0.6%	-	
jump	1.3%	0.7%	2.1%	JMP
call	1.1%	0.6%	6.4%	CALL
return, jmp ind	1.5%	0.6%	6.4%	RET
shift	6.2%	1.1%	0.2%	2/, 2*
and	1.6%	4.6%	0%	AND
or	4.2%	8.5%	-	
other (xor, not)	0.5%	2.5%	6.5%	XOR, NOT
other (moves)	-	-	29.2%	DUP, DROP, OVER, >R, R>, >A, A>
other	-	-	0%	NOP, +*

Table 7.1: Compilers Dynamic Instruction Mix

Benchmark	Lisp(92)	Perl (2000)	VM	
DLX/MIPS Instr.	DLX	MIPS	Gullwing	Gullwing Instr.
load	31.3%	28.7%	9.3%	@A, @A+, @R+
store	16.7%	16.2%	4.6%	!A, !A+, !R+
add	11.1%	16.7%	7.7%	+
sub	0.0%	2.5%	-	
mul	0.0%	0.0%	-	
div	0.0%	-	-	
compare	5.4%	3.8%	-	
load imm	2.4%	1.7%	18.0%	LIT, PC@
cond branch	14.6%	10.9%	2.0%	JMP0, JMP+ (incl. TAKEN jumps)
cond move	-	1.9%	-	
jump	1.8%	1.7%	2.7%	JMP
call	3.1%	1.1%	5.0%	CALL
return, jmp ind	3.5%	1.1%	7.5%	RET
shift	0.7%	0.5%	12.6%*	2/, 2*
and	2.1%	1.2%	3.4%	AND
or	6.2%	8.7%	-	
other (xor, not)	0.1%	3.1%	3.9%	XOR, NOT
other (moves)	-	-	23.3%	DUP, DROP, OVER, >R, R>, >A, A>
other	-	-	0%	NOP, +*

Table 7.2: Interpreters Dynamic Instruction Mix

## 7.2.2 Cycles Per Instruction

The CPI of Gullwing can be readily determined from the ratio of the total number of cycles to instructions in each test (Appendix B.2.1). Both tests exhibit a CPI of about 1.3 (Appendix B.2.3).

Table 7.4 shows the contribution to the CPI from each instruction type, calculated from product of the cycle count and of the frequency of each instruction type listed in Appendix B.2.4. A more detailed breakdown can be derived from the instruction frequencies in Appendix B.2.2.

There is sufficient published data in Hennesy & Patterson [HP96] from the SPECint92 suite to estimate the CPI of the DLX processor for the GCC and Lisp components, beginning with a base CPI of 1.00 and then adding the penalties from branch and load stalls. The load penalty is calculated as the percentage of all the loads<sup>6</sup> (as percentage of all instructions) which stall [HP96, fig. 3.16]. The branch penalty is given directly [HP96, fig. 3.38]. The resulting CPI is 1.11 for the GCC component, and 1.15 for Lisp (Table 7.3).

<sup>6</sup>including immediate loads

The higher CPI of Gullwing does not compare favourably with that of DLX. However, loads and taken jumps on Gullwing take two cycles, which is as if a load or branch stall always occurred. Therefore, Gullwing is really architecturally equivalent to a DLX without load delay slots and without delayed branches, which always stalls on loads and taken branches. For example, if 100% of loads are assumed to stall on DLX, their load penalty increases to 29.6% for GCC and 33.7% for Lisp, which raises the total CPI of DLX to 1.34 and 1.41 respectively, which is comparable to Gullwing.

Correspondingly, there are some possible optimizations to Gullwing which would reduce the CPI of loads and jumps and make Gullwing equivalent to a normal DLX (Section 9.2.3).

Test	GCC	Lisp
Total Loads	29.6%	33.7%
Load Stalls	23%	24%
Load Penalty	6.81%	8.10%
Branch Penalty	4%	7%
Total Penalty	10.8%	15.1%
Overall CPI	1.11	1.15

Table 7.3: DLX CPI with Load and Branch Penalties

Test	Extensions	VM
Instr. Type	Fraction of Total CPI	
Conditionals	0.071	0.031
Subroutine	0.300	0.304
Fetches	0.168	0.180
Load/Store	0.312	0.276
ALU	0.169	0.277
Stack	0.292	0.233
Total	1.312	1.301

Table 7.4: Gullwing CPI by Instruction Type

### 7.2.3 Memory Accesses Per Cycle

The memory bandwidth usage of a processor can be expressed as the average number of memory accesses per cycle. Assuming a memory access time equal to the cycle time of the processor, one memory access per cycle implies full usage of the available bandwidth to memory.

For the MIPS and DLX processors, determining the memory bandwidth usage is straightforward. An instruction is fetched and another completed every cycle, assuming no stalls. Only load and store instructions explicitly access memory otherwise. Immediate operands are within the instruction opcode and so are counted as part of the instruction fetches. Large numbers which require two loads are rare and ignored. Flow control instructions do not add to the instruction fetches since they only steer the Instruction Fetch pipeline stage.

Table 7.6 takes data from the benchmarks of Section 7.2.1 and shows that loads and stores make up on average 42.1% of the total number of executed instructions. Thus, they contribute an additional average of 0.421 memory accesses per cycle. When added to the instruction fetches, this totals to an average 1.421 memory accesses per cycle for DLX/MIPS, divided between separate instruction and data memories.

Measuring the memory usage of Gullwing is a little more complicated. There is no pipeline. Several instructions are loaded in one memory access (see next section and Section 6.2.7). Flow control instructions do their own fetching or loading of instructions. There are no immediate operands. Different instructions take different numbers of cycles to execute and make different numbers of memory accesses. However, there is only one memory for both instruction and

data fetches/loads and thus only one Memory Address Register (MAR) (Section 6.1.1). Thus, a memory access is defined as an alteration of the MAR.

The average number of memory accesses per cycle for an instruction is calculated as so:

$$\text{Avg. \# of accesses/cycle} = (\# \text{ of accesses} \div \# \text{ of cycles}) \times \text{avg. fraction of total cycles}$$

The number of accesses is determined from the register transfer description (Section 6.3) as is the number of cycles, which can alternately be seen in Appendix B.2.4. The percentage of total cycles is listed under the C/C column in Appendix B.2.2. Table 7.5 condenses this data.

In summary, the Gullwing processor performs an average of 0.667 memory accesses per cycle while using a single memory bus, compared to 1.421 for DLX/MIPS which uses two.

It would be ideal to reduce Gullwing's memory bandwidth further, but part of the reason it is already low is because of the extra instructions required to manipulate the stack (which do not access memory). Reducing this instruction overhead will increase the proportion of memory-accessing instructions and thus increase the average number of memory accesses per cycle, towards a maximum of one (Section 9.2.3).

Either way, since loads and stores cannot (and have no need to) overlap instruction fetches on Gullwing, the maximum number of memory accesses per cycle cannot exceed one. By comparison, MIPS must have a *minimum* of one memory access per cycle. When MIPS must manipulate its call stack, it must perform additional memory accesses in the form of data loads and stores, increasing its memory bandwidth further<sup>7</sup>.

Test				Extensions	VM
Instruction	Accesses	Cycles	Acc./Cyc.	Fraction of Total Cycles	
PC@	1	1	1	0.031	0.017
FOLDS	1	1	1	0.032	0.058
CALL	2	2	1	0.098	0.076
JMP	2	2	1	0.033	0.042
JMP0, JMP+	1	1	1	0.046	0.009
JMP0, JMP+ (TAKEN)	2	2	1	0.007	0.001
RET	2	2	1	0.098	0.115
LIT	1	1	1	0.097	0.122
@A, @A+, @R+	2	2	1	0.233	0.142
!A, !A+, !R+	2	2	1	0.006	0.070
all others	0	1	0	0.352	0.392
Total of (Acc./Cyc.) × Fraction				0.681	0.652
Average				0.667	

Table 7.5: Gullwing Memory Accesses Per Cycle (Total)

<sup>7</sup>Implying, of course, a second memory bus.

Test	Compiler		Interpreter	
CPU	DLX	MIPS	DLX	MIPS
Component	GCC92	GCC00	Lisp	Perl
Loads	22.8%	25.1%	31.3%	28.7%
Stores	14.3%	13.2%	16.7%	16.2%
Total	37.1%	38.3%	48.0%	44.9%
Average	37.7%		46.5%	
Average	42.1%			

Table 7.6: DLX/MIPS Memory Accesses Per Cycle Caused by Loads and Stores

## 7.2.4 Instructions per Memory Word

For DLX/MIPS, the density of code in memory is simple: there is exactly one instruction per memory word. The opcode and its operands or literals are contrived to fit. For Gullwing, opcodes have no explicit operands and literals are placed in the next memory word (Section 6.2.7). This allows up to six instruction opcodes to fit in a single memory word. The actual number is variable due to the word-addressed nature of jumps, calls, and returns.

However, if the total number of instructions is compared to the total number of instruction slots (Appendix B.1.5), it is apparent that most of the available instruction slots are wasted. Chapter 8 explains why and shows a method to make these wasted slots available.

Appendix B.1.4 shows that in most cases there is one, two, or six instructions per word that contains instructions (instead of a literal). The ones and twos imply higher-level code where the memory word contains a call or jump, while the sixes are low-level code without changes in program flow. The result is an overall code density of about 1.2 instructions per memory word (Appendix B.1.5). If the words containing literals and addresses are excluded, the average memory word containing instructions contains between two and three instructions.

Despite the modest overall increase in code density, the higher density of words which contain instructions reduces the number of memory accesses required to fetch instructions. Until a group of instructions is exhausted, the memory bus is free for loads and stores, and for fetching addresses and literals and the next instruction word also.

Gullwing spends 0.441 memory accesses per cycles to fetch or load instructions<sup>8</sup>, compared to 1.00 for MIPS. The fraction of all memory accesses used for fetching or loading instructions on Gullwing is still approximately the same relative to MIPS (66.1% versus 70.4%, based on Section 7.2.3). Therefore, the important benefit of packing multiple instructions per memory word is not so much a reduction in code size, as one in instruction fetch memory bandwidth.

---

<sup>8</sup>Calculated as the sum of the fraction of total memory accesses done by instructions which alter the Program Counter (PC): PC@, FOLDS, CALL, RET, all JMPs, and LIT, averaged over both the Ext. and VM tests.

### 7.2.4.1 Basic Blocks and Instruction Fetch Overhead

This reduction in the number of instruction fetches manifests as a reduction of the number of PC Fetch (PC@) instructions required to load the next sequential instruction word. If a basic block of code spans several memory words, the last instruction in each word will be a PC@ which fetches the next group of instructions (Sections 6.2.7 and 6.3.1). The call and jump instructions that terminate basic blocks do their own instruction loading.

Appendix B.2.5 shows that the average length of a basic block, measured in instructions, fits well within the six instruction slots in each memory word. In fact, 71.1 to 80.9% of basic blocks fit in a single memory word. This places a limit on how much overhead the PC@ instructions can cause.

This PC@ overhead drops as the memory word gets wider and more instructions can be packed in one. At the limit, when all basic blocks fit into one memory word, no PC@ instructions will be executed and all loading of instructions will come from calls and jumps. In effect, each memory word behaves as a cache line. This effect is amplified by the mechanism proposed in Chapter 8.

## 7.3 Behaviour of Iteration, Recursion, and Subroutine Calls

A clearer understanding of the differences between Gullwing and MIPS is obtained by comparing the behaviour of simple, demonstrative programs written for both processors. These demonstrators are examples of universal small-scale features of code, regardless of the higher language used: iteration, recursion, and subroutine calls. They are not meant as benchmarks of overall performance, but as precise, singular tests made to expose the low-level details of how each processor executes code, unclouded by the complexities and biases of actual purposeful software. A direct comparison can be made between both processors since the algorithms and implementation styles are identical, contrary to the heterogeneous tests from the previous section.

### 7.3.1 Measured Properties

The code examples given are simple enough that their properties do not depend on input data and can thus be determined by inspection. The properties are measured for a single algorithm step, which is one loop, recursive call, or sequence of subroutine calls. For looping and tail-recursive code, only the code inside the loop is considered since the entry and exit code contribute a fixed amount regardless of the number of iterations. For recursive and subroutine code, the entire code is considered. The measured properties are:

**Memory Words** This is a measure of code size, assuming 32-bit memory words. Instructions are displayed such that one line represents one word of memory.

**Instructions** This is the number of instructions executed. All other things being equal, a difference in number suggests a difference in suitability to the given task.

**Memory Accesses** This is the count of the number of memory fetches, loads, and stores<sup>9</sup>. For MIPS, this also include the fetching of instructions. There is no such distinction in Gullwing: fetching instructions is a special case of data loads.

**Cycles** This is the count of the number of cycles required. The cycle time is assumed to be the same for both processors. All MIPS instructions each count as one cycle, while the Gullwing instructions count as one or two cycles<sup>10</sup>. The MIPS pipeline is assumed to be full at the start and to never stall.

**Memory Accesses Per Cycle** This is a measure of the memory bandwidth required. A measure of one access per cycle implies a fully-utilized memory bus.

**Cycles Per Instruction (CPI)** The MIPS pipeline is assumed to never stall and thus to always have a CPI of one. Since Gullwing instructions take one or two cycles, the CPI depends on the code being executed.

**Instructions Per Memory Word** This is a measure of code density. For MIPS, this measure is always one. For Gullwing, it is variable.

## 7.3.2 Demonstrators

The C source for the demonstrators was compiled to assembly language, at various optimization levels, with a little-endian MIPS32 cross-compiling version of GCC 3.4.4. The simplest resulting code was then hand-optimized, if possible, to eliminate the quirks of the compiler and reproduce optimal hand-assembled code. From this, the equivalent stack-oriented code was written with an effort towards keeping the algorithm unchanged.

The first demonstrator is the triangular numbers function. It is the additive analogue of the factorial function: each triangular number is the sum of all the preceding positive integers. It is quite possibly the tiniest non-trivial example of looping code (since the number of repetitions is part of the calculations). It is also interesting since it is expressible in several forms. The iterative, recursive, and tail-recursive forms are analyzed<sup>11</sup>.

---

<sup>9</sup>In the MIPS pipeline, all instructions require one memory access to fetch the instruction, and another access if the instruction is a load or a store. For Gullwing, there is no division between instruction and data memory. The fetching or loading of instructions is done by the flow control instructions. Calls, returns, jumps, and taken conditionals perform two memory accesses. Untaken conditionals and PC@ (PC Fetch) do only one. A PC@ is implied at the end of a group of instructions that does not end in a flow control instruction. Literal fetches take one cycle.

<sup>10</sup>For Gullwing, conditional jumps use a variable number of cycles. If taken, they behave like a jump, call, or return, taking two cycles. If not taken, they merely advance to the next instruction, taking one cycle. Fortunately, in the programs shown, the conditional jump is used to test the loop exit condition or the recursion terminal case and thus takes one cycle for all algorithm steps except the last one. It is thus considered a one-cycle instruction. See Section 6.3 for details.

<sup>11</sup>There is a closed form of the triangular numbers algorithm which I've not investigated here because it requires multiplication, which Gullwing does not have:  $Tri_n = \frac{1}{2}n(n + 1)$ . Since it's a straightforward expression, it should evaluate in the same manner on stack-based and register-based computers, with perhaps a small stack manipulation overhead on stack computers, depending on the exact instruction set.

The second demonstrator is a sequence of progressively more complex subroutine calls implementing the sum of their parameters. While the function is trivial, it is implemented in a manner that highlights parameter passing in nested subroutines.

### **7.3.3 Iterative Triangular Numbers**

This is a straightforward iterative version. It adds a decrementing loop counter to a sum.

Gullwing requires over twice as many instructions for iterative code since values must be explicitly duplicated, moved about the two stacks, or loaded from memory. In contrast, MIPS has random access to all registers and explicit source and destination operands which can also be small literals. This implicitly combines duplication, manipulation, and literal fetches with the actual arithmetic/logical operation. It naturally follows that many more of the simpler stack operations will be required to accomplish the same effect. Sections 9.2.2 and 9.2.3 discuss means of implementing more complex stack operations.

These extra instructions required by Gullwing are mostly stack manipulations, which add cycles but do not access memory, and thus artificially reduce the memory bandwidth. Similarly, the CPI is lower than in all the other demonstrators due to these additional single-cycle instructions.

The size overhead of these instructions is entirely absorbed by the capacity to pack multiple instructions per memory word, but any potential reduction in the number of memory accesses is offset by the greater number of literals and addresses which must be fetched.

---

**Algorithm 11** Triangular Iterative C Source

---

```
int triangular (int foo) {
    int bar = 0;
    while(foo != 0){
        bar = bar + foo;
        foo = foo - 1;
    }
    return bar;
}
```

---

---

**Algorithm 12** Triangular Iterative MIPS32  
Assembly

---

```
        move    $2,$0
$L7:    beq     $4,$0,$L8
        addu    $2,$2,$4
        b      $L7
        addiu   $4,$4,-1
$L8:    j      $31
        nop
```

---

---

**Algorithm 13** Triangular Iterative Gullwing  
Assembly

---

```
        LIT >R
        0
        Loop:  DUP JMP0 R> OVER + >R
        End
        LIT + JMP
        -1
        Loop
        End:  DROP R> RET
```

---

Main Loop	MIPS	Stack	Stack/MIPS
Mem Words	4	5	1.25
Instructions	4	9	2.25
Mem Accesses (instr+data)	(4+0)	5	1.25
Cycles	4	10	2.50
Derived Measures	MIPS	Stack	Stack/MIPS
Accesses/Cycle	1.00	0.50	0.50
Cycles/Instruction	1.00	1.11	1.11
Instructions/Word	1.00	1.80	1.80

Table 7.7: Triangular Iterative Code Comparison

MIPS		Gullwing	
move	0	4	>R, R>, DUP, OVER
addu, addiu	2	2	+
load imm	0	1	LIT
beq	1	1	JMP0
b	1	1	JMP

Table 7.8: Iterative Dynamic Instruction Mix

### 7.3.4 Recursive Triangular Numbers

The recursive version stores the intermediate values on the stack across a series of recursive calls, summing them once the terminal case is reached.

The stack-oriented instruction set of Gullwing eliminates all the explicit call stack manipulations and copying of arguments done by MIPS, approximately halving the number of memory words required and also resulting in straightforward code<sup>12</sup>.

Gullwing has to perform very little extra stack manipulation and so does not have an artificially reduced memory bandwidth as in the iterative example. On the other hand, MIPS must now manage a stack in memory, increasing its cycle and memory access counts to well above those of Gullwing. The result is that Gullwing genuinely requires about half the memory bandwidth than MIPS. Section 9.2.1 discusses a method which could eliminate the stack management overhead of MIPS.

---

**Algorithm 14** Triangular Recursive C Source

---

```
int triangular (int foo) {
    if (foo == 0) {
        return 0;
    } else {
        return foo + triangular(foo-1);
    }
}
```

---

---

<sup>12</sup>It is fair to say that the MIPS code shown, which has been cleaned-up from the actual compiler output, is spaghetti-code. It's not possible to optimize it further without altering the algorithm (recurring to a separate entry point), and even then it would only save one instruction. The convoluted nature of efficient code created by a compiler is not usually a human concern, but there's no way I can call it a good thing.

**Algorithm 15** Triangular Recursive MIPS32  
Assembly

```

Tri: addiu    $sp,$sp,-32
      sw      $31,28($sp)
      sw      $16,24($sp)
      move    $16,$4
      beq     $4,$0,$L1
      move    $2,$0
      addiu   $4,$4,-1
      jal     Tri
      addu    $2,$2,$16
$L1: lw      $31,28($sp)
      lw      $16,24($sp)
      j       $31
      addiu   $sp,$sp,32

```

**Algorithm 16** Triangular Recursive Gullwing  
Assembly

```

Tri: DUP JMP0 DUP LIT + CALL
      End
      -1
      Tri
      +
End: RET

```

Entire Code	MIPS	Stack	Stack/MIPS
Mem Words	13	6	0.46
Instructions	13	8	0.62
Mem Accesses (instr+data)	(13+4)	7	0.41
Cycles	13	10	0.77
Derived Measures	MIPS	Stack	Stack/MIPS
Accesses/Cycle	1.31	0.70	0.53
Cycles/Instruction	1.00	1.25	1.25
Instructions/Word	1.00	1.33	1.33

Table 7.9: Triangular Recursive Code Comparison

MIPS		Gullwing	
move	2	2	DUP
addu, addiu	4	2	+
load imm	0	1	LIT
sw	2	0	A!, etc...
lw	2	0	A@, etc...
beq	1	1	JMP0
jal	1	1	CALL
j	1	1	RET

Table 7.10: Recursive Dynamic Instruction Mix

### 7.3.5 Tail-recursive Triangular Numbers

Expressing the recursive algorithm in a tail-recursive form reuses the parameters across calls, reducing the data stack depth to that of the iterative case, and provides the opportunity to eliminate the tail call.

The advantage of this representation is that the tail call is trivially eliminated from the Gullwing code by replacing the `CALL` instruction with a `JMP` and eliminating the following `RET` instruction. This eliminates the accumulation of return addresses from the return stack. Optimizing the MIPS code in the same manner yields the initial iterative code. The tail of the Gullwing code is not counted since it executes only once.

Expressing the algorithm in a tail-recursive form brings it closer to the iterative case and so some of the stack manipulation overhead reappears in the Gullwing code. However, the memory bandwidth remains lower than MIPS as in the recursive case.

---

**Algorithm 17** Triangular Tail-Recursive C Source

---

```
int triangular (int foo, int acc) {
    if (foo == 0){
        return acc;
    } else {
        acc += foo;
        foo -= 1;
        return triangular(foo, acc);
    }
}
```

---

---

**Algorithm 18** Triangular Tail-Recursive MIPS32 Assembly

```

Tri: addiu    $sp,$sp,-32
      sw      $31,24($sp)
      beq     $4,$0,$L1
      addu    $5,$5,$4
      jal     Tri
      addiu   $4,$4,-1
$L1:
      lw      $31,24($sp)
      j       $31
      addiu   $sp,$sp,32

```

---



---

**Algorithm 19** Triangular Tail-Recursive Gullwing Assembly

```

Tri: OVER JMP0 OVER + >R
      End
      LIT + R> CALL
      -1
      Tri
      RET
End: >R DROP R> RET

```

---

Main Body	MIPS	Stack	Stack/MIPS
Mem Words	9	6	0.67
Instructions	9	10	1.11
Mem Accesses (instr+data)	(9+2)	7	0.64
Cycles	9	12	1.33
Derived Measures	MIPS	Stack	Stack/MIPS
Accesses/Cycle	1.22	0.58	0.48
Cycles/Instruction	1.00	1.20	1.20
Instructions/Word	1.00	1.67	1.67

Table 7.11: Triangular Tail-Recursive Code Comparison

MIPS		Gullwing	
move	0	4	DUP, OVER, >R, R>
addu, addiu	4	2	+
load imm	0	1	LIT
sw	1	0	A!, etc...
lw	1	0	A@, etc...
beq	1	1	JMP0
jal	1	1	CALL
j	1	1	RET

Table 7.12: Tail-Recursive Dynamic Instruction Mix

### 7.3.6 Subroutine Calls

Nested subroutine calls are really a form of expression evaluation (Section 4.2), which is a process that naturally maps onto a stack. However, the efficiency of this mapping is quite dependent on how the expression is arranged by the programmer or compiler. A fundamental assumption here is that subroutines are library calls and thus cannot be inlined or have their registers globally allocated by a smart compiler.

The MIPS subroutine call overhead, manifested as additional instructions and memory accesses to manage an in-memory stack and move parameters between registers, is quickly amplified as the nesting of subroutines increases. This overhead is so large that even the low density and high CPI of sequence of calls in Gullwing code does not negate it. Section 9.2.1 discusses a method which could eliminate the stack management overhead of MIPS.

**Add2** The stack code cycle overhead here comes from (the unoptimized) RET requiring two cycles to execute.

---

#### Algorithm 20 Add2 C Source

---

```
int add2 (int a, int b){
    return a + b;
}
```

---



---

#### Algorithm 21 Add2 MIPS32 Assembly

---

```
add2: j      $31
      addu   $2, $4, $5
```

---



---

#### Algorithm 22 Add2 Gullwing Assembly

---

```
add2: + RET
```

---

Entire Code	MIPS	Stack	Stack/MIPS
Mem Words	2	1	0.50
Instructions	2	2	1.00
Mem Accesses (instr+data)	(2+0)	2	1.00
Cycles	2	3	1.50
Derived Measures	MIPS	Stack	Stack/MIPS
Accesses/Cycle	1.00	0.67	0.67
Cycles/Instruction	1.00	1.50	1.50
Instructions/Word	1.00	2.00	2.00

Table 7.13: Add2 Code Comparison

MIPS		Gullwing	
addu	1	1	+
j	1	1	RET

Table 7.14: Add2 Dynamic Instruction Mix

**Add3** The stack code shows a great advantage, as previously seen in the recursive triangular number example, even for a single nested call with one additional parameter.

---

**Algorithm 23** Add3 C Source

---

```
int add3 (int a, int b, int c){
    return add2(add2(a,b),c);
}
```

---



---

**Algorithm 24** Add3 MIPS32 Assembly

---

```
add3: addiu    $sp,$sp,-32
      sw      $31,28($sp)
      sw      $16,24($sp)
      move    $16,$6
      jal     add2
      move    $4,$2
      move    $5,$16
      jal     add2
      lw      $31,28($sp)
      lw      $16,24($sp)
      j       $31
      addiu   $sp,$sp,32
```

---



---

**Algorithm 25** Add3 Gullwing Assembly

---

```
add3: CALL
      add2
      CALL
      add2
      RET
```

---

MIPS		Gullwing	
move	3	0	DUP, etc...
addu, addiu	2	0	+
sw	2	0	A!, etc...
lw	2	0	A@, etc...
jal	2	2	CALL
j	1	1	RET

Table 7.15: Add3 Dynamic Instruction Mix

Entire Code	MIPS	Stack	Stack/MIPS
Mem Words	12	5	0.42
Instructions	12	3	0.25
Mem Accesses (instr+data)	(12+4)	6	0.38
Cycles	12	6	0.50
Derived Measures	MIPS	Stack	Stack/MIPS
Accesses/Cycle	1.33	1.00	0.75
Cycles/Instruction	1.00	2.00	2.00
Instructions/Word	1.00	0.60	0.60

Table 7.16: Add3 Code Comparison

**Add4** This more complex example requires some stack manipulation. Nonetheless, its performance is still far better than the corresponding MIPS code.

---

**Algorithm 26** Add4 C Source

---

```
int add4 (int a, int b, int c, int d){
    return add2(add2(a,b),add2(c,d));
}
```

---



---

**Algorithm 27** Add4 MIPS32 Assembly

---

```
add4: addiu    $sp,$sp,-40
      sw      $31,36($sp)
      sw      $18,32($sp)
      sw      $17,28($sp)
      sw      $16,24($sp)
      move    $16,$6
      move    $17,$7
      jal     add2
      move    $18,$2
      move    $4,$16
      move    $5,$17
      jal     add2
      move    $4,$18
      move    $5,$2
      jal     add2
      lw      $31,36($sp)
      lw      $18,32($sp)
      lw      $17,28($sp)
      lw      $16,24($sp)
      j       $31
      addiu   $sp,$sp,40
```

---



---

**Algorithm 28** Add4 Gullwing Assembly

---

```
add4: CALL
      add2
      >R CALL
      add2
      R> CALL
      add2
      RET
```

---

MIPS		Gullwing	
move	7	2	>R, R>
addiu	2	0	+
sw	4	0	A!, etc...
lw	4	0	A@, etc...
jal	3	3	CALL
j	1	1	RET

Table 7.17: Add4 Dynamic Instruction Mix

Entire Code	MIPS	Stack	Stack/MIPS
Mem Words	21	7	0.33
Instructions	21	6	0.29
Mem Accesses (instr+data)	(21+8)	8	0.28
Cycles	21	10	0.48
Derived Measures	MIPS	Stack	Stack/MIPS
Accesses/Cycle	1.38	0.80	0.56
Cycles/Instruction	1.00	1.67	1.67
Instructions/Word	1.00	0.86	0.86

Table 7.18: Add4 Code Comparison

## 7.4 Pipelining

The unpipelined view of the Gullwing processor presented in Chapter 6 shows a simple computer with a number of components comparable to the DLX processor of Hennessy & Patterson. However, the cycle time of Gullwing as shown must be greater than that of the DLX simply because an instruction cycle includes the full path from the Instruction Shift Register (ISR), through the (implicit) decoding logic, to the controlled units such as the ALU.

Pipelining Gullwing would overlap the decoding and executing of an instruction and reduce the cycle time to that of the slowest stage, which is usually the adder in the ALU. I'll show this change in the same manner as Koopman [Koo90], as a transformation of the well-known DLX pipeline, but in much more detail. The resulting Gullwing pipeline has a structure that implies a comparable cycle time to the DLX pipeline.

### 7.4.1 Transforming the DLX Pipeline to Gullwing

For reference, Figure 7.1 [HP02, Fig. A-3] shows the basic shape of the DLX pipeline. Each stage contains a particular subsystem: Instruction Memory (IM), Register File (Reg), Arithmetic and Logic Unit (ALU), and Data Memory (DM). These stages are commonly referred to as the Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Write-Back (WB) stages. Stages are separated by pipeline registers. The Register file is simultaneously read in the ID stage and written in the WB stage.

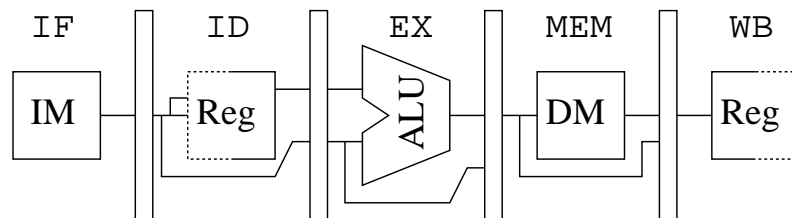


Figure 7.1: DLX Pipeline Block Diagram

The pipelining of the Gullwing processor can be explained by exchanging the subsystems in the DLX pipelining with those of Gullwing and following the implications.

The first change is that Gullwing has a single memory bus for instructions and data. This removes the distinction between IM and DM. Since the IF stage fetches an instruction every cycle, this implies a structural hazard for every data access. However, the zero-operand instruction format of Gullwing means that several instructions can be packed as a group into a single memory word (Section 6.2.7). This reduces the occurrence of the structural hazard to between 2 and 4 percent of executed instructions (PC@ count in Appendix B.2.2). Furthermore, the end of a group of instructions is explicitly marked by the inclusion of a PC@ (PC Fetch) instruction which fetches the next group of instructions in parallel with the current instruction if possible. Combined, these two features accomplish the function of the IF stage and effectively divides it between MEM, where the instructions are fetched, and ID, where they are held and decoded.

The second change is the replacement of the register file with a stack, which has the disadvantage of forcing a RAW (Read-After-Write) dependency between all instructions. This means that an instruction in ID must wait for the previous instruction in EX to reach WB before being able to continue. However, a stack has the advantage of having its inputs and outputs immediately available, without having to decode addresses<sup>13</sup>. This effectively makes them into registers equivalent to the ID/EX and EX/MEM pipeline registers connected to the ALU. Thus the stack can be moved out of ID and placed into EX without any speed penalty. This eliminates the WB stage and simplifies ID.

The third change is the use of direct addressing. The DLX processor uses displacement addressing to simulate a number of other addressing modes. This requires the ALU to compute an address for each load or store instruction, forcing EX to precede MEM. Direct addressing removes this dependency and so both stages can now operate in parallel. Since MEM already contains the incrementer for the Program Counter (brought over from IF in the first transformation), it can be re-used to implement post-incrementing direct addressing.

The end result of these changes results in the pipeline shown in Figure 7.2a, where ID decodes the instructions from the ISR, and EX and MEM execute the instructions. Figure 7.2b shows how these stages map to the existing Gullwing functional blocks. Note that the EX and MEM stages both contain adding circuitry and so place a lower limit on the cycle time that is comparable to that of the DLX EX stage.

The operation of the pipeline is similar to that of the DLX (Figure 7.3). Since the pipeline introduces one stage of latency to execution, the next group of instructions is loaded into the ISR while the last instruction in the current group (5) decodes. This process is detailed in the next section. Instructions that require two cycles to execute, such as loads, occupy the ID stage for the duration of their execution (Figure 7.4). Loads and stores must take two cycles since they currently cannot be overlapped, but on the other hand there is no load delay slot. Overlapping loads and stores are discussed in Section 9.2.3.

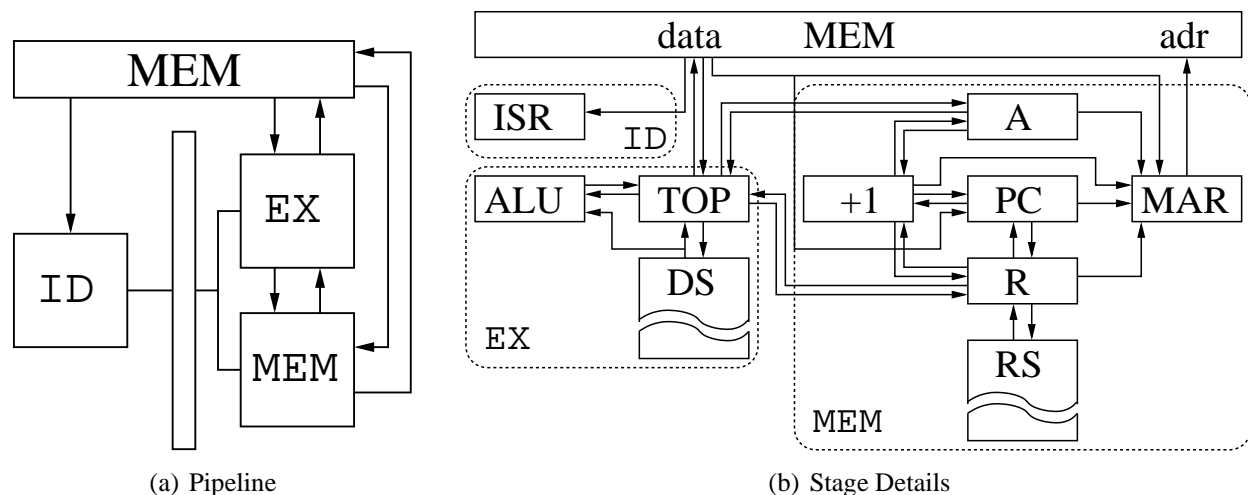


Figure 7.2: Gullwing Pipeline Block Diagram

<sup>13</sup>Since a stack is only ever accessed sequentially, the addressing of the individual stack registers reduces to a single-bit shift register, one bit per stack register, with no decoding required. A more aggressive design would further reduce the entire stack to a word-wide shift register.

Instruction	Cycle (Current Instruction Word)						1
	1	2	3	4	5	6	
0	ID	EX/MEM					
1		ID	EX/MEM				
2			ID	EX/MEM			
3				ID	EX/MEM		
4					ID	EX/MEM	
5	(next word loaded while 4 executes and 5 decodes)					ID	EX/MEM
0							ID

Figure 7.3: Gullwing Pipeline Operation

Instruction	Cycle				Notes
	4	5	6	7	
@A	ID	MEM			@A stays in ID
@A		ID	EX/MEM		
4			ID	EX/MEM	
5				ID	

Figure 7.4: Gullwing Load/Stores Pipeline Diagram

## 7.4.2 Altering the ISR to Deal with the Additional Latency

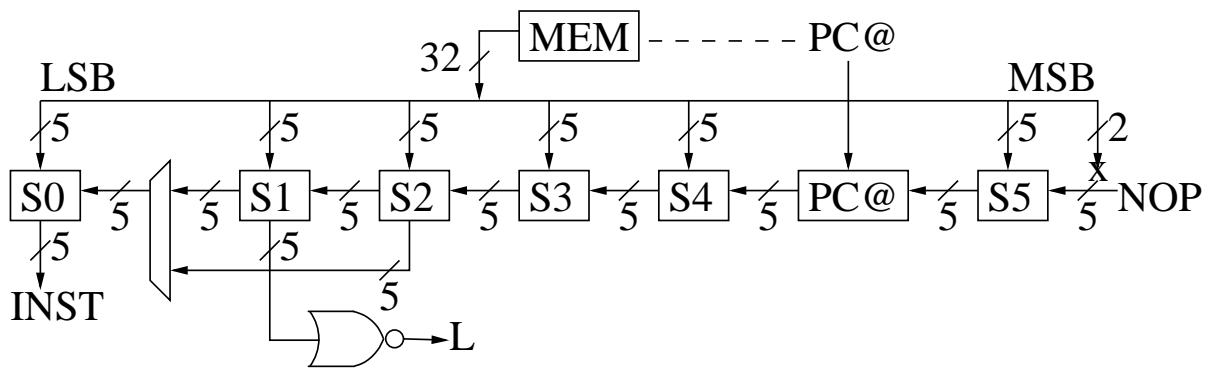
The pipelining of Gullwing adds a latency of one stage to the execution of instructions. This affects the use of the PC@ instruction to fetch the next group of instructions when the current one is exhausted (Section 6.1.3). After the last instruction in a group has finished, while the current PC@ is in the EX stage, another PC@ would be in ID and would enter EX just as the ISR was reloaded by the first PC@. The spurious second PC@ would then load the ISR again after only the first instruction from the new group had begun executing, skipping over the remainder.

The solution to this side-effect of pipelining is to move ahead by one the insertion point of the PC@ instruction so that it begins executing while the actual last instruction in the group begins decoding and is executed just as the ISR is reloaded. Figure 7.5 shows the necessary alterations to the original ISR (Figure 6.2). The PC@ instruction is inserted at the same time as the rest of the ISR is loaded, in between the last and before-last instruction slots (S5 and S4). The slots are now filled-in with NOPs instead of PC@s as the instructions are shifted out.

The multiplexer between S0 and S1 is required to handle the instruction fetch overlap optimization (Section 6.3.1). Once the penultimate instruction (originally in S4) reaches S0, the

inserted PC@ will be in S1 and enable the L signal, signalling the last instruction. If the instruction in S0 does not access memory, then both it and the PC@ execute in parallel and S0 and S1 are then loaded from S2<sup>14</sup>, which contains the actual last instruction (Figure 7.6). If the instruction in S0 accesses memory, then the instruction fetch is not overlapped, and the instructions are shifted as usual (Figure 7.7).

If not all the instruction slots are filled, then the compiler must make sure that a PC@ is compiled before the last instruction and fill the remaining slots with NOPs. It must also make sure the built-in PC@ is never executed after a previous PC@ in the same word. For example, if the last instruction would end up in S4, then it must be moved to S5 and a NOP placed in S4. Such instruction reorderings usually leads to a one-cycle penalty in execution.



(a) ISR for Pipeline

Figure 7.5: Gullwing ISR Modified for Pipeline

Instruction	Cycle				Notes
	4	5	6	1	
3	ID	EX/MEM			
4		ID	EX		Doesn't use MEM
PC@		(L is set)	MEM		Executed concurrently
5			ID	EX/MEM	Loaded from S2
0				ID	

Figure 7.6: Gullwing Instruction Fetch (with Overlap) Pipeline Diagram

<sup>14</sup>S1 is overwritten by S2 so as to prevent two PC@ from being executed in sequence. The instruction in S2, being the actual last instruction, can never be a PC@, else the situation described in the first paragraph occurs.

Instruction	Cycle					Notes
	4	5	6	7	1	
3	ID	EX/MEM				
4		ID	MEM			Uses MEM
PC@		(L is set)	ID	MEM		Executed sequentially
5				ID	EX/MEM	Loaded from S1
0					ID	

Figure 7.7: Gullwing Instruction Fetch (without Overlap) Pipeline Diagram

### 7.4.3 The Effect of Pipelining on Calls, Jumps, and the CPI

Jumps behave on a pipelined Gullwing in the same manner as on the DLX. The jump target is loaded one cycle after the jump instruction has finished executing, thus the following instruction is a branch delay slot which is always executed and must be appropriately filled by the compiler (Figure 7.8).

Also, as in the DLX, a data hazard occurs if a conditional jump depends on the result of the immediately preceding instruction. However, because the stack provides only a single point for all results it makes this data hazard inevitable. As with the branch delay slot, the compiler must find a way to fill this data hazard slot with useful work. One example is to fill it with a DUP instruction which would duplicate the top of the stack before the conditional jump consumes it, thus saving its value.

In the worst case where they can only be filled with NOPs, the data hazard and branch delay slots will add two cycles to conditional jumps, raising them to four cycles, and will add one cycle to calls, unconditional jumps, and returns, raising them to three cycles. Factoring this overhead into the CPI data from Table 7.4 increases the CPI contribution of Subroutine instructions to 0.450 for Extensions and 0.453 for VM, of Conditionals to 0.142 and 0.062, for a new total CPI of 1.533 and 1.481 respectively. This estimate does not take into account the second-order effect of the lower instruction density caused by the NOP-filled slots, which will increase the proportion of instruction fetches.

Instruction	Cycle					Notes
	3	4	5	6	1	
2	ID	EX/MEM				Data Hazard Slot
JMP0		ID	MEM			JMP0 stays in ID
JMP0			ID	MEM		
4				ID	EX/MEM	Branch Delay Slot
0					ID	Branch Target

Figure 7.8: Gullwing Taken Jumps or Calls Pipeline Diagram

## 7.5 Summary and Performance Comparison

In Section 7.2, the comparison of benchmark statistics revealed these facts about Gullwing, relative to DLX/MIPS:

- A greater number of literal fetches, subroutine calls, and stack permutations are executed.
- An average CPI of 1.31, which is poor compared to the average of 1.14 for a DLX. However, Gullwing is actually architecturally equivalent to a DLX without load delay slots or delayed branches, whose average CPI is 1.38.
- An average number of memory accesses per cycle of 0.667, compared to 1.421 for DLX/MIPS (Section 7.2.3).
- An average code density of only 1.2 instructions per memory word, out of a potential maximum of three, because most of the instruction slots remain unused.

In Section 7.3, a detailed inspection and analysis of equivalent programs which express fundamental code features uncovered these differences between Gullwing and a generic MIPS-32 computer:

- The random-access registers and explicit operands of the MIPS design are a definite advantage when multiple values must be maintained at once in an algorithm. Gullwing must instead execute additional instructions to manipulate the stacks to get to the value it needs.
- The MIPS processor must simulate a stack in software for subroutine calls and recursive procedures. The extra instructions to implement this stack consume more memory bandwidth and processor cycles than the equivalent Gullwing code.
- The Gullwing processor requires less memory bandwidth, often half that of MIPS, regardless of the number of cycles required for an algorithm.

In Section 7.4, the pipelined form of Gullwing is derived through incremental transformation of the DLX pipeline. The result is a 2-stage pipeline composed of an Instruction Decode stage followed by parallel Execute and Memory stages. Each stage is structurally no more complex than any stage from the DLX pipeline, which implies that the cycle time will be similar. The Gullwing pipeline exhibits similar branch data hazard and delay slots as the DLX pipeline. In the worst case, these delays should increase the average CPI from 1.31 to 1.51.

In summary, a pipelined Gullwing processor would have a similar cycle time relative to a DLX processor. However, with the exception of subroutine calls, Gullwing usually requires a greater number of cycles to execute the same algorithms. It also suffers from a higher CPI due to un-optimized load and branch delays. Therefore, to bring Gullwing up to the same performance as DLX, the number of executed instructions and/or the average CPI must be reduced (Sections 9.2.2 and 9.2.3).

# Chapter 8

## Improving Code Density

The density of Gullwing low-level code is very good. A 32-bit word holds six instruction slots. However, instructions that require an in-line argument such as calls, jumps, and literal fetches, also additionally use up one whole subsequent memory word<sup>1</sup>. Including one such instruction in a group, while keeping all slots filled, raises the memory usage to two words and thus halves the code density to three. Adding another such instruction drops the density to two, and so on until all six instructions require an in-line argument, with a resulting minimum code density of six-sevenths ( $\sim 0.86$ ). As the number of slots in a memory word increases, the minimum code density increases towards unity.

This situation is unfortunately a narrow best case. It is only applicable when all instruction slots can be filled. This is true for literal fetches since they do not alter the program flow, and for conditional jumps since they simply continue with the next instruction if the condition is not met. Calls and unconditional jumps always load a new instruction group<sup>2</sup>. Memory is word-addressed and groups always begin execution with the first slot, therefore jumping or returning to the instructions in the slots after a call or jump is impossible and must instead go to the next memory word after the argument. Sequences of calls thus end up wasting most of the available slots (Figure 8.1), bringing the minimum code density down to one-half.

Sequences of calls are typical of high-level code, where a procedure is primarily composed of calls to other procedures. Sequences of jumps are rare, never executed sequentially, and are not considered further. The actual usage of the instruction slots is listed in Appendix B.1.5. The low ratio of filled instruction slots suggests that there is room for significant improvement in code density.

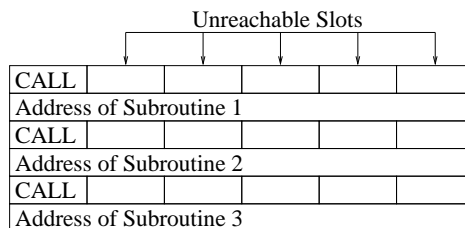


Figure 8.1: Gullwing High-Level Code with Unavailable Slots

<sup>1</sup>Returns take their argument from the Return Stack and so require no extra memory.

<sup>2</sup>As do returns.

## 8.1 Improving High-Level Code Density by Adding an Instruction Stack

The key to improving the density of high-level code is the observation that if the instructions to be executed after a subroutine call are placed in the same memory word as the call, they will be fetched along with the call and they should not need to be fetched again when the subroutine returns. The instructions simply need to be temporarily stored in the processor while the subroutine executes. The number of words that need to be stored is identical to the current nesting depth of the program. This suggests extending the Instruction Shift Register (ISR) with a stack that operates in synchrony with the Return Stack (RS). Figure 8.2 illustrates the process. For clarity, only four slots are depicted and the Return Register (R) is omitted (see Figure 6.1).

When a subroutine call is executed, the remaining instructions are pushed onto an Instruction Stack (IS) at the same time that the return address is pushed from the Program Counter (PC) onto the RS. When the subroutine returns, the saved instructions are popped from the IS and placed at the head of the ISR at the same time that the return address is popped from the RS into the PC. The last slot in the ISR is filled with a Program Counter Fetch (PC@) instruction, as during normal execution. The net effect is that the instructions following a call are now executed upon return from the subroutine. This makes it possible to increase the minimum code density of high-level code back to six-sevenths (Figure 8.3).

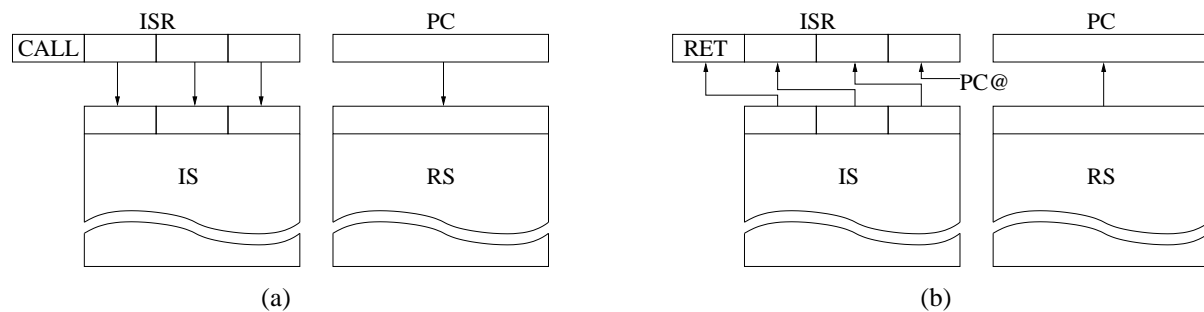


Figure 8.2: Instruction Stack During Call and Return

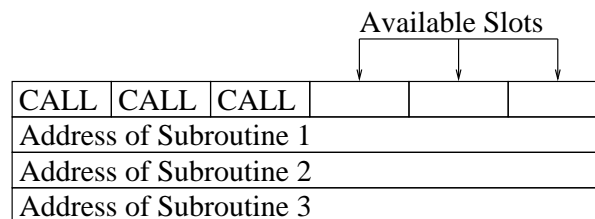


Figure 8.3: Gullwing High-Level Code with Available Slots

### 8.1.1 Side-Effects on Return Stack Manipulation

A consequence of the Instruction Stack (IS) is that the return addresses on the Return Stack (RS) must always be matched with the corresponding instructions stored on the IS. Any offset between the Instruction and Return stacks would mean that from that point onwards all returns to calling procedures would execute a few random instructions before fetching the next (correct) groups of instructions!

The >R (“To R”) and R> (“R From”) instructions move data back and forth between the top of the Data Stack (TOP) and the RS. Hence they could leave the RS with a different number of elements than the IS. To compensate, >R pushes a group of PC@ instructions onto the IS, and R> pops the IS, discarding the instructions. If >R is used to push a return address onto the RS, such as when the address of a function call is computed at runtime, then the next subroutine return will execute the stored PC@ and forcibly fetch the first instruction group of that procedure. If R> is used to discard a return address, possibly for some kinds of error handling, the stored instructions for that procedure are also discarded. Figure 8.4 illustrates the process.

It is still possible to cause incorrect execution with the unusual code sequence 'R> >R' which would replace the stored instructions with a PC@. This would skip a few instructions upon return unless the instruction slots after the corresponding call we deliberately left unused (and thus filled with PC@ anyway). However, the need to inspect or alter the address of the calling procedure's caller is rather unusual. Similarly, unless there exists a means of loading or storing the contents of the IS under program control, the RS cannot be saved to memory for the purpose of context switching, debugging, or exception handling.

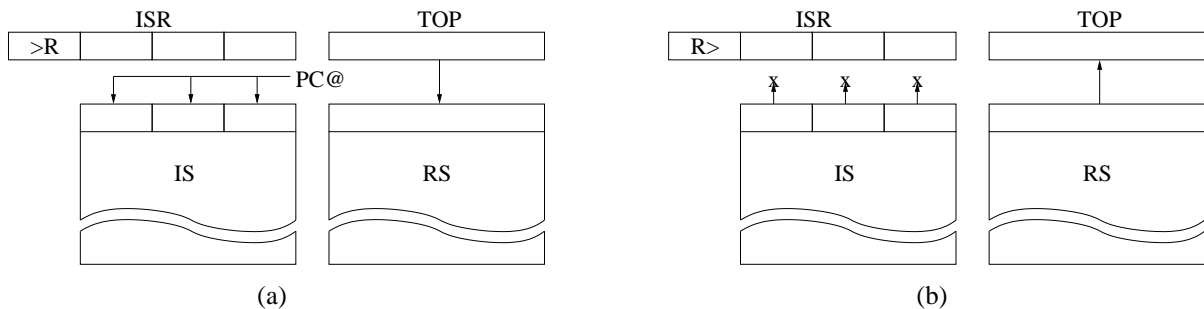


Figure 8.4: Instruction Stack During >R and R>

## 8.2 Implementation

The implementation of the code density optimization is straightforward, consisting mainly of adding control lines to push and pop the IS as required. Algorithm 29 shows the changes required. Added controls are in bold, while removed controls are struck through. The instruction fetch overlap optimizations (Section 6.3.1) are not included as they are orthogonal.

**Algorithm 29** Alterations to Gullwing to Support an Instruction Stack

Inputs		Outputs			
INST	TOP	S	N	Control	
CALL	X	0	1	PC $\rightarrow$ (+1) $\rightarrow$ R, R $\rightarrow$ RS, RS(PUSH), MEM $\rightarrow$ MAR, PC, <b>ISR<math>\rightarrow</math>IS, IS(PUSH)</b>	
CALL	X	1	0	PC $\rightarrow$ (+1) $\rightarrow$ PC, MAR, MEM $\rightarrow$ ISR	
RET	X	0	<b>0</b>	RS(POP), RS $\rightarrow$ R, R $\rightarrow$ PC, MAR, <b>IS(POP), IS<math>\rightarrow</math>ISR</b>	
<del>RET</del>	<del>X</del>	<del>1</del>	<del>0</del>	<del>PC<math>\rightarrow</math>(+1)<math>\rightarrow</math>PC, MAR, MEM<math>\rightarrow</math>ISR</del>	
>R	X	0	0	DS(POP), DS $\rightarrow$ TOP, TOP $\rightarrow$ R, R $\rightarrow$ RS, RS(PUSH), <b>PC<math>\rightarrow</math>IS, IS(PUSH),</b> ISR<<	
R>	X	0	0	RS(POP), RS $\rightarrow$ R, R $\rightarrow$ TOP, TOP $\rightarrow$ DS, DS(PUSH), <b>IS(POP),</b> ISR<<	

## 8.3 Side-Effect on Code Size, Silicon Area, and Subroutine Overhead

**Code Size** By allowing sequences of calls to fill all instruction slots in a memory word, the Instruction Stack (IS) mechanism significantly reduces the size of high-level code. In the example previously given, the code size is reduced by 33% (from six memory words down to four). In the most extreme case where all six slots are filled with calls, the code size is reduced by 42% (from twelve memory words down to seven). For a large enough number of slots per memory word ( $N$ ), the reduction tends towards a limit of 50% as the number of memory words goes from  $2N$  down to  $N + 1$ .

As a further estimate of the reduction in code size, if the number of instructions in the VM test (Appendix B.1.5) is assumed to be evenly packed into all six instruction slots in a word, these instructions will then use only 610 words of memory instead of 1565 (Appendix B.1.1). Adding to that the 1464 memory words which hold literals and addresses, which are not affected by the new instruction packing, the new total size of the VM test would be 2074 memory words, which is a 31.5% reduction in size.

In reality, the size reduction is lessened by a second-order effect from calls and jumps: since memory is word-addressed, the target code of a call or jump must begin at the first slot of a memory word, which means that there will usually be a break in the sequence of used instruction slots, wasting a few. This typically happens in small code loops and at the end of procedures (after the return instruction). This effect is also discussed in Section 6.2.7.

**Silicon Area** Adding a third stack increases the silicon area required by the processor. The additional area consumed by the IS is similar to that of the Return Stack since it must be of the same depth and slightly less wide.

The breakeven point between the additional area of the IS and the saved area in main memory, given a uniform word width, is when the reduction  $R$  in the original size  $S$  of a program, due to the addition of the IS mechanism, is equal to the depth  $D$  of the IS stack:  $S - S(1 - R) = D$ .

For example, if  $R$  is taken to be the previously determined value of 31.5%, then  $S$  needs to be 3.17 times larger than  $D$  for its size to be reduced by the same amount as the size of the IS. Since a stack rarely needs to be more than 32 words deep (Section 5.5),  $S$  would only need to be equal to 102 memory words to justify the additional area of the IS mechanism.

For larger programs, the reduction in code size would greatly outweigh the area of the IS. The size of the main memory can thus be correspondingly reduced<sup>3</sup>. The lowered total silicon area (processor and memory) would especially benefit embedded systems.

**Subroutine Overhead** The use of the IS eliminates the need to fetch the remaining instruction slots after a call instruction. Algorithm 29 shows that the load in the eliminated second cycle of the return instruction is no longer required because the remaining instruction slots are now loaded from the IS during the first cycle. This reduces the overhead of calling and returning from a subroutine to three cycles, down from four, and also reduces the associated memory traffic by the same 25%.

### 8.3.1 The Instruction Stack as an Instruction Cache

Section 7.2.4.1 discusses how the packing of multiple instructions in a memory word makes each memory word into a cache line of sorts. This caching effect reduces the number of sequential instruction fetches within a basic block, as performed by PC@ (“PC Fetch”) instructions, leaving virtually all loading of instructions to calls and jumps.

With the addition of the Instruction Stack, multiple basic blocks, or at least fragments thereof, can fit into a single memory word. This extends the caching effect across basic blocks separated by calls. The basic block fragment following a call instruction had already been previously loaded and kept on the IS while the subroutine was executing.

---

<sup>3</sup>This assumes that the stacks and main memory are implemented using the same memory technology.



## Chapter 9

# Conclusions, Contributions, and Further Work

The first part of this thesis presented the historical origins of the first generation of stack computers and found that these machines were derived from Bauer's stack principle for the allocation of storage in nested subroutines, later used in the specification of ALGOL and now seen as the call stack of C and other programming languages, and that the second generation of stack computers was based on Hamblin's independently discovered stack principle geared at function composition instead of storage allocation. The English Electric KDF9, made commercially available around 1963, was found to stand out as the first second-generation stack computer and the only one until the NOVIX NC4016 in 1985. This gap, and the coincidence with the appearance of RISC microprocessors, accounts for the obscurity of the second generation.

The second part of this thesis built upon the first by proposing a set of criteria to distinguish first and second-generation stack computers. In summary, second generation stack computers keep their stacks in the CPU instead of main memory, use the stacks for the evaluation and composition of functions instead of procedure storage allocation, and use simple, RISC-like instructions instead of complex microcoded operations. Chapter 5 then presented a rebuttal to the influential arguments against stack architectures cited by Hennessy & Patterson and found that they are not applicable to second-generation stack computers due to their different design and to advances in hardware technology and compilers. The most telling finding is that some modern processors, such as the Intel Pentium IV and the Digital Alpha AXP 21064, use a 16-deep internal stack to cache return addresses in the same manner as second-generation stack computers.

The third part of this thesis specified the design of a small second-generation stack machine, named 'Gullwing'. The first unusual feature found was that the packing of multiple instructions per memory word, made possible by the zero-operand instruction format, reduced the number of sequential instruction fetches to between 8.3% and 9.7% of the total number of executed instructions despite achieving an average code density of only 1.2 instructions per memory word. An additional simple optimization of the instruction fetch mechanism reduced this fetching overhead by 50.6 to 77.3%, down to 4.1 to 2.2% of the total number of executed instructions, effectively eliminating the need for a second memory bus dedicated to fetching instructions.

Chapter 7 then compared Gullwing to the DLX and MIPS processors, via some aggregate

benchmarks and some low-level code comparisons. In Section 7.2, it was observed that 23.3 to 29.2% of the executed instructions on Gullwing are stack manipulation instructions whose actions are implicit in the three-operand instruction format of MIPS. Similarly, Gullwing must perform 10 to 16% more immediate loads since there are no operands to hold small constants.

The average CPI of Gullwing was between 1.301 and 1.312: a penalty of 13.1 to 18.2% over DLX. However, Gullwing is architecturally equivalent to a DLX processor without delayed branches and loads. Without these optimizations the CPI of DLX would have been 1.34 to 1.41 instead: 2.1 to 8.4% worse than Gullwing.

It was also found that MIPS performed an average of 1.421 memory accesses per cycle, while Gullwing required only 0.667: a 53% reduction in memory bandwidth. For instruction fetching alone, Gullwing required 55.9% fewer memory accesses per cycle on average (0.441 vs. 1.00 for MIPS). These improvements were found to originate in the packing of multiple instructions per memory word: 71.1 to 80.9% of code basic blocks fit within a single memory word on Gullwing.

Section 7.3 showed that Gullwing is at a disadvantage when multiple intermediate results are required. A comparison of iterative code demonstrated that Gullwing required 25% more memory space, 125% more instructions, 25% more memory accesses, and 150% more cycles, compared to MIPS, to execute the iterative algorithm due to the need to shuffle values on and off the Data Stack.

On the other hand, Gullwing exhibits extremely efficient function calls. The same algorithm, implemented recursively on Gullwing, required 54% less memory space, 38% fewer instructions, 59% fewer memory accesses, and 23% fewer cycles than the equivalent recursive MIPS code due to the elimination of the instructions required to simulate a stack in main memory.

When looking at pure nested subroutines, as would be the case with precompiled libraries, Gullwing's advantage at subroutine calls is amplified further: 58 to 67% less (code) memory space, 71 to 75% fewer instructions, 62 to 72% fewer memory accesses, 50 to 52% fewer cycles, and a 25 to 44% reduction in memory bandwidth, despite an average CPI between 1.67 and 2.00 and a code density between 0.60 and 0.86 instructions per word.

As originally specified, the Gullwing processor was not pipelined, and so its cycle time would have exceeded that of the pipelined DLX processor. A pipelined form of Gullwing was specified by transforming the stages of the DLX pipeline to Gullwing's stack architecture. The result is a two-stage pipeline with parallel EX and MEM stages, which has branch delay slots like DLX, but no load delay slots and additional branch hazard slots due to the use of a stack instead of registers. In the worst case where these slots could not be used productively, the average CPI of Gullwing would increase by 12.9 to 17.8%, to a range of 1.481 to 1.533. Assuming that the ALU is the critical path of a simple pipeline, then the two-stage pipelined form of Gullwing should have a similar cycle time to the five-stage DLX pipeline.

Finally, Chapter 8 proposed the use of a third stack to temporarily hold instructions during subroutine calls. This Instruction Stack would maximize the density of high-level code with many branches and calls, reducing the overall code size by a little over 30% (up to a theoretical limit of 50%), and would reduce the memory traffic and cycle count overhead of calling and returning from a subroutine by 25%.

## 9.1 Contributions

This thesis makes the following contributions:

1. a historical review of first-generation stack computers which uncovers the origins of the conceptual difference between first and second-generation machines (Chapter 2);
2. a historical review of second-generation stack computers which provides a summary of many lesser-known and unpublished machines (Chapter 3);
3. a set of criteria to distinguish first and second-generation stack computers which expand on those given by Feldman and Retter [FR93, pp.599-604] (Chapter 4);
4. a rebuttal of the arguments against stack computers cited by Hennessy and Patterson, showing that they are applicable to the first generation of stack computers, but not the second (Chapter 5);
5. a register-transfer level description of Gullwing, a simple, modern second-generation stack computer, along with an optimization of its instruction fetching mechanism (Chapter 6);
6. an initial comparison of the execution statistics of non-numerical code on both Gullwing and MIPS (Section 7.2);
7. a detailed comparison of the behaviour of iteration, recursion, tail-recursion, and subroutine calls on both Gullwing and MIPS (Section 7.3);
8. the design of the Gullwing pipeline as a transformation of the DLX pipeline (Section 7.4);
9. the proposal of an instruction stack to maximize the code density and accelerate the subroutine returns of Gullwing (Chapter 8).

## 9.2 Further Work

Section 7.5 summarized some comparisons between DLX/MIPS and Gullwing. As Gullwing currently stands, its overall performance is lacklustre when compared to a basic MIPS processor. Although the cycle time should be similar, Gullwing lacks result forwarding between functional units, which makes its pipelining incomplete and inflates its CPI, as evidenced by the two-cycle loads, stores, branches, and calls. The instructions of Gullwing are also simpler than those of MIPS and thus more are required to accomplish the same task, with the exception of subroutine calls, which Gullwing performs with fewer instructions, cycles, and memory accesses than MIPS.

This section presents some future improvements to Gullwing which could bring its CPI closer to the ideal of 1.00, as well as reduce the impact of its higher instruction count. Additionally, the addition of a stack to the MIPS architecture is discussed, in order to grant MIPS the efficient subroutine calls of Gullwing without otherwise altering the instruction set or pipeline.

## 9.2.1 Reducing the DLX/MIPS Subroutine Call Overhead by Adding Stacks

The MIPS architecture has very inefficient subroutine calls compared to Gullwing, stemming from the need to save and restore registers to a stack in memory (Section 7.3.6). Conversely, Gullwing has poor performance for iterative code because it lacks random-access registers. A combination of both might yield the best of both worlds.

Figure 9.1 shows a conceptual modification to the MIPS register file which would create the possibility of running stack-like code for efficient subroutine calls, while otherwise leaving the instruction set, architecture, and pipeline unchanged. A Data Stack and a Return Stack are added 'underneath' some registers. The Return Stack goes under register \$31 since it is where the current return address is normally stored by the `jal` (Jump And Link) instruction. The Data Stack could be placed under any other registers, but is placed here under registers \$2 and \$1 for illustrative purposes. Two registers are used so that two operands can be taken from the stack when needed. Register \$0 is of course a source of zeroes on reads and a sink on writes.

If the stacks are disabled, the register file behaves as usual, and existing code runs unchanged. When the stacks are enabled, a read from a register connected to a stack (\$31, \$2, and \$1) pops the content of the stack into the register, making the read operation destructive, and a write to a register pushes the previous contents of the register onto the stack. The exceptions to this behaviour are when both \$2 and \$1 are read together, only the contents of \$2 are overwritten by the stack, and if \$1 is both read and written in that same instruction, it is simply overwritten as if it had been popped, then pushed.

Given this rough sketch of stack and register interaction, Table 9.1 shows how a number of stack computer instructions can be synthesized by suitable MIPS instructions. Small inline constants and branch target labels are denoted by '\$Ln'.

The stack manipulation instructions, such as `DROP` and `OVER`, should never be required. A compiler would use other registers as usual for the storage of counters and common subexpressions, thus avoiding the stack manipulation overhead of iterative code seen in Section 7.3.3, and avoiding the RAW (Read After Write) dependency of the stack. Conversely, the stack could be used to hold local variables and arguments to subroutines, which would reduce or outright eliminate the loads and stores required for nested subroutine calls on MIPS (Section 7.3.6).

Algorithm 30 shows the recursive example from Section 7.3.4 implemented using stacks. Tables 9.2 and 9.3 show that, relative to the original MIPS code, the loads and stores are eliminated, the algorithm uses 38% fewer cycles, and memory bandwidth is reduced by 24%.

A similar register and stack mechanism was proposed by Richard L. Sites [Sit78], with the primary intent of simplifying expression evaluation. He observed that a stack placed under one of the registers allowed the evaluation of an expression using fewer actual registers. In current systems, this could make possible smaller register files, or give a compiler more room to do register allocation.

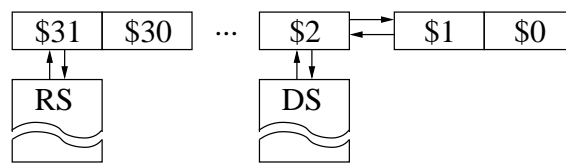


Figure 9.1: MIPS Register File with Stacks

Stack	MIPS	Stack	MIPS
CALL	jal \$Ln	NOT	xor \$1,\$1,-1
RET	j \$31	AND	and \$1,\$1,\$2
JMP	b \$Ln	+	add \$1,\$1,\$2
JMP0	beq \$1,\$0,\$Ln	DROP	add \$0,\$0,\$1
LIT	addi \$1,\$0,\$Ln	SWAP	add \$1,\$2,\$0
@	lw \$1,(\$30)	PUSH	add \$1,\$30,\$0
!	sw \$1,(\$30)	POP	add \$30,\$1,\$0

Table 9.1: Synthesized Stack Operations on MIPS with Stacks

**Algorithm 30** Triangular Recursive MIPS32  
Assembly with Stacks Added

```

Tri: add  $30,$1,$0 // POP
      beq  $30,$0,$L1 // JMP0
      add  $1,$30,$0 // PUSH
      addi $1,$0,-1 // LIT
      jal  Tri // CALL
      add  $1,$1,$2 // +
      add  $1,$1,$2 // +
$L1: j    $31 // RET

```

Instructions	Without	With
move	2	0
addu, addiu	4	5
sw	2	0
lw	2	0
beq	1	1
jal	1	1
j	1	1

Table 9.2: Recursive MIPS32 Instruction Distribution With and Without Stacks

Entire Code	Without	With	With/Without
Mem Words	13	8	0.62
Instructions	13	8	0.62
Mem Accesses (instr+data)	(13+4)	8	0.62
Cycles	13	8	0.62
Derived Measures	Without	With	With/Without
Accesses/Cycle	1.31	1.00	0.76
Cycles/Instruction	1.00	1.00	1.00
Instructions/Word	1.00	1.00	1.00

Table 9.3: Triangular Recursive MIPS32 Code Comparison With and Without Stacks

## 9.2.2 Reducing Gullwing's Instruction Count with Compound Stack Operations

There has been a hidden assumption throughout this thesis about which operations can be performed on a stack while avoiding random access to the elements it contains. The commonly given model of stack behaviour assumes that a dyadic operation pops two items from the stack in sequence, stores them in some working registers, operates upon the items, and then pushes the (usually single-item) result onto the stack.

This process is usually optimized to a single step by making the visible top of the stack into a separate register, thus allowing two items to be read simultaneously and the result to be stored into the register while the stack proper simply discards its topmost element. This mechanism is illustrated by the Data Stack in Figure 6.1.

However, the capacity to see the *two* topmost items creates the possibility of more complex operations *without adding cycles, data lines, or control lines*, and thus without most of the pitfalls of past, high-level instruction sets.

For example, addition on a stack destroys both its arguments, which brings about the use of the OVER instruction to preserve the second element by pushing onto the stack a copy of the second element which is then added to the first. This is seen in the iterative and tail-recursive code examples (Sections 7.3.3 and 7.3.5). Similarly, both arguments can be preserved by first copying them both with two consecutive OVER instructions.

By simply not popping the stack proper when performing an addition, the 'OVER +' sequence is accomplished in a single instruction. Similarly, by merging the addition operation into the implementation of the OVER instruction, the sequence 'OVER OVER +' reduces to a single instruction. The Gullwing implementation of these new operations is shown by the first two lines of Algorithm 31. Removed actions are struck out, while added ones are bolded. Similarly, the last three lines show the implementation of a non-destructive JMP0 instruction, replacing the common 'DUP JMP0' sequence, where the stack is simply not manipulated.

To provide enough opcodes to support these compound operations, the width of the Gullwing opcodes would have to be increased to six bits. While this is still comparable to MIPS, further study is needed to make sure that at least one compound operation would occur on average per executed group of instructions, thus balancing out the reduction from six to five of the number of instruction per group, and still saving one cycle compared to the execution of the original instruction group.

---

**Algorithm 31** Gullwing Compound Stack Operations

---

Inputs	Outputs			
INST	TOP	S	N	Control
OVER_+	X	0	0	TOP, DS → ALU(+) → TOP, <del>DS(POP)</del> , ISR < <
OVER_OVER_+	X	0	0	<b>TOP</b> , DS → <b>ALU(+)</b> → TOP, TOP → DS, DS(PUSH), ISR < <
DUP_JMP0	=0	0	1	<del>DS → TOP, DS(POP)</del> , MEM → PC, MAR
DUP_JMP0	! =0	0	0	<del>DS → TOP, DS(POP)</del> , PC → (+1) → PC, MAR, ISR < <
DUP_JMP0	X	1	0	PC@

---

### 9.2.3 Reducing Gullwing's CPI by Executing Multiple Instructions using Generalized Instruction Folding

Section 6.3.1 describes a change to the decoding of instructions which allows the overlapping of the execution of the PC@ ('PC Fetch') instruction with that of any instruction that does not access memory, retiring both instructions at once. This mechanism could be generalized to a greater number of pairs of instructions.

For example, the 'OVER +' and 'DUP JMP0' compound operations described in Section 9.2.2 could be implemented, without adding new opcodes, by decoding both the current and the next instruction and then shifting them both out. This feature widens the instruction input to the decoder and might present a time or area penalty since the instruction encoding is dense<sup>1</sup>.

Loads and stores can especially benefit from the decoding of two instructions at once. Instead of holding steady for two cycles the instruction input to the decoder while the load or store executes its two phases, the first phase can be executed when the instruction enters the first part of the two-instruction window of the improved decoder, and the second phase when the instruction enters the second part of the window. Algorithm 32 shows how the instruction sequence 'DROP A@+ R@+ XOR RET'<sup>2</sup> would be executed: The first line shows the first phase of A@+ folded with the execution of DROP. The second line contains the overlapped second phase of A@+ and first phase of R@+. The third line does not overlap the execution of XOR with the second phase of @R+ since the load must complete before XOR can operate on the loaded value. Finally, the fourth and fifth lines show how XOR and RET are overlapped.

The execution of this five-instruction code sequence now takes five cycles instead of eight. Its CPI went from 1.60 to 1.00. The number of memory accesses per cycle increased from 0.75 to 1.00. The performance is now on par with that of non-branching code on MIPS.

The data dependency of the '@R+ XOR' sequence could be avoided by enabling the ALU to use MEM as an input. Similarly, the 'LIT JMP0' sequence could be overlapped if an additional zero-detect circuit was connected to MEM, and '>A @A' would benefit from a direct path between TOP and MAR. This forwarding of data, equivalent to the result forwarding of the MIPS processor, does add multiplexers in the data path and thus the cycle time to cycle count trade-off should be considered carefully.

**Algorithm 32** Example Gullwing Instruction Sequence Using Generalized Folding

Inputs		Outputs			
S0	S1	TOP	S	N	Control
DROP	@A+	X	0	0	DS→TOP, DS(POP), A→MAR, (+1)→A, ISR<<
@A+	@R+	X	0	0	MEM→TOP, TOP→DS, DS(PUSH), R→MAR, (+1)→R, ISR<<
@R+	XOR	X	0	0	MEM→TOP, TOP→DS, DS(PUSH), PC→MAR, ISR<<
XOR	RET	X	0	0	TOP, DS→ALU(XOR)→TOP, DS(POP), \\ cont. next line RS(POP), RS→R, R→PC, MAR, ISR<<
RET	PC@	X	0	0	PC→(+1)→PC, MAR, MEM→ISR

<sup>1</sup>By comparison, MIPS R-type instructions use a total of twelve opcode bits, but do not encode 2<sup>12</sup> unique instructions. The encoding must thus be sparse and simple to decode.

<sup>2</sup>A simple hypothetical comparison routine based on COMPARE\_STRING from Appendix A.1.3.



# Appendix A

## Gullwing Benchmarks Source

The appendix provides the source code for the benchmarks described in Section 7.1.

### A.1 Flight Language Kernel

The Flight language kernel defines the basic functions needed to define, lookup, and execute functions. These basic functions enable the system to extend itself without external software. The kernel is written in Gullwing machine language, described here symbolically. The main loop of the kernel is described in Section A.1.9. I have added comments, but they are not present in the actual source code.

#### A.1.1 Internal Variables and Memory Map

These variables contain the state of the kernel. Each is a single memory word. They define the boundaries of areas of memory illustrated in Figure A.1.

**HERE** Contains the address of the memory location that is the current target for compilation.

**HERE\_NEXT** Contains the address of the next memory word where code is to be compiled.

**SLOT** Contains the bitmask which defines the current available instruction slot in the memory word pointed-to by **HERE**.

**THERE** Contains the address of the top of the function name dictionary. Also the pointer to the bottom of the input buffer.

**NAME\_END** Contains the address of the end of the function name dictionary. It is used to detect the failure of a dictionary search.

**INPUT** Contains the address of the top of the input buffer, which is the beginning of the most recently received string.

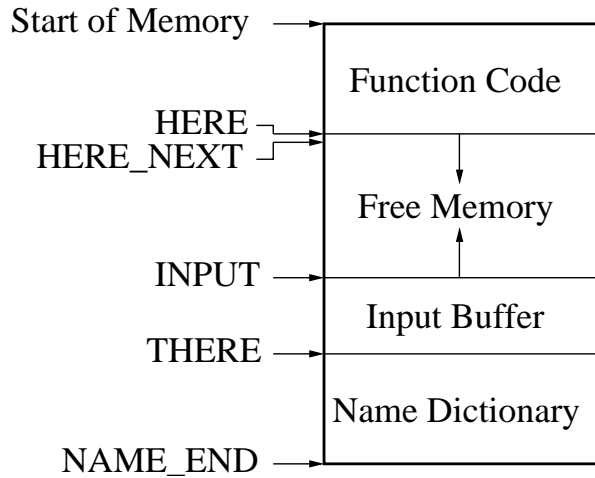


Figure A.1: Flight Language Kernel Memory Map

### A.1.1.1 Counted Strings

The storage and transmission format for strings is that of counted strings (Figure A.2). Contrary to C-style strings, counted strings have no built-in delimiter. They are instead preceded by a single memory word which contains the count of memory words used by the body of the string. The contents of the body are arbitrary.

Once received and stored into memory, a counted string gets a tail appended, which is not taken as part of the count. This tail contains either its own address, thus guaranteeing that a string comparison will always terminate<sup>1</sup>, or the address of the code with which the string is associated, thus forming a name dictionary entry.



Figure A.2: Counted String Format

<sup>1</sup>Comparing a string to itself is a corner case not dealt with here.

## A.1.2 Utility Functions

These are small functions which synthesize operations not implemented by the Gullwing processor. They are shown here as subroutines but are normally compiled in-line due to their size.

**MINUS** Negates the number on the top of the Data Stack before adding it to the next number.

```
NOT LIT 1 PLUS PLUS RET
```

**OR** Performs the bit-wise logical OR of the top two numbers on the Data Stack.

```
OVER NOT AND XOR RET
```

## A.1.3 String Functions

These are the lowest level input buffer manipulation routines. The input buffer behaves like a simple stack of counted strings. The top of stack pointer is INPUT, the bottom pointer is THERE.

**STRING\_TAIL** Returns the address of the tail of a string. Takes the head address of the string. The tail contains an address used for referencing code or terminating string comparisons.

```
//get string length and skip it
>A A@+
//string tail address
A> + RET
```

**PUSH\_STRING** Alters INPUT to allocate the space for a counted string. Sets the tail of the string to point to its own address. Takes the length of the string from the Data Stack.

```
// account for string count and address tail
DUP LIT 1 + NOT
// adjust INPUT
LIT [address of INPUT] >A A@ + DUP A!
// store string length count at [INPUT]
>A A!
LIT [address of INPUT] >A A@
CALL STRING_TAIL
DUP >A A! RET
```

**POP\_STRING** Alters INPUT to discard the most recently SCAN'ed string.

```
LIT [address of INPUT] >A A@
CALL STRING_TAIL
// point to head of next string
LIT 1 +
LIT [address of INPUT] >A A! RET
```

For comparison, this alternate version keeps a copy of the address of INPUT on the stack. It avoids a literal fetch and a function call, but is harder to follow. An optimizing compiler might generate code like this.

```
LIT [address of INPUT] DUP >R
// get [INPUT]
>A A@ DUP
// get string length
>A A@
// add length+2 to [INPUT] to point to next string
+ LIT 2 +
// update INPUT
R> >A A! RET
```

**COMPARE\_STRINGS** Takes the head address of two strings. Returns the addresses (in the same order) of the first non-matching pair of symbols. Note that the practise of tailing each string with the address of the tail guarantees that the comparison will terminate (at the tail).

```
>A >R
LOOP:
R@+ A@+ XOR
JMP0 LOOP
LIT -1 R> +
LIT -1 A> + RET
```

## A.1.4 Input Functions

These functions read in counted strings from the outside world.

**READ1** Presumed here is that READ1 always returns a memory word. Implementation depends on interface to outside world. Here I assume a memory port for illustration purposes.

```
// read in one memory word
LIT [address of input port] >A A@ RET
```

**SCAN\_STRING** Reads a string into a pushed string entry.

```
// get string length and prep
LIT [address of INPUT] >A A@ >R R@+
SCAN_STRING_LOOP:
// READ1 uses A
CALL READ1 R!+
LIT -1 + DUP JMP0 DONE
JMP SCAN_STRING_LOOP
// storage address should be equal to (INPUT)+length+1 now
DONE:
// store address of end of string in itself for LOOK termination
DROP R> DUP >A A! RET
```

**SCAN** Reads in a string from the outside world. Input is a counted string, where the first memory word contains the following number of memory words used by the string, regardless of symbol encoding. Always reads in the string. There is no check.

The location after the string is kept free to hold its own address (which is [THERE]) as a termination marker for LOOK, or an actual code address should it become a name entry.

To see if there is enough free space, the sender of the string can check in advance if the result of "INPUT HERE -" is greater than the string length. The check depends on the fact that HERE points to an address that begins at 0 and increments, while INPUT points to a location that begins at THERE and decrements. THERE begins at the end of memory and decrements towards zero. If they meet, then there is no more free memory. Thus if the start address is lesser than the one HERE points at, there is no room for the string (or for anything else at all!).

```
CALL READ1
CALL PUSH_STRING
JMP SCAN_STRING
```

## A.1.5 Name Lookup

**LOOK** Searches the name dictionary for the topmost string in the input buffer. Returns the address of the code associated with the name or NAME\_END if no match. No error checking other than for the end of the dictionary.

```
// address of latest name entry string
LIT [address of THERE] >A A@ LIT 1 + DUP
LOOP:
// address of topmost SCANed string
LIT [address of INPUT] >A A@
CALL COMPARE_STRINGS
LIT [address of INPUT] >A A@
CALL STRING_TAIL
XOR JMP0 MATCH
```

```

// uses the DUPed name entry address
DROP
CALL STRING_TAIL
// point to start of next name entry
LIT 1 +
// are we at end of name dict?
DUP LIT [address of NAME_END] >A A@ XOR JMP0 NOMATCH
DUP JMP LOOP
MATCH:
// return code address for name entry
>A DROP A@ RET
NOMATCH:
// return address of end of dict
DROP LIT [address of NAME_END] >A A@ RET

```

### A.1.6 Function Definition Functions

These functions create new name dictionary entries and setup the compilation of the code that will be associated with the new name.

**FIRST\_SLOT** Begin compiling at zeroth instruction slot.

```

LIT [address of SLOT] >A
LIT [mask for slot 0] A! RET

```

**LAST\_SLOT** Setup at fifth instruction slot, so the next compilation will occur at the zeroth slot.

```

LIT [address of SLOT] >A
LIT [mask for slot 5] A! RET

```

**NULL\_SLOT** Null instruction slot mask. Denotes that HERE is empty.

```

LIT [address of SLOT] >A
LIT [mask for null slot] A! RET

```

**ALIGN** Makes HERE point to the next free location, so we don't point a name at the tail end of the previous procedure or clobber literals. Make HERE\_NEXT point to the following location. Mark HERE as empty with NULL\_SLOT.

```

LIT [address of HERE_NEXT] >A A@
// zero out location, and increment address
DUP >R LIT 0 R!+
// update HERE_NEXT
R> A!

```

```

// update HERE
LIT [address of HERE] >A A!
JMP NULL_SLOT

```

**NEXT\_SLOT** Point to first slot if HERE is empty (NULL\_SLOT), else point to next free slot at HERE, else ALIGN.

```

// get slot mask
LIT [address of SLOT] >A A@
LIT [null slot mask] OVER XOR JMP0 HEREEMPTY
LIT [fifth slot mask] OVER XOR JMP0 HEREFULL
// next 5-bit slot
2* 2* 2* 2* 2* A! RET
HEREFULL:
DROP CALL ALIGN JMP FIRST_SLOT
HEREEMPTY:
DROP JMP FIRST_SLOT

```

**DEFN\_AS** Takes the ALIGNED address of a code entry (usually [HERE]) and the address of a SCANed string (from INPUT) and converts it to a name entry. Updates THERE.

The name string must be the only one in the input buffer stack as it is converted in place, so INPUT must be pointing to it and its tail must be at THERE. Because of this, INPUT does not need to be changed.

```

// store address in [THERE]
LIT [address of THERE] >A A@ >A A!
// get string start address
LIT [address of INPUT] >A A@
// move to first free location before it
LIT -1 +
LIT [address of THERE] >A A! RET

```

**NEW\_WORD** Returns an aligned address at which to compile code.

```

CALL ALIGN
// get [HERE]
LIT [address of HERE] >A A@ RET

```

**DEFN** Defines a name entry for the current code location.gate

```

CALL NEW_WORD
JMP DEFN_AS

```

## A.1.7 Compilation Functions

These functions enable the kernel to compile the Gullwing opcodes. By convention, functions that compile a Gullwing opcode or inline another function have parentheses around their name.

**COMPILE\_INIT** Initialize HERE, HERE\_NEXT, and SLOT for compilation.

```
LIT [address of HERE] >A
LIT [initial compilation address] A!+ A>
LIT [address of HERE_NEXT] >A A!
LIT [address of SLOT] >A
LIT [mask for slot 5] A!
LIT [address of HERE] >A A@ >A
// Zero out memory to compile to
LIT 0 A! RET
```

**COMPILE\_OPCODE** Takes an opcode (in slot 0) and compiles it at the next empty HERE/SLOT location. Assumes the current slot is full. Leaves SLOT pointing to next full slot.

This function fails for the PC@ instruction, since its opcode is all zeroes. It shouldn't be necessary to compile it explicitly, as that's what ALIGN does when it zeroes the memory word. Note the addition instead of logical OR at the end. We can do this since empty slots are filled with PC@ opcodes (zeroes).

```
CALL NEXT_SLOT
// get slot mask and invert
>R LIT [address of SLOT] >A A@ ~
//place opcode on top
R>
// slot 0
OVER OVER AND JMP0 COMPILE
// shift opcode by one slot
2* 2* 2* 2* 2*
// slot 1
OVER OVER AND JMP0 COMPILE
2* 2* 2* 2* 2*
// slot 2
OVER OVER AND JMP0 COMPILE
2* 2* 2* 2* 2*
// slot 3
OVER OVER AND JMP0 COMPILE
2* 2* 2* 2* 2*
// slot 4
OVER OVER AND JMP0 COMPILE
2* 2* 2* 2* 2*gate
// then we *have* to be in slot 5
```

```

COMPILE:
LIT [address of HERE] >A A@
// compile opcode, drop mask, return
>A A@ + A! DROP RET

```

**COMPILE\_LITERAL** Takes a procedure address or literal (for CALL, JMP, JMP0, or LIT) and compiles it at HERE\_NEXT. Increments HERE\_NEXT.

```

LIT [address of HERE_NEXT] >A A@gate
// store address/literal and increment
>A A!+
// update HERE_NEXT
A> LIT [address of HERE_NEXT] >A A! RET

```

**CMPC, (CALL)** Takes the address of a procedure and compiles a call to it. Then aligns to the next free word since later slots in current word will never execute.

```

LIT [opcode for CALL]
CALL COMPILE_OPCODE
CALL COMPILE_LITERAL
JMP ALIGN

```

**CMPJ, (JMP)**

```

LIT [opcode for JMP]
CALL COMPILE_OPCODE
CALL COMPILE_LITERAL
JMP ALIGN

```

**CMPO, (JMP0)** If JMP0 is not taken, the next opcode in the same memory word runs instead, unlike CALL and JMP.

```

LIT [opcode for JMP0]
CALL COMPILE_OPCODE
JMP COMPILE_LITERAL

```

**CMP+, (JMP+)**

```

LIT [opcode for JMP+]
CALL COMPILE_OPCODE
JMP COMPILE_LITERAL

```

**NUMC, (LIT)** Compiles a literal fetch instruction.

```

LIT [opcode for LIT]
CALL COMPILE_OPCODE
JMP COMPILE_LITERAL

```

## A.1.8 Inline Compilation

Copying a word slot-by-slot requires decompiling it instruction by instruction and recompiling it at the new location. This requires reimplementing the compilation words 'in reverse'. This is complicated and ugly. Simply ALIGNing and copying existing code is no better since the words that will be worthwhile to inline are the shortest, and so many slots will get proportionally wasted. The extreme case is when inlining machine instructions: you would end up with one opcode per word! So that's the key: we know in advance which words will need to be inlined, and so we write a function which when run compiles the code in situ. It's a macro! The kernel contains a full set of inlining functions for the Gullwing opcodes.

**(DUP), (DROP), (+), (A!+), etc...**

```
LIT [opcode in slot 0]
JMP COMPILER_OPCODE
```

## A.1.9 Main Loop

These functions constitute the main loop of the kernel.

**EXECUTE** Synthesizes a function call or jump using an address on the top of the Data Stack. If called: Takes a procedure address and calls to it. If inlined or jumped-to: Takes a procedure address and jumps to it.

```
// Place the function address on the Return Stack
// and execute a function return, 'returning' to the
// pushed address.
>R RET
```

**NXEC** This is the user-interface loop. Reads in a string, looks up the function address, and calls to it. In effect, the kernel does nothing but execute all commands given to it.

```
CALL SCAN
CALL LOOK
CALL POP_STRING
CALL EXECUTE
JMP NXEC
```

**EXEC** Reads in the address of a function and calls to it. It is an alternate main loop meant to be used by communicating instances of the kernel since it is more efficient to pass function addresses than names.

```
CALL READ1
CALL EXECUTE
JMP EXEC
```

### A.1.10 Decimal to Binary Conversion

By design, the encoding of strings is irrelevant. However, numbers cannot avoid a predefined decimal encoding. Ideally, this would be UNICODE, but I settled for decimal ASCII numerals for now (0 -> 48,...9 -> 57), one per memory word. Wasteful, but simple.

**TENSTAR, 10\*** Multiply an integer on the Data Stack by 10.

```
DUP 2* 2* 2* OVER + + RET
```

**NUMI** Takes the string address of an unsigned decimal number and returns the corresponding integer on the stack. No error or overflow checking!

```
LIT [address of INPUT] >A A@
// read in length
>R R@+
LIT 0 >A
// if N == 0
DUP JMP0 DONE
LOOP:
// shift total by radix, get number, convert to int, add
A> CALL 10* R@+ LIT -48 + + >A
LIT -1 + DUP JMP0 DONE
JMP LOOP
DONE:
DROP R> DROP A>
JMP POP_STRING
```

## A.2 Flight Language Extensions

The source to the Flight language extensions is a sequence of commands fed to the Flight language kernel. It is important to keep in mind that the kernel does not parse its input. It only looks up names and executes functions. Parsing is implemented by executing words which consume some of the input stream before returning control to the kernel main loop. Compilation is implemented by executing code whose action is to compile code.

There is no built-in syntax. The only purpose of white space is to separate names. Indentation and other formatting is for clarity only. I have added comments after double slashes '//', but they are not present in the actual source. To convey meaning, I use some naming conventions throughout the code:

- Functions which compile code have their names between parentheses '()'.  
• The use of strings in the input buffer is denoted by a dollar sign '\$'.  
• The use of integer values is denoted by a hash sign '#'.  
• The use of floating point values is denoted by a percent sign '%'.  
• The use of pointers is denoted by an ampersand '&'.  
• The use of arrays is denoted by a bracket '[' and a closing bracket ']'.

- Moving data, input, and output is denoted by angle brackets '<' and '>'.

The Flight language evolves very quickly at the beginning. The first few functions are used throughout the latter code and must be understood before proceeding further.

### A.2.1 Making the Flight Language More Tractable

These utility functions make it easy to define functions, compile calls, read in numbers, and do name look-ups. They are used throughout the rest of the code.

```
// Creates a function named ":"
// It scans in a string and creates a dictionary entry.
SCAN : DEFN
SCAN SCAN LOOK POP_STRING CMPCALL
SCAN DEFN LOOK POP_STRING CMPCALL CMPRET

// Reads in a name and leaves the address of the
// associated code on the stack.
: l
SCAN SCAN LOOK POP_STRING CMPCALL
SCAN LOOK LOOK POP_STRING CMPCALL
SCAN POP_STRING LOOK POP_STRING CMPCALL CMPRET

// Takes a string from the input buffer
// and compiles a call to its associated code
: $c
l LOOK CMPCALL
l POP_STRING CMPCALL
l CMPCALL CMPCALL CMPRET

// Reads in a function name
// and compiles a call to it
: c SCAN SCAN $c SCAN $c $c CMPRET

// Reads an unsigned decimal number and leaves
// its binary representation on the stack
: n c SCAN c NUMI CMPRET

// Creates a new dictionary entry for
// an existing function
: alias c l c SCAN c DEFN_AS CMPRET

// Alias the built-in function names
// to get away from C-style identifiers.
alias CMPJMP      (JMP)
```

```

alias CMPJMPZERO      (JMP0)
alias CMPJMPPLUS     (JMP+)
alias CMPCALL        (CALL)
alias CMPRET         ;
alias NUMC           #
alias NUMI           $n
alias LOOK           $l
alias POP_STRING     $pop
alias DEFN           $:
alias SCAN           >$
alias WRITE1        #>
alias READ1         >#
alias TENSTAR        10*
alias CMPLOADA      (A@)
alias CMPSTOREA     (A!)
alias CMPLOADAPLUS  (A@+)
alias CMPSTOREAPLUS (A!+)
alias CMPLOADRPLUS  (R@+)
alias CMPSTORERPLUS (R!+)
alias CMPXOR         (XOR)
alias CMPAND         (AND)
alias CMPNOT         (NOT)
alias CMPTWOSTAR    (2*)
alias CMPTWOSLASH   (2/)
alias CMPPLUS       (+)
alias CMPPLUSSTAR   (+*)
alias CMPDUP         (DUP)
alias CMPDROP       (DROP)
alias CMPOVER       (OVER)
alias CMPTOR        (>R)
alias CMPRFROM      (R>)
alias CMPTOA        (>A)
alias CMPAFROM      (A>)
alias CMPNOP        (NOP)

// Compile jumps and conditional jumps
: $j c $l c $pop c (JMP) ;
: j c >$ c $j ;

: $j0 c $l c $pop c (JMP0) ;
: j0 c >$ c $j0 ;

: $j+ c $l c $pop c (JMP+) ;
: j+ c >$ c $j+ ;

```

## A.2.2 Interactively Usable Opcodes

These functions create interpreted versions of some opcodes to interactively manipulate memory and the Data Stack. Opcodes which manipulate the A register and the Return Stack cannot be interpreted since the interpretation process alters their contents.

```
: @      (>A) (A@) ;
: !      (>A) (A!) ;
: XOR    (XOR) ;
: AND    (AND) ;
: NOT    (NOT) ;
: OR     (OVER) (NOT) (AND) (XOR) ;
: 2*     (2*) ;
: 2/     (2/) ;
: +      (+) ;
: ++     (++) ;
: DUP    (DUP) ;
: DROP   (DROP) ;
: OVER   (OVER) ;
: NOP    (NOP) ;
```

## A.2.3 Basic Compiling Functions

These are some compiling functions for common operations, mostly arithmetic.

```
// Compiles addition with a fetched constant
: #+ c # c (+) ;
// Compiles a decimal number literal fetch
: n# c n c # ;
// Compiles a function address literal fetch
: l# c l c # ;

// Compile loads and stores and bitwise OR
: (@) c (>A) c (A@) ;
: (!) c (>A) c (A!) ;
: (OR) c (OVER) c (NOT) c (AND) c (XOR) ;

// Negate, or compile its code
: negate (NOT) n# 1 (+) ;
: (negate) c (NOT) n# 1 c #+ ;

// Read in and negate a decimal
: -n c n (negate) ;
// Same, from the input buffer
: -$n c $n (negate) ;
```

```

// Same, and compile as literal
: -n# c -n c # ;

// Compiles code to read in a decimal
// and compile it as a constant addition
// Negate beforehand to make a subtraction
: (N-) c -n c #+ ;
: (N+) c n c #+ ;

// Synthesize subtraction
: - (negate) (+) ;
: (-) c (negate) c (+) ;

```

## A.2.4 Terminal Control Characters

These output the basic terminal control characters (as counted strings of length 1).

```

: \a n# 7 n# 1 c #> j #>
: \b n# 8 n# 1 c #> j #>
: \t n# 9 n# 1 c #> j #>
: \n n# 10 n# 1 c #> j #>
: \v n# 11 n# 1 c #> j #>
: \f n# 12 n# 1 c #> j #>
: \r n# 13 n# 1 c #> j #>
: \s n# 32 n# 1 c #> j #>

```

## A.2.5 Conditionals and Comparisons

These functions implement the usual if/then construct. The two conditions are “if non-zero” and “if negative”. Some usage examples follow.

```

// Compile a JMP0 to 0, and leave the address
// of the jump address on the stack
: if
n# 0 c (JMP0)
l# HERE_NEXT (@)
(N-) 1 ;

: if-
n# 0 c (JMP+)
l# HERE_NEXT (@)
(N-) 1 ;

```

```

// Backpatch the jump to target
// the next memory word
: else
(>R) c NEW_WORD (R>)
(>A) (A!) ;

: max
(OVER) (OVER) (-)
if-
  (>R) (DROP) (R>) ;
else
  (DROP) ;

: min
(OVER) (OVER) (-)
if-
  (DROP) ;
else
  (>R) (DROP) (R>) ;

: abs
(DUP) if- (negate) ; else ;

: <=
(-) (DUP)
if-
  (DROP) -n# 1 ;
else
  if
    n# 0 ;
  else
    -n# 1 ;

: >=
(-) (DUP)
if-
  (DROP) -n# 0 ;
else
  if
    n# 0 ;
  else
    -n# 1 ;

```

## A.2.6 Code Memory Allocation

This function allocates a zeroed-out span of memory in the code area, usually for static storage of data.

```
: allot
(DUP)
if
  c ALIGN
  (N-) 1
  j allot
else
  (DROP) ;
```

## A.2.7 String Copying and Printing

These functions print strings and copy them between the input buffer and the code area.

```
// Copies a string between a source
// and a destination address
: $copy
(>R) (>A)
(A@) (A>) (+) (N+) 1
: $copy-loop (A>) (OVER) (XOR)
if (A@+) (R!+) j $copy-loop
else (DROP)
(R>) (DUP) (>A) (A!) ;

// Copy a string from the input buffer
// to the code area
: $>c
l# INPUT (@)
l# HERE (@)
(OVER) (@) (N+) 1 c allot
c $copy
c ALIGN
j POP_STRING

// Copy a string from the code area
// to input buffer
: c>$
(DUP) (@) c PUSH_STRING
l# INPUT (@)
c $copy ;
```

```

// Output a string, given its address
: cs>
(DUP) (@) (+) (N+) 1
(A>) (>R)
: cs>-loop (R>) (OVER) (OVER) (XOR)
if (>R) (R@+) c #> j cs>-loop
else (DROP) (DROP) ;

// Output a string from the input buffer
: $>
l# INPUT (@)
c cs>
j POP_STRING

// Print a string, given its name
// copy first to input buffer
: $print c l c c>$ j $>
// Same, without copying
: csprint c l j cs>

```

## A.2.8 De-Allocating Functions

Executing 'forget foo' will move back the HERE, HERE\_NEXT, THERE, and INPUT pointers to points just before the name and the code of the function 'foo', in effect erasing it from the language. This will also forget all functions that had been defined after 'foo'.

```

: match?
l# INPUT (@) c STRING_TAIL (XOR) ;

: end?
l NAME_END @ # (XOR) ;

: erase
(DUP) l# THERE (!)
(DUP) (N+) 1 l# INPUT (!)
(@) (DUP)
(N-) 1 l# HERE (!)
l# HERE_NEXT (!)
j ALIGN

```

```

: forget
c >$
l# THERE (@) (N+) 1 (DUP)
: forget-loop
l# INPUT (@)
c COMPARE_STRINGS
c match?
if (DROP) c STRING_TAIL (N+) 1 (DUP) c end?
  if
    (DUP) j forget-loop
  else
    (DROP) c POP_STRING ;
  else
    c erase (DROP) ;

```

## A.2.9 Unsigned Multiplication and Division

These functions synthesize unsigned integer multiplication and division. The multiplication function takes two 15-bit integers and returns the 30-bit product. The division function has a simple error reporting mechanism if a division by zero is attempted. It will stop and send the remaining input to the output so the location of the fault is made visible.

```

: 15x15
(>R)
(2*) (2*) (2*) (2*)
(2*) (2*) (2*) (2*)
(2*) (2*) (2*) (2*)
(2*) (2*) (2*)
(R>)
(++ (2/) (++ (2/) (++ (2/) (++ (2/))
(++ (2/) (++ (2/) (++ (2/) (++ (2/))
(++ (2/) (++ (2/) (++ (2/) (++ (2/))
(++ (2/) (++ (2/) (++ (2/) (++
(>R) (DROP) (R>) ;

: divby0msg >$ DIV_BY_0_ERROR $>c
: divby0 l# divby0msg c cs> ;
: errcontext c \s c >$ c $> j errcontext
: divby0check
(DUP) if ; else (DROP) (DROP) c divby0 j errcontext

```

```

: U/
c divby0check
n# 0 (>A)
: U/-loop
(OVER) (>R) (DUP) (R>) c <=
if
  (DUP) (>R) (-) (R>)
  (A>) (N+) 1 (>A)
  j U/-loop
else
  (DROP) (A>) ;

```

## A.2.10 Binary to Decimal Conversion

```

: minus?
if- n# 1 ; else n# 0 ;

: numstrlen
c abs (DUP) (N-) 10
if- (DROP) n# 1 ; else (DUP) (N-) 100
if- (DROP) n# 2 ; else (DUP) (N-) 1000
if- (DROP) n# 3 ; else (DUP) (N-) 10000
if- (DROP) n# 4 ; else (DUP) (N-) 100000
if- (DROP) n# 5 ; else (DUP) (N-) 1000000
if- (DROP) n# 6 ; else (DUP) (N-) 10000000
if- (DROP) n# 7 ; else (DUP) (N-) 100000000
if- (DROP) n# 8 ; else (DUP) (N-) 1000000000
if- (DROP) n# 9 ; else (DROP) n# 10 ;

// Converts a signed integer to a string
// in the input buffer
: #>$
(DUP) c numstrlen (OVER) c minus? (+) c PUSH_STRING
(DUP) if- n# 45 l# INPUT (@) (N+) 1 (!) c abs else
(>R) l# INPUT (@) c STRING_TAIL (N-) 1 (R>)
: #>$-loop
n# 10 c U/ (>R) n# 48 (+) (OVER) (!) (N-) 1 (R>)
(DUP) if j #>$-loop else (DROP) (DROP) ;

// Convert and print a decimal number
: #>$ c #>$ c $> ;

```

## A.2.11 Simple Fibonacci Examples

```
// Given two Fibonacci numbers,
// compute the next pair
// eg: 1 1 -> 1 2 -> 2 3 ...
: lfib
(OVER) (>R) (+) (R>) ;

// Given starting numbers and a count
// generate a string of Fibonacci numbers
// in the input buffer
// Terminates by comparing current position in
// the buffer with that of the string tail
: nfibx
c PUSH_STRING l# INPUT (@)
(DUP) (>R) c STRING_TAIL (R>) (N+) 1 (>A)
: nfib-loop
(>R) c lfib (DUP) (A!+) (R>) (DUP) (A>) (XOR)
if j nfib-loop
else (DROP) (DROP) (DROP) ;

// As above, but terminates by decrementing
// the counter to zero
: nfibc
(DUP) c PUSH_STRING l# INPUT (@) (N+) 1 (>A)
: nfib-loop
(>R) c lfib (DUP) (A!+) (R>) (N-) 1 (DUP)
if j nfib-loop
else (DROP) (DROP) (DROP) ;

: manyfibs
c lfib c lfib c lfib j lfib

// Calculates the mean of all numbers in a string
// Could be used after nfibx as an example of
// function composition through the input buffer
: nmean
n# 0
l# INPUT (@) (DUP) (>R) c STRING_TAIL (R>) (N+) 1 (>A)
: nmean-loop
(>R) (A@+) (+) (R>) (DUP) (A>) (XOR)
if j nmean-loop
else (DROP) l# INPUT (@) (@) c U/ c POP_STRING ;
```

## A.2.12 Static Variables

This code creates the ability to store data in the code area at the time a function is created. It is equivalent to C language static variables. This is used to implement named variables as functions which return the address of the associated storage.

```
// Allocates one memory word and compiles code
// to place its address on the Data Stack
: create
n# 0 c # l# HERE_NEXT (@) (N-) 1 (>R)
n# 0 c (JMP) l# HERE (@) (N-) 1
l# HERE (@) (R>) (!) ;

// This ends the data allocation and begins
// the code that will use the address of the
// allocated area.
// Exact same code as else, so just alias it instead
alias else does

// Allocate a number of memory words, then
// compile code to return its address
: var c create (>R) (N-) 1 c allot (R>) c does ;

// Create a global variable named 'first'
// with a size of one memory location
: first n 1 var ;

// Store integer '4' in first
// then load and print it
n 4 first !
first @ #> \t

// Compile code as data
// then call to it
: pass
n# 1 c #> c \t
create
  n# 2 c #> c \n ;
does
  n# 3 c #> c \t
c EXECUTE ;

// Output: 1 3 2
pass
```

## A.2.13 Accumulator Generator

This is a simple example of generating code with an initial argument that becomes static data.

```
// Compile an integer into the code area
: #>c l# HERE (@) (!) ;

// Compiles code that returns an accumulator
// function using a provided integer
// Returns the function's address
: accgen
c NEW_WORD (>R)
c create (>R) c #>c (R>) c does
c (@) c (+) c (DUP) c (A!) c ;
(R>) ;

// Read in a name and associate it
// with an address on the stack.
: name-as c >$ c DEFN_AS ;

// Create two accumulators and test them.
// Output: 8 7 10 9 17 16 117 116
n 3 accgen name-as foo
n 2 accgen name-as bar
n 5 foo #> \t
n 5 bar #> \t
n 2 foo #> \t
n 2 bar #> \t
n 7 foo #> \t
n 7 bar #> \t
n 100 foo #> \t
n 100 bar #> \t \n
```

## A.2.14 Fibonacci Generator

Slightly more elaborate examples of code generation with static data.

```
// Generate a Fibonacci function that stores
// its initial arguments in memory
: fibgen1
c :
c create (>R) c #>c (R>) c does
c create (>R) c #>c (R>) c does
c (OVER) c (@) c (OVER) c (@) c (+) c (>R)
c (OVER) c (@) c (OVER) c (!) c (DROP)
c (>A) c (R>) c (DUP) c (A!) c ; ;
```

```

// Factored out Fibonacci code
: memfib
(OVER) (@) (OVER) (@) (+) (>R)
(OVER) (@) (OVER) (!) (DROP)
(>A) (R>) (DUP) (A!) ;

// Takes 2 numbers in the Fibonacci sequence
// and returns a function that outputs the
// next number in the sequence when called.
// The two current sequence numbers are stored
// within the function body.
: fibgen2
c :
c create (>R) c #>c (R>) c does
c create (>R) c #>c (R>) c does
l# memfib c (CALL) c ; ;

n 0 n 1 fibgen2 fibonacci

// Output: 1 2 3 5 8 13 21 34
fibonacci #> \t
fibonacci #> \t
fibonacci #> \t
fibonacci #> \t
fibonacci #> \t
fibonacci #> \t
fibonacci #> \t
fibonacci #> \t
fibonacci #> \t \n

```

## A.2.15 Caesar Cipher Generator

An initial example of a function being passed as a parameter.

```

// Add a given number to a memory location
: caesar
(>R) (>A) (R>) (A@) (+) (A!) ;

// Make one argument of caesar (the number)
// a built-in parameter
: caesargen
c :
c create (>R) c #>c (R>) c does
c (@) l# caesar c (CALL) c ; ;

```

```

n 3 caesargen encode
-n 3 caesargen decode

// Takes the address of a string
// and the name of a function.
// Maps the function to each
// element of the string.
: map1
(DUP) (>R) c STRING_TAIL (R>)
(N+) 1
c 1 (>R)
: map1-loop
(DUP) (R>) (DUP) (>R) c EXECUTE (N+) 1
(OVER) (OVER) (XOR)
if j map1-loop
else (DROP) (DROP) (R>) (DROP) ;

// Input: ABCD
// Output: ABCD DEFG ABCD
>$ ABCD
// Print the string
l INPUT @ DUP cs> \t
DUP map1 encode
// Print the ciphered version
DUP cs> \t
// Print the deciphered version
DUP map1 decode
$> \t \n

// Since the argument is a constant
// it can be compiled as a literal fetch instead
: caesargen
c :
c #
l# caesar c (JMP) ;

```

## A.2.16 Higher-Order Function (Map)

This is an example of a mapped function generator.

```
// Takes an integer and the function name of
// a function that alters memory.
// Compiles code which applies the given
// function to each location in a string
// at the interval provided by the integer
: mapgen
c (DUP) c (>R) l# STRING_TAIL c (CALL) c (R>)
n 1 # c # c (+)
c NEW_WORD
c (DUP) c l c (CALL) (>R) c # (R>) c (+)
c (OVER) c (OVER) c (XOR)
c if (>R) c (JMP) (R>)
c else c (DROP) c (DROP) c ; ;

n 5 caesargen encode1
-n 5 caesargen decode1

// Apply encode1 to every other character in a string
: cipher n 2 mapgen encode1
// Apply decode1 to each character in a string
: decipher n 1 mapgen decode1

// Input: lmnopq
// Output:lmnopq qmsouq lhnjpl
>$ lmnopq
l INPUT @ DUP cs> \t
DUP cipher
DUP cs> \t
DUP decipher
$> \t \n

// Given a memory address, print the
// decimal expression of its contents
: print# (@) c #> c \t ;
// Apply print# to each location in a string
: print$# n 1 mapgen print#

// Generate and print
// the first 8 Fibonacci numbers
// Output: 1 1 2 3 5 8 13 21
n 1 n 0 n 8 nfibx
l INPUT @ print$# POP_STRING
```

## A.3 Virtual Machine

The virtual machine is an emulation of the Gullwing hardware. The opcodes are emulated directly on the hardware if possible, and their memory accesses are bounds-checked.

### A.3.1 VM

```
// Define an 8kB memory for the VM
: MEMSIZE n# 8192 ;
: MEM MEMSIZE var ;

: OPCODEWIDTH n# 5 ;
: OPCODEMASK n# 31 ;

: MEM_INPUT MEM MEMSIZE + -n 2 + # ;
: MEM_OUTPUT MEM MEMSIZE + -n 1 + # ;

: MEMHEAD MEM # ;
: MEMTAIL MEM_OUTPUT # ;

: (check_low)
MEMHEAD negate # c # c (+) ;

: (check_high)
c (negate) MEMTAIL # c # c (+) ;

: mem_in_range?
(DUP) (check_low) (OVER) (check_high) (OR) ;

: mem_access_msg
create >$ ILLEGAL_MEMORY_ACCESS: $>c
does j cs>

: report_mem_error
c mem_access_msg
c \s c #> j \n

: access_check
c mem_in_range?
if-
  c report_mem_error
  j errcontext
else ;
```

```

: PCFETCHopcode      n# 0 ;
: CALLopcode         n# 1 ;
: RETopcode          n# 2 ;
: JMPopcode          n# 3 ;
: JMPZEROopcode      n# 4 ;
: JMPPLUSopcode      n# 5 ;
: LOADAopcode        n# 6 ;
: STOREAopcode       n# 7 ;
: LOADAPLUSopcode    n# 8 ;
: STOREAPLUSopcode   n# 9 ;
: LOADRPLUSopcode    n# 10 ;
: STORERPLUSopcode   n# 11 ;
: LITopcode          n# 12 ;
: UND0opcode         n# 13 ;
: UND1opcode         n# 14 ;
: UND2opcode         n# 15 ;
: XORopcode          n# 16 ;
: ANDopcode          n# 17 ;
: NOTopcode          n# 18 ;
: TWOSTARopcode      n# 19 ;
: TWOSLASHopcode     n# 20 ;
: PLUSopcode         n# 21 ;
: PLUSSTARopcode     n# 22 ;
: DUPopcode          n# 23 ;
: DROPopcode         n# 24 ;
: OVERopcode         n# 25 ;
: TORopcode          n# 26 ;
: RFROMopcode        n# 27 ;
: TOAopcode          n# 28 ;
: AFROMopcode        n# 29 ;
: NOPopcode          n# 30 ;
: UND3opcode         n# 31 ;

```

```

// Emulated A, PC and ISR
: AREG   n 1 var ;
: PCREG  n 1 var ;
: ISRREG n 1 var ;

```

```

: do_pcfetch
PCREG # (@)
c access_check (DUP)
(N+) 1 (A!)
(@) ISRREG # (!) ;

```

```

: do_call
// Move run_vm return address
(R>)
PCREG # (@)
(DUP) (N+) 1 (>R)
(@) c access_check (DUP)
(N+) 1 PCREG # (!)
(@) ISRREG # (!)
// Restore run_vm return address
(>R) ;

: do_ret
(R>)
(R>) c access_check (DUP)
(N+) 1 PCREG # (!)
(@) ISRREG # (!)
(>R) ;

: do_jump
PCREG # (@) (@)
c access_check (DUP)
(N+) 1 PCREG # (!)
(@) ISRREG # (!) ;

: do_jumpzero
if
  PCREG # (@) (N+) 1 (A!) ;
else
  j do_jump

: do_jumpplus
if-
  PCREG # (@) (N+) 1 (A!) ;
else
  j do_jump

: do_loada
AREG # (@)
(DUP) MEM_INPUT # (XOR)
if
  c access_check (@) ;
else
  (DROP) j >#

```

```

: do_storea
AREG # (@)
(DUP) MEM_OUTPUT # (XOR)
if
  c access_check (!) ;
else
  (DROP) j #>

: do_loada_plus
AREG # (@)
(DUP) MEM_INPUT # (XOR)
if
  c access_check
  (>A) (A@+) (A>)
  AREG # (!) ;
else
  (N+) 1 AREG # (!) j >#

: do_storea_plus
AREG # (@)
(DUP) MEM_OUTPUT # (XOR)
if
  c access_check
  (>A) (A!+) (A>)
  AREG # (!) ;
else
  (N+) 1 AREG # (!) j #>

: do_loadr_plus
(R>) (>A)
(R>)
(DUP) MEM_INPUT # (XOR)
if
  c access_check
  (>R) (R@+)
  (A>) (>R) ;
else
  (N+) 1 (>R)
  (A>) (>R)
  j >#

```

```

: do_storerplus
(R>) (>A)
(R>)
(DUP) MEM_OUTPUT # (XOR)
if
  c access_check
  (>R) (R!+)
  (A>) (>R) ;
else
  (N+) 1 (>R)
  (A>) (>R)
  j #>

: do_lit
PCREG # (@)
(DUP) (N+) 1 (A!)
(@) ;

: do_und      n 31 COMPILE_OPCODE ;
: do_xor      (XOR) ;
: do_and      (AND) ;
: do_not      (NOT) ;
: do_twostar  (2*) ;
: do_twoslash (2/) ;
: do_plus     (+) ;
: do_plusstar (+*) ;
: do_dup      (DUP) ;
: do_drop     (DROP) ;
: do_over     (OVER) ;

: do_tor
(R>) (>A)
(>R)
(A>) (>R) ;

: do_rfrom
(R>) (>A)
(R>)
(A>) (>R) ;

: do_toa  AREG # (!) ;
: do_afrom AREG # (@) ;
: do_nop  (NOP) ;

```

```

: ,
c #>c c ALIGN ;

: &>
c l c , ;

// Indexed by the opcode
: instruction_call_table
create
&> do_pcfetch
&> do_call
&> do_ret
&> do_jump
&> do_jumpzero
&> do_jumpplus
&> do_loada
&> do_storea
&> do_loadaplus
&> do_storeaplus
&> do_loadrplus
&> do_storerplus
&> do_lit
&> do_und
&> do_und
&> do_und
&> do_xor
&> do_and
&> do_not
&> do_twostar
&> do_twoslash
&> do_plus
&> do_plusstar
&> do_dup
&> do_drop
&> do_over
&> do_tor
&> do_rfrom
&> do_toa
&> do_afrom
&> do_nop
&> do_und
does ;

```

```

: (shift_isr)
c (2/) c (2/) c (2/) c (2/) c (2/) ;

: (extract_instruction)
OPCODEMASK # c # c (AND) ;

: do_next_instruction
ISRREG # (@)
(DUP) (shift_isr) (A!)
(extract_instruction)
instruction_call_table # (+) (@)
(>R) ;

// All input now gets processed by the software
// inside the VM instead of the original Flight language kernel
: run_vm
MEM # PCREG # (!)
n 0 # ISRREG # (!)
n 123456 # AREG # (!)
: vm_loop
c do_next_instruction
j vm_loop

// Output short message to show
// that compilation reached this point
>$ VM4-1 $> \n

```

### A.3.2 Metacompiler

The metacompiler saves and restores the internal state of the language kernel. This allows redirecting the operation of the kernel to a different memory area. In this case, it is used to direct compilation and execution of code to the previously defined Virtual Machine memory area.

```

// Virtual Machine state
: vm_here n 1 var ;
: vm_here_next n 1 var ;
: vm_there n 1 var ;
: vm_slot n 1 var ;
: vm_input n 1 var ;
: vm_name_end n 1 var ;

// Native Machine state
: nm_here n 1 var ;
: nm_here_next n 1 var ;

```

```

: nm_there n 1 var ;
: nm_slot n 1 var ;
: nm_input n 1 var ;
: nm_name_end n 1 var ;

: save_nm_here l# HERE (@) nm_here # (!) ;
: save_nm_here_next l# HERE_NEXT (@) nm_here_next # (!) ;
: save_nm_slot l# SLOT (@) nm_slot # (!) ;
: save_nm_input l# INPUT (@) nm_input # (!) ;
: save_nm_there l# THERE (@) nm_there # (!) ;
: save_nm_name_end l# NAME_END (@) nm_name_end # (!) ;

: save_vm_here l# HERE (@) vm_here # (!) ;
: save_vm_here_next l# HERE_NEXT (@) vm_here_next # (!) ;
: save_vm_slot l# SLOT (@) vm_slot # (!) ;
: save_vm_input l# INPUT (@) vm_input # (!) ;
: save_vm_there l# THERE (@) vm_there # (!) ;
: save_vm_name_end l# NAME_END (@) vm_name_end # (!) ;

: restore_nm_here nm_here # (@) l# HERE (!) ;
: restore_nm_here_next
nm_here_next # (@) l# HERE_NEXT (!) ;
: restore_nm_slot nm_slot # (@) l# SLOT (!) ;
: restore_nm_input nm_input # (@) l# INPUT (!) ;
: restore_nm_there nm_there # (@) l# THERE (!) ;
: restore_nm_name_end nm_name_end # (@) l# NAME_END (!) ;

: restore_vm_here vm_here # (@) l# HERE (!) ;
: restore_vm_here_next
vm_here_next # (@) l# HERE_NEXT (!) ;
: restore_vm_slot vm_slot # (@) l# SLOT (!) ;
: restore_vm_input vm_input # (@) l# INPUT (!) ;
: restore_vm_there vm_there # (@) l# THERE (!) ;
: restore_vm_name_end vm_name_end # (@) l# NAME_END (!) ;

: init_vm_here MEM # vm_here # (!) n# 0 (A@) (!) ;
: init_vm_here_next
MEM n 1 + # vm_here_next # (!) n# 0 (A@) (!) ;
: init_vm_slot n# 0 vm_slot # (!) ;
: init_vm_there MEM_INPUT n 1 - # vm_there # (!) ;
: init_vm_input MEM_INPUT # vm_input # (!) ;
: init_vm_name_end MEM_INPUT # vm_name_end # (!) ;

```

```

// Flight Language Kernel main loop while in VM.
// (Compare with NXEC)
// Lookup names in VM dictionary, and if not found,
// repeat in native machine dictionary.
: vm_nxec
c SCAN
c LOOK
(DUP) l# NAME_END (@) (XOR)
// If found in VM memory
if
  c POP_STRING
  c EXECUTE
  j vm_nxec
else
// Else find in native machine memory
// but execute with kernel pointed at VM memory
(DROP)
  c save_vm_there
  c save_vm_name_end
  c restore_nm_there
  c restore_nm_name_end
  c LOOK
  c save_nm_there
  c save_nm_name_end
  c restore_vm_there
  c restore_vm_name_end
  c POP_STRING
  c EXECUTE
  j vm_nxec

: init_name_dict
create >$ zeroword $>c
does c c>$ n# 0 c DEFN_AS ;

// Drop current caller address
: (unwind) c (R>) c (DROP) ;

// Begin execution inside VM
: VM(
c init_vm_here
c init_vm_here_next
c init_vm_slot
c init_vm_there
c init_vm_input
c init_vm_name_end

```

```

c save_nm_here
c save_nm_here_next
c save_nm_slot
c save_nm_there
c save_nm_input
c save_nm_name_end
c restore_vm_here
c restore_vm_here_next
c restore_vm_slot
c restore_vm_there
c restore_vm_input
c restore_vm_name_end
c init_name_dict
// Change the main loop
(unwind)
j vm_nxec

// End execution inside VM
: )VM
c save_vm_here
c save_vm_here_next
c save_vm_slot
c save_vm_there
c save_vm_input
c save_vm_name_end
c restore_nm_here
c restore_nm_here_next
c restore_nm_slot
c restore_nm_there
c restore_nm_input
c restore_nm_name_end
// Change the main loop
(unwind)
j NXEC

// Print this to signal we got this far
>$ METACOMP1 $> \n

```

### A.3.3 Self-hosted Kernel

This is a reimplementaion of the Flight language kernel, written in the language it defines, using all the extensions previously compiled. It compiles to the same original machine code in which the kernel was originally implemented (Section A.1), but it is compiled in the Virtual Machine memory area instead.

```
VM(

n 0 (JMP) MEM n 1 + >$ START_WORD DEFN_AS

: HERE
: HERE_NEXT
: THERE
: SLOT
: INPUT
: NAME_END

: MINUS (NOT) n# 1 (+) (+) ;
: OR (OVER) (NOT) (AND) (XOR) ;

: READ1 MEM_INPUT # (@) ;
: WRITE1 MEM_OUTPUT # (!) ;
: STRING_TAIL (>A) (A@+) (A>) (+) ;

: PUSH_STRING
(DUP) n# 1 (+) (NOT)
l# INPUT (@) (+) (DUP) (A!)
(!)
l# INPUT (@)
c STRING_TAIL
(DUP) (!) ;

: POP_STRING
l# INPUT (@)
c STRING_TAIL
n# 1 (+)
l# INPUT (!) ;

: COMPARE_STRINGS
(>A) (>R)
: CS_LOOP
(R@+) (A@+) (XOR)
j0 CS_LOOP
-n# 1 (R>) (+)
-n# 1 (A>) (+) ;
```

```

: SCAN_STRING
l# INPUT (@) (>R) (R@+)
: SS_LOOP
c READ1 (R!+)
-n# 1 (+) (DUP)
if
  j SS_LOOP
else
  (DROP) (R>) (DUP) (!) ;

: SCAN
c READ1
c PUSH_STRING
j SCAN_STRING

: FIRST_SLOT
l# SLOT (>A)
n# 31 (A!) ;

: LAST_SLOT
l# SLOT (>A)
n# 1040187392 (A!) ;

: NULL_SLOT
l# SLOT (>A)
n# 0 (A!) ;

: ALIGN
l# HERE_NEXT (@)
(DUP) (>R) n# 0 (R!+)
(R>) (A!)
l# HERE (!)
j NULL_SLOT

: NEXT_SLOT
l# SLOT (@)
n# 0 (OVER) (XOR)
if
  n# 1040187392 (OVER) (XOR)
  if
    (2*) (2*) (2*) (2*) (2*) (A!) ;
  else
    (DROP) c ALIGN j FIRST_SLOT
else
  (DROP) j FIRST_SLOT

```

```

: DEFN_AS
l# THERE (@) (!)
l# INPUT (@)
-n# 1 (+)
l# THERE (!) ;

: NEW_WORD
c ALIGN
l# HERE (@) ;

: DEFN
c NEW_WORD
j DEFN_AS

: LOOK
l# THERE (@) n# 1 (+) (DUP)
: LOOK_LOOP
l# INPUT (@)
c COMPARE_STRINGS
l# INPUT (@)
c STRING_TAIL (XOR)
if
  (DROP) c STRING_TAIL
  n# 1 (+)
  (DUP) l# NAME_END (@) (XOR)
  if
    (DUP) j LOOK_LOOP
  else
    (DROP) l# NAME_END (@) ;
else
  (>A) (DROP) (A@) ;

: COMPILE_OPCODE
c NEXT_SLOT
(>R) l# SLOT (@) (NOT)
(R>) (OVER) (OVER) (AND) if
(2*) (2*) (2*) (2*) (2*)
(OVER) (OVER) (AND) if
(2*) (2*) (2*) (2*) (2*)
(OVER) (OVER) (AND) if
(2*) (2*) (2*) (2*) (2*)
(OVER) (OVER) (AND) if
(2*) (2*) (2*) (2*) (2*)
(OVER) (OVER) (AND) if

```

```

(2*) (2*) (2*) (2*) (2*)
else else else else else
l# HERE (@)
(@) (+) (A!) (DROP) ;

: COMPILE_LITERAL
l# HERE_NEXT (@)
(>A) (A!+)
(A>) l# HERE_NEXT (!) ;

: CMPCALL
CALLopcode # c COMPILE_OPCODE c COMPILE_LITERAL j ALIGN
: CMPJMP
JMPopcode # c COMPILE_OPCODE c COMPILE_LITERAL j ALIGN
: CMPJMPZERO
JMPZEROopcode # c COMPILE_OPCODE j COMPILE_LITERAL
: CMPJMPPLUS
JMPPLUSopcode # c COMPILE_OPCODE j COMPILE_LITERAL
: NUMC
LITopcode # c COMPILE_OPCODE j COMPILE_LITERAL
: CMPPCFETCH      PCFETCHopcode # j COMPILE_OPCODE
: CMPRET          RETopcode # j COMPILE_OPCODE
: CMPLOADAPLUS   LOADAPLUSopcode # j COMPILE_OPCODE
: CMPLOADRPLUS   LOADRPLUSopcode # j COMPILE_OPCODE
: CMPLOADA       LOADAopcode # j COMPILE_OPCODE
: CMPSTOREAPLUS  STOREAPLUSopcode # j COMPILE_OPCODE
: CMPSTORERPLUS  STORERPLUSopcode # j COMPILE_OPCODE
: CMPSTOREA     STOREAopcode # j COMPILE_OPCODE
: CMPNOT         NOTopcode # j COMPILE_OPCODE
: CMPAND         ANDopcode # j COMPILE_OPCODE
: CMPXOR         XORopcode # j COMPILE_OPCODE
: CMPPLUS        PLUSopcode # j COMPILE_OPCODE
: CMPTWOSTAR     TWOSTARopcode # j COMPILE_OPCODE
: CMPTWOSLASH    TWOSLASHopcode # j COMPILE_OPCODE
: CMPPLUSSTAR    PLUSSTARopcode # j COMPILE_OPCODE
: CMPAFROM       AFROMopcode # j COMPILE_OPCODE
: CMPTOA         TOAopcode # j COMPILE_OPCODE
: CMPDUP         DUPopcode # j COMPILE_OPCODE
: CMPDROP        DROPopcode # j COMPILE_OPCODE
: CMPOVER        OVERopcode # j COMPILE_OPCODE
: CMPTOR         TORopcode # j COMPILE_OPCODE
: CMPRFROM       RFROMopcode # j COMPILE_OPCODE
: CMPNOP         NOPopcode # j COMPILE_OPCODE

: EXECUTE (>R) ;

```

```

: EXEC
c READ1
c EXECUTE
j EXEC

: NXEC
c SCAN
c LOOK
c POP_STRING
c EXECUTE
j NXEC

: TENSTAR
(DUP) (2*) (2*) (2*) (OVER) (+) (+) ;

: NUMI
l# INPUT (@)
(>R) (R@+)
n# 0 (>A)
(DUP)
if
: NUMI_LOOP
(A>) c TENSTAR (R@+) -n# 48 (+) (+) (>A)
-n# 1 (+) (DUP)
if
j NUMI_LOOP
else
else
(DROP) (R>) (DROP) (A>)
j POP_STRING

l NXEC l START_WORD !

save_vm_here vm_here @ l HERE !
save_vm_here_next vm_here_next @ l HERE_NEXT !
save_vm_slot vm_slot @ l SLOT !
save_vm_input vm_input @ l INPUT !
save_vm_there vm_there @ l THERE !
save_vm_name_end vm_name_end @ l NAME_END !

)VM

>$ FIF1 $> \n

```

### A.3.4 Flight Language Extensions

This is a repetition of the compilation of the Extensions from Section A.2, except that they are processed by the new kernel within the Virtual Machine. This indirection greatly alters and increases the nature of the code that is executed during the compilation of the extensions.

```
\n >$ FIFTEST1-VM-BEGIN $> \n  
run_vm  
  
// Extensions code goes here  
  
\n >$ FIFTEST1-VM-DONE $> \n
```

# Appendix B

## Static and Dynamic Gullwing Code Analyses

The following analyses are based on the software developed in Appendix A for the Gullwing processor. The analyses are imperfect since it cannot always be known if a memory word contains instructions or a literal. For example, this happens wherever the software contains directly accessed memory locations. These memory locations are neither executed nor accessed in-line like literal fetches. The analysis software considers these memory locations to contain instructions by default, despite actually being literals. Fortunately, such cases are infrequent and contribute very little noise to the data, showing up as rare, large literal values and UND instructions.

### B.1 Static Analyses

The static analyses are done on the binary code resident in memory after each test was compiled and run. The symbol table associated with the executable code was removed prior to analysis.

#### B.1.1 Memory Usage

Table B.1 shows the total size of the code measured in (32-bit) memory words. This is divided into words which contain either instructions or literal values. The literals are further divided as addresses and as actual literal values.

32-bit Words	Bare	Ratio	Ext.	Ratio	VM	Ratio
Total	286	1.000	1185	1.000	3029	1.000
Instruction	123	0.430	593	0.500	1565	0.517
Literal	163	0.570	592	0.500	1464	0.483
Literal Types (words)						
Literal	79	0.485	215	0.363	593	0.405
Address	84	0.515	377	0.637	871	0.595

Table B.1: Compiled Flight Code Memory Usage

## B.1.2 Range of Literals

Table B.2 shows the distribution of the required number of bits required to represent the absolute value of literals in immediate fetches. Most of these are small constants used in calculations.

Bits	Bare	Ratio	Ext.	Ratio	VM	Ratio
4	56	0.709	168	0.781	340	0.573
8	17	0.215	22	0.102	63	0.106
12	1	0.013	13	0.060	73	0.123
16	2	0.025	4	0.019	101	0.170
20	0	0.000	2	0.009	5	0.008
24	0	0.000	1	0.005	2	0.003
28	0	0.000	1	0.005	2	0.003
32	3	0.038	4	0.019	7	0.012

Table B.2: Range of Literals by Absolute Value

## B.1.3 Range of Addresses

Table B.3 shows the distribution of the required number of bits required to represent the absolute value of addresses used for calls and jumps. As the code size increases, the number of bits required to represent an address increases proportionally.

Bits	Bare	Ratio	Ext.	Ratio	VM	Ratio
4	3	0.036	22	0.058	34	0.039
8	70	0.833	187	0.496	224	0.257
12	7	0.083	164	0.435	516	0.592
16	1	0.012	2	0.005	95	0.109
20	0	0.000	0	0.000	0	0.000
24	2	0.024	0	0.000	0	0.000
28	0	0.000	2	0.005	2	0.002
32	1	0.012	0	0.000	0	0.000

Table B.3: Range of Addresses by Absolute Value

### B.1.4 Instructions per Instruction Word

Table B.4 shows the distribution of the number of instructions compiled into memory words. The PC@ (PC Fetch) instruction is not counted since it is used to fill the empty instruction slots. A memory word which contains zero instructions is thus actually filled with PC@ instructions.

#/Word	Bare	Ratio	Ext.	Ratio	VM	Ratio
0	0	0.000	32	0.054	126	0.081
1	28	0.228	242	0.408	595	0.380
2	38	0.309	132	0.223	347	0.222
3	3	0.024	44	0.074	140	0.089
4	16	0.130	34	0.057	75	0.048
5	7	0.057	18	0.030	44	0.028
6	31	0.252	91	0.153	238	0.152

Table B.4: Instructions per Instruction Word

### B.1.5 Instruction Density

Table B.5 shows the number of instructions per memory word. The averages are computed against both all memory words and against only those which contain instructions. The maximum number of instructions per any memory word is a function of the division between instruction and literal words seen in Table B.1.

Instr. per Mem. Word	Bare	Ext.	VM
Avg. per Any Word	1.392	1.190	1.207
Max. per Any Word	2.580	3.003	3.100
Avg. per Instruction Word	3.236	2.380	2.337
Total # of Instr.	398	1410	3657
Total Instr. Slots	738	3558	9390
Slot Usage	0.539	0.396	0.389

Table B.5: Instruction Density

## B.1.6 Compiled Instruction Counts

Table B.6 shows the number of times each possible instruction was found in memory. The ratios are calculated relative to the total number of instructions (C/I) and to the total number of instruction slots (C/S).

Instr.	Bare	C/I	C/S	Ext.	C/I	C/S	VM	C/I	C/S
JMP0	14	0.035	0.019	33	0.023	0.009	75	0.021	0.008
JMP+	0	0.000	0.000	19	0.013	0.005	40	0.011	0.004
CALL	30	0.075	0.041	244	0.173	0.069	566	0.155	0.060
RET	20	0.050	0.027	141	0.100	0.040	419	0.115	0.045
JMP	40	0.101	0.054	81	0.057	0.023	190	0.052	0.020
PC@	340	0.854	0.461	2148	1.523	0.604	5733	1.568	0.611
LIT	79	0.198	0.107	215	0.152	0.060	593	0.162	0.063
@A	22	0.055	0.030	54	0.038	0.015	154	0.042	0.016
@A+	4	0.010	0.005	4	0.003	0.001	17	0.005	0.002
!A	19	0.048	0.026	33	0.023	0.009	120	0.033	0.013
!A+	2	0.005	0.003	5	0.004	0.001	53	0.014	0.006
@R+	4	0.010	0.005	5	0.004	0.001	14	0.004	0.001
!R+	2	0.005	0.003	4	0.003	0.001	17	0.005	0.002
XOR	5	0.013	0.007	19	0.013	0.005	47	0.013	0.005
AND	5	0.013	0.007	8	0.006	0.002	19	0.005	0.002
NOT	2	0.005	0.003	18	0.013	0.005	43	0.012	0.005
2*	33	0.083	0.045	49	0.035	0.014	104	0.028	0.011
2/	0	0.000	0.000	17	0.012	0.005	39	0.011	0.004
+	16	0.040	0.022	83	0.059	0.023	184	0.050	0.020
+*	0	0.000	0.000	18	0.013	0.005	41	0.011	0.004
DUP	12	0.030	0.016	59	0.042	0.017	136	0.037	0.014
DROP	9	0.023	0.012	56	0.040	0.016	123	0.034	0.013
OVER	13	0.033	0.018	41	0.029	0.012	87	0.024	0.009
>R	6	0.015	0.008	43	0.030	0.012	109	0.030	0.012
R>	5	0.013	0.007	41	0.029	0.012	101	0.028	0.011
>A	43	0.108	0.058	93	0.066	0.026	286	0.078	0.030
A>	8	0.020	0.011	14	0.010	0.004	39	0.011	0.004
NOP	1	0.003	0.001	2	0.001	0.001	8	0.002	0.001
UND0	1	0.003	0.001	0	0.000	0.000	2	0.001	0.000
UND1	0	0.000	0.000	2	0.001	0.001	6	0.002	0.001
UND2	0	0.000	0.000	1	0.001	0.000	8	0.002	0.001
UND3	3	0.008	0.004	8	0.006	0.002	17	0.005	0.002

Table B.6: Compiled Instruction Counts

## B.2 Dynamic Analyses

The dynamic analyses are done on an execution trace log of each test in Appendix A. The 'Bare' case present in the static analyses is not included since the Flight language kernel is built into the simulator and does virtually nothing without external input.

### B.2.1 Overall Execution

Table B.7 lists the total number of executed instructions and the number of cycles they required. An average CPI (Cycles Per Instruction) of 1.3 is implied by these values.

Test	Ext.	VM
Instructions	5,018,751	204,325,372
Cycles	6,574,996	265,567,537

Table B.7: Overall Execution

### B.2.2 Executed Instruction Counts

Table B.8 lists the number of times each instruction was executed and its ratios relative to the total number of instructions ( $C/I$ ) and cycles ( $C/C$ ) executed. The conditional jumps are divided into taken and not taken instances since they have different cycle counts (2 and 1, respectively).

A fold is a case where the fetching of the next group of instructions (a PC@) is executed concurrently with the last instruction from the current group, thus occurring 'for free'. The implementation of this feature is detailed in Section 6.3.1. A PC@ (PC Fetch) instruction is executed after the last instruction when this folding cannot be done. The sum of the folds and PC@s is the total number of instruction fetches not originating from jumps, calls, or returns.

Instruction	Ext.	C/I	C/C	VM	C/I	C/C
JMP0	304,141	0.061	0.046	2,280,533	0.011	0.009
JMP+	63	0.000	0.000	63	0.000	0.000
JMP0 TAKEN	23,307	0.005	0.007	83,610	0.000	0.001
JMP+ TAKEN	137	0.000	0.000	1,875,075	0.009	0.014
CALL	320,857	0.064	0.098	10,143,368	0.050	0.076
RET	321,997	0.064	0.098	15,306,286	0.075	0.115
JMP	107,803	0.021	0.033	5,617,365	0.027	0.042
PC@	230,798	0.041	0.031	4,428,928	0.022	0.017
FOLDS	210,080	0.042	0.032	15,381,940	0.075	0.058
LIT	636,924	0.127	0.097	32,273,461	0.158	0.122
@A	321,272	0.064	0.098	16,753,698	0.082	0.126
@A+	320,744	0.064	0.098	1,546,398	0.008	0.012
!A	12,909	0.003	0.004	9,326,086	0.046	0.070
!A+	428	0.000	0.000	1272	0.000	0.000
@R+	120,753	0.024	0.037	560,414	0.003	0.004
!R+	6038	0.001	0.002	28,593	0.000	0.000
XOR	319,174	0.064	0.049	4,203,870	0.021	0.016
AND	2247	0.000	0.000	7,042,637	0.034	0.027
NOT	2249	0.001	0.000	3,758,267	0.018	0.014
2*	9914	0.002	0.002	32,069	0.000	0.000
2/	0	0.000	0.000	25,802,660	0.126	0.097
+	515,465	0.103	0.078	15,671,697	0.077	0.059
+*	0	0.000	0.000	0	0.000	0.000
DUP	212,572	0.042	0.032	10,923,102	0.053	0.041
DROP	103,643	0.021	0.016	511,936	0.003	0.002
OVER	6777	0.001	0.001	3,770,383	0.018	0.014
>R	104,923	0.021	0.016	7,110,502	0.035	0.027
R>	103,781	0.021	0.016	1,947,580	0.010	0.007
>A	633,291	0.126	0.096	21,482,640	0.105	0.081
A>	302,944	0.060	0.046	1,843,179	0.009	0.007
NOP	0	0.000	0.000	0	0.000	0.000
UND0	0	0.000	0.000	0	0.000	0.000
UND1	0	0.000	0.000	0	0.000	0.000
UND2	0	0.000	0.000	0	0.000	0.000
UND3	0	0.000	0.000	0	0.000	0.000

Table B.8: Executed Instruction Counts

### B.2.3 Average CPI

Table B.9 shows the computed average CPI (Cycles Per Instruction) values based on the instruction counts from Table B.8. The 'Worst' and 'Best' values are for the hypothetical boundary cases where all conditional jumps are taken or not, respectively.

Test	Ext.	VM
Best	1.305	1.290
Actual	1.310	1.300
Worst	1.371	1.311

Table B.9: Average CPI

### B.2.4 Instruction Types

Table B.10 lists the executed instructions grouped by type. The ratios are relative to the total number of instructions executed. The relative contribution to the CPI of each instruction type can be readily inferred from the product of the cycles and the frequencies.

Test			Extensions		Virtual Machine	
Instr. Type	Members	Cycles	Count	Freq.	Count	Freq.
Conditionals	JMP+, JMP0	1	304,204	0.061	2,280,596	0.011
Conditionals (Taken)	JMP+ TAKEN, JMP0 TAKEN	2	23,444	0.005	1,958,685	0.010
Subroutine	CALL, RET, JMP	2	750,657	0.150	31,067,019	0.152
Fetches	PC@, LIT	1	840,722	0.168	36,702,389	0.180
Load/Store	@A, @A+, !A, !A+, @R+, !R+	2	782,144	0.156	28,216,461	0.138
Arithmetic & Logic	XOR, AND, NOT, 2*, 2/, +, +*	1	849,649	0.169	56,510,900	0.277
Stack Manipulation	DUP, DROP, OVER, >R, R>, >A, A>	1	1,467,931	0.292	47,589,322	0.233
NOP/UND	NOP, UND[0-3]	1	0	0.000	0	0.000

Table B.10: Instruction Types

## B.2.5 Basic Block Length

Table B.11 lists the lengths, measured in instructions, of the basic blocks encountered during execution. Calls, returns, and jumps (taken or not) terminate a basic block. A block length of zero signifies two consecutive calls or jumps.

The odd peak at length 17 is the main loop of the VM, `do_next_instruction`, which was deliberately inlined into a single basic block for performance reasons.

Instructions	Ext.	Ratio	VM	Ratio
0	24,613	0.023	17,143,651	0.486
1	300,464	0.279	2,372,695	0.067
2	394	0.000	661,029	0.019
3	230,485	0.214	2,118,717	0.060
4	208,813	0.194	940,229	0.027
5	103,499	0.096	741,651	0.021
6	3292	0.003	1,099,523	0.031
7	100,262	0.093	719,878	0.020
8	100,824	0.094	394,367	0.011
9	3108	0.003	647,462	0.018
10	-	-	317,034	0.009
11	77	0.000	322,764	0.009
12	43	0.000	132,018	0.004
13	61	0.000	646,811	0.018
14	634	0.001	1,884,477	0.053
16	1728	0.002	3462	0.000
17	-	-	5,160,532	0.146
20	8	0.000	-	-
Total Blocks	1,078,305		35,306,300	
Average	3.654		4.787	

Table B.11: Basic Block Length

## B.2.6 Data Stack Depth

Table B.12 shows the distribution of the number of items on the Data Stack over the entire execution.

Depth	Ext.	Ratio	VM	Ratio
0	109,303	0.022	1,469,323	0.007
1	876,384	0.175	13,429,593	0.066
2	1,379,820	0.275	29,249,154	0.143
3	1,437,394	0.286	44,388,630	0.217
4	879,234	0.175	44,484,643	0.218
5	272,451	0.054	34,373,299	0.168
6	52,157	0.010	20,746,317	0.102
7	9314	0.002	9,725,059	0.048
8	2184	0.000	4,301,403	0.021
9	463	0.000	1,710,608	0.008
10	47	0.000	367,851	0.002
11	-	-	65,179	0.000
12	-	-	11,231	0.000
13	-	-	2677	0.000
14	-	-	405	0.000
Average	2.636		3.924	

Table B.12: Data Stack Depth

## B.2.7 Return Stack Depth

Table B.13 shows the distribution of the usage of the Return Stack over the entire execution. The depth of the Return Stack is usually equal to the call depth of the program, plus some transient, temporary storage.

Depth	Ext.	Ratio	VM	Ratio
0	5666	0.001	19,823	0.000
1	1,862,081	0.371	8,362,538	0.041
2	1,452,360	0.289	15,583,345	0.076
3	1,147,591	0.229	57,244,617	0.280
4	379,797	0.076	48,202,927	0.236
5	169,104	0.034	44,737,258	0.219
6	1978	0.000	17,140,510	0.084
7	174	0.000	10,796,692	0.053
8	-	-	1,452,808	0.007
9	-	-	778,445	0.004
10	-	-	6079	0.000
11	-	-	330	0.000
Average	2.110		4.037	

Table B.13: Return Stack Depth

# Bibliography

- [ABB64] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, Architecture of the IBM System 360, IBM Journal of Research and Development **8** (1964), no. 2, 87–101.
- [All85] Murray W Allen, Charles Hamblin (1922-1985), Aust. Comput. J. **17** (1985), no. 4, 194–195.
- [Bai94] Chris Bailey, HLL enhancement for stack based processors, EuroMicro Journal of Microprocessing and Microprogramming **40** (1994), 665–668.
- [Bai96] ———, Optimisation Techniques for Stack-Based Processors, PhD thesis, University of Teesside, UK, July 1996, Amongst other things, analyzes the use of stack buffers to reduce memory traffic.
- [Bai00] ———, Achieving minimal and deterministic interrupt execution in stack-based processor architectures., in EUROMICRO [DBL00], pp. 1368–.
- [Bai04] ———, A proposed mechanism for super-pipelined instruction-issue for ilp stack machines, DSD '04: Proceedings of the Digital System Design, EUROMICRO Systems on (DSD'04) (Washington, DC, USA), IEEE Computer Society, 2004, pp. 121–129.
- [Bar61a] R. S. Barton, A new approach to the functional design of a digital computer, AFIPS Conference Proceedings **19** (1961), 393–396, presented at IRE-AIEE-ACM Computer Conference, May 9-11, 1961.
- [Bar61b] ———, System description for an improved information processing machine, Proceedings of the 1961 16th ACM national meeting (New York, NY, USA), ACM Press, 1961, pp. 103.101–103.104.
- [Bar61c] Robert S. Barton, Functional design of computers, Commun. ACM **4** (1961), no. 9, 405.
- [Bar87] R. S. Barton, A new approach to the functional design of a digital computer, IEEE Annals of the History of Computing **09** (1987), no. 1, 11–15.
- [Bau60] Friedrich L. Bauer, The formula-controlled logical computer "STANISLAUS", Math. Tabl. Aids Comp **14** (1960), 64–67.

- [Bau90] F. L. Bauer, The cellar principle of state transition and storage allocation, IEEE Ann. Hist. Comput. **12** (1990), no. 1, 41–49.
- [Bau02] Friedrich L. Bauer, From the Stack Principle to ALGOL, pp. 26–42, in Broy and Denert [BD02], 2002, Points to possible earlier origins of stacks for computation.
- [BBG<sup>+</sup>60] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger, Report on the algorithmic language ALGOL 60, Commun. ACM **3** (1960), no. 5, 299–314.
- [BBRS58] F. L. Bauer, H. Bottenbruch, H. Rutishauser, and K. Samelson, Proposal for a universal language for the description of computing processes, pp. 355–373, in Carr [Car58], 1958.
- [BCM<sup>+</sup>70] G. Bell, R. Cody, H. McFarland, B. DeLagi, J. O’Laughlin, R. Noonan, and W. Wulf, A new architecture for mini-computers: the DEC PDP-11, Proc. AFIPS SJCC (1970), 657–675.
- [BD02] Manfred Broy and Ernst Denert (eds.), Software pioneers: contributions to software engineering, Springer-Verlag New York, Inc., New York, NY, USA, 2002.
- [BFPB97] Gerrit A. Blaauw and Jr. Frederick P. Brooks, Computer architecture: Concepts and evolution, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
- [BI86] Leo B. Brodie and FORTH Inc., Starting FORTH, second ed., Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.
- [Bla77] Russell P. Blake, Exploring a stack architecture, Computer **10** (1977), no. 5, 30–39, QA76.5.I54, not in ACM Digital Library.
- [Bla90] Mario De Blasi, Computer architecture, Addison-Wesley Longman Publishing Co., Inc., 1990.
- [Bro84] Leo B. Brodie, Thinking FORTH: a language and philosophy for solving problems, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.
- [BSa] Friedrich Ludwig Bauer and Klaus Samelson, French patent 1.204.424: Machine à calculer automatique et procédé pour son exploitation, Filed March 28, 1958. Delivered August 10, 1959. Published January 26, 1960.
- [BSb] ———, Gb patent 892,098: Improvements in and relating to computing machines, Filed March 31, 1958. Published MArch 21, 1962.
- [BSc] ———, German patent 1 094 019: Verfahren zur automatischen Verarbeitung von kodierten Daten und Rechenmaschine zur Ausübung des Verfahrens, Filed March 30, 1957. Granted December 1, 1960.

- [BSd] \_\_\_\_\_, Us patent 3,047,228: Automatic computing machines and method of operation, Filed March 28, 1958. Granted July 31, 1962.
- [BS94] C. Bailey and R. Sotudeh, HLL enhancement for stack based processors, Selected papers of the short notes session on Euromicro '94 (Amsterdam, The Netherlands, The Netherlands), Elsevier Science Publishers B. V., 1994, pp. 685–688.
- [Bul77] D. M. Bulman, Stack computers, *Computer* **10** (1977), no. 5, 14–16.
- [Bur63] Burroughs Corporation, Detroit, Michigan, Operational characteristics of the processors for the Burroughs B5000, 1963, Pub. No. 5000-21005, Revision A.
- [Bur67] Burroughs Corporation, Detroit, Michigan, Burroughs B5500 information processing systems reference manual, May 1967, Pub. No. 1021326.
- [Bur73] Burroughs Corporation, Detroit, Michigan, B7700 systems reference manual, 1973, Form 1060233.
- [Bur81] Burroughs Corporation, Detroit, Michigan, B6900 system reference manual, July 1981, Form 5100986.
- [Bur98] Stephen D. Burd, Systems architecture: Hardware and software in information systems, South-Western Publishing Company, 1998.
- [Car58] J. W. Carr (ed.), Summer school 1958, University of Michigan, 1958.
- [Cha95] Robert James Chapman, Stack quarks, Proc. Rochester Forth Conference on Emerging Technology (Rochester, New York) (Lawrence P. G. Forsley, ed.), University of Rochester, The Institute for Applied Forth Research, Inc., June 1995, Decomposes the typical stack permutation operations into smaller primitives. Online as of Nov. 2006: <http://www.compumart.ab.ca/rc/Papers/StackQuarksPaper.pdf>.
- [Cha97] \_\_\_\_\_, A writable computer, Proc. Rochester Forth Conference on Emerging Technology (Rochester, New York) (Lawrence P. G. Forsley, ed.), University of Rochester, The Institute for Applied Forth Research, Inc., June 1997, Describes the VHDL design of a small stack computer. Online as of Nov. 2006: <http://www.compumart.ab.ca/rc/Papers/writablecomputer.pdf>.
- [Cha98] Rob Chapman, A Stack Processor: Synthesis, web page, January 1998, Project Report for EE602. Online as of Nov. 2006: <http://www.compumart.ab.ca/rc/Papers/spsynthesis.pdf>.
- [Chu75] Yaohan Chu, High-level language computer architecture, Academic Press, Inc., Orlando, FL, USA, 1975.
- [DBL00] 26th EUROMICRO 2000 conference, informatics: Inventing the future, 5-7 september 2000, Maastricht, The Netherlands, IEEE Computer Society, 2000.

- [Dor75a] R. W. Doran, The International Computers Ltd. ICL2900 computer architecture, SIGARCH Comput. Archit. News **4** (1975), no. 3, 24–47.
- [Dor75b] Robert W. Doran, Architecture of stack machines, pp. 63–108, in [Chu75], 1975.
- [DP80] David R. Ditzel and David A. Patterson, Retrospective on high-level language computer architecture, ISCA '80: Proceedings of the 7th annual symposium on Computer Architecture (New York, NY, USA), ACM Press, 1980, pp. 97–104.
- [DP86] ———, Retrospective on high-level language computer architecture, pp. 44–51, in [FL86], 1986.
- [DP98a] ———, Retrospective: a retrospective on high-level language computer architecture, ISCA '98: 25 years of the international symposia on Computer architecture (selected papers) (New York, NY, USA), ACM Press, 1998, pp. 13–14.
- [DP98b] ———, Retrospective on high-level language computer architecture, ISCA '98: 25 years of the international symposia on Computer architecture (selected papers) (New York, NY, USA), ACM Press, 1998, pp. 166–173.
- [Dun77] Fraser George Duncan, Stack machine development: Australia, great britain, and europe, Computer **10** (1977), no. 5, 50–52.
- [Eng63] English Electric-LEO Computers Ltd., Kidsgrove, Stoke-On-Trent, Staffordshire, England, KDF9 programming manual, circa 1963, Online as of March 2006 at: <http://www.jeays.ca/kdf9.html> and <http://frink.ucg.ie/~bfoley/edhist/kdf9pm/kdf9pm.html> and <http://acms.synonet.com/deuce/KDF9pm.pdf>.
- [FL86] Eduardo B. Fernandez and Tomas Lang, Software-oriented computer architecture, IEEE Computer Society Press, Los Alamitos, CA, USA, 1986.
- [Fox98] Jeff Fox, F21 CPU, web page, 1998, Online as of April 2007: <http://ultratechnology.com/f21.html>.
- [Fox04] ———, Forth Chips, web page, 2004, Online as of April 2007: <http://ultratechnology.com/chips.htm>.
- [FR93] James M. Feldman and Charles Retter, Computer architecture; a designer's text based on a generic RISC, McGraw-Hill, Inc., 1993.
- [Fre98] Paul Frenger, Forth in space, or, so NEAR yet so far out, SIGPLAN Not. **33** (1998), no. 6, 24–26.
- [Fre01] ———, Close encounters of the Forth kind, SIGPLAN Not. **36** (2001), no. 4, 21–24.
- [GL03] William F. Gilreath and Phillip A. Laplante, Computer Architecture, Kluwer Academic Publishers, 2003.

- [GWS91] II George William Shaw, Sh-BOOM: the sound of the RISC market changing, Proceedings of the second and third annual workshops on Forth, ACM Press, 1991, p. 125.
- [Hal62] A. C. D. Haley, The KDF.9 computer system, Proceedings of the AFIPS Fall Joint Computer Conference, vol. 21, 1962, pp. 108–120.
- [Ham57a] Charles L. Hamblin, An addressless coding scheme based on mathematical notation, Proceedings of the First Australian Conference on Computing and Data Processing (Salisbury, South Australia: Weapons Research Establishment), Jun 1957.
- [Ham57b] ———, Computer languages, The Australian Journal of Science **20** (1957), 135–139.
- [Ham85] ———, Computer languages, Aust. Comput. J. **17** (1985), no. 4, 195–198, Reprint of 1957 paper in volume 20 of The Australian Journal of Science.
- [Hay97] John P. Hayes, Computer architecture and organization, McGraw-Hill, Inc., 1997.
- [Hen84] John L. Hennessy, VLSI processor architecture, IEEE Transaction on Computers **C-33** (1984), no. 12, 1221–1246.
- [Hen86] ———, VLSI processor architecture, pp. 90–115, in [FL86], 1986.
- [Hew84] Hewlett-Packard, Cupertino, California, HP 3000 computer systems general information manual, October 1984, Pub. No. 5953-7983.
- [HH00] Richard E. Haskell and Darrin M. Hanna, Implementing a Forth engine microcontroller on a Xilinx FPGA, Looking Forward – The IEEE Computer Society’s Student Newsletter (A Supplement to Computer) **8** (2000), no. 1, Online as of Nov. 2006: <http://www.cse.secs.oakland.edu/hanna/research/IEEE2.pdf> and <http://www.cse.secs.oakland.edu/haskell/VHDL/IEEEStudent.PDF>.
- [HJB<sup>+</sup>82] John Hennessy, Norman Jouppi, Forest Baskett, Thomas Gross, and John Gill, Hardware/software tradeoffs for increased performance, ASPLOS-I: Proceedings of the first international symposium on Architectural support for programming languages and operating systems (New York, NY, USA), ACM Press, 1982, pp. 2–11.
- [HJP<sup>+</sup>82] John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, Thomas Gross, Forest Baskett, and John Gill, MIPS: A microprocessor architecture, MICRO 15: Proceedings of the 15th annual workshop on Microprogramming (Piscataway, NJ, USA), IEEE Press, 1982, pp. 17–22.
- [HP96] John L. Hennessy and David A. Patterson, Computer architecture: A quantitative approach, 2nd ed., Morgan Kaufmann Publishers Inc., 1996, The dominant textbook on current computer architecture, which I’ve found to be erroneous with regards to current stack architecture.

- [HP02] \_\_\_\_\_, Computer architecture: A quantitative approach, Morgan Kaufmann Publishers Inc., 2002.
- [HVZ95] V. Carl Hamacher, Zvonko G. Vranesic, and Safwat G. Zaky, Computer organization, McGraw-Hill Higher Education, 1995.
- [Hwa92] Kai Hwang, Advanced computer architecture: Parallelism, scalability, programmability, McGraw-Hill Higher Education, 1992.
- [IC78] R. N. Ibbett and P. C. Capon, The development of the MU5 computer system, Commun. ACM **21** (1978), no. 1, 13–24.
- [Int00] Intersil, Data sheet for HS-RTX2010RH, March 2000, File Number 3961.3.
- [Int06] Intel Corporation, Dever, CO, Intel® 64 and IA-32 architectures optimization reference manual, November 2006, Order Number: 248966-014.
- [Kat85] Manolis G. H. Katevenis, Reduced instruction set computer architectures for VLSI, Massachusetts Institute of Technology, Cambridge, MA, USA, 1985, One of the PhD theses from the Berkeley RISC project. Goes into the deep technical details and reasonings behind RISC.
- [KDF61] KDF9: Very high speed data processing system for commerce, industry, science, English Electric, Kidsgrove, Stoke-On-Trent, Staffordshire, England, 1961, Sales brochure for the KDF9.
- [KKC92a] William F. Keown, Philip Koopman, and Aaron Collins, Performance of the Harris RTX 2000 stack architecture versus the Sun 4 SPARC and the Sun 3 M68020 architectures, SIGARCH Comput. Archit. News **20** (1992), no. 3, 45–52.
- [KKC92b] \_\_\_\_\_, Real-time performance of the Harris RTX 2000 stack architecture versus the Sun 4 SPARC and the Sun 3 M68020 architectures with a proposed real-time performance benchmark, SIGMETRICS Perform. Eval. Rev. **19** (1992), no. 4, 40–48.
- [Kog90] Peter M. Kogge, The architecture of symbolic computers, McGraw-Hill, Inc., 1990.
- [Koo89] Philip J. Koopman, Stack computers: the new wave, Halsted Press, 1989, A compendium of stack computer architectures. Has useful experimental data.
- [Koo90] \_\_\_\_\_, Modern stack computer architecture, System Design and Network Architecture Conference (1990), 153–164.
- [Koo91] \_\_\_\_\_, Some ideas for stack computer design, Rochester Forth Conference (1991), 58.
- [Koo94] \_\_\_\_\_, A preliminary exploration of optimized stack code generation, Journal of Forth Applications and Research **6** (1994), no. 3, 241–251.

- [Lav80] Simon Hugh Lavington, Early british computers: The story of vintage computers and the people who built them, Butterworth-Heinemann, Newton, MA, USA, 1980.
- [LTL98] P. H. W. Leong, P. K. Tsang, and T. K. Lee, A FPGA based forth microprocessor, FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (Washington, DC, USA), IEEE Computer Society, 1998, p. 254.
- [Luk29] Jan Lukasiewicz, Elements of mathematical logic, Warsaw, 1929, [English translation of 1958 edition: Macmillan, 1963].
- [McK80] William M. McKeeman, Stack computers, pp. 319–362, in [Sto80], 1980.
- [McL93] Edward McLellan, The Alpha AXP architecture and 21064 processor, IEEE Micro **13** (1993), no. 3, 36–47.
- [ME97] Martin Maierhofer and M. Anton Ertl, Optimizing stack code, Forth-Tagung 1997, 1997.
- [ME98] ———, Local stack allocation, Compiler Construction 1998, Springer LNCS 1383, 1998, pp. 189–203.
- [MK97] M. Morris Mano and Charles R. Kime, Logic and computer design fundamentals, Prentice-Hall, Inc., 1997.
- [ML70] Charles H. Moore and Geoffrey C. Leach, FORTH – a language for interactive computing, Mohasco Industries, Inc., Amsterdam, NY, 1970, Internal publication.
- [Moo91] Charles H. Moore, Forth - the early years, Unpublished notes that became the papers by Rather, Colburn, and Moore [RCM93] [RCM96]. Accesible at <http://www.colorforth.com/HOPL.html> as of Nov. 2006., 1991.
- [Moo01a] ———, 25x emulator, Proceedings of the 17th EuroForth Conference (Schloss Dagstuhl, Saarland, Germany), University of Teesside, November 2001, ISBN: 0 907550 97 6.
- [Moo01b] ———, c18 colorForth compiler, Proceedings of the 17th EuroForth Conference (Schloss Dagstuhl, Saarland, Germany), University of Teesside, November 2001, One of the few published papers by Chuck Moore. Describes the c18 instruction set in detail.
- [MP95] Silvia M. Muller and Wolfgang J. Paul, The complexity of simple computer architectures, Springer-Verlag New York, Inc., 1995.
- [MT95] Charles H. Moore and C. H. Ting, MuP21 – a MISC processor, Forth Dimensions (1995), 41, <http://www.ultratechnology.com/mup21.html>.
- [Mur86] Robert W. Murphy, Under the hood of a superchip: the NOVIX Forth engine, J. FORTH Appl. Res. **3** (1986), no. 2, 185–188.

- [Mur90] William D. Murray, Computer and digital system architecture, Prentice-Hall, Inc., 1990.
- [Omo94] Amos R. Omondi, Computer arithmetic systems: algorithms, architecture and implementation, Prentice Hall International (UK) Ltd., 1994.
- [Omo99] \_\_\_\_\_, The microarchitecture of pipelined and superscalar computers, Kluwer Academic Publishers, 1999.
- [Org73] Elliott Irving Organick, Computer system organization: The B5700/B6700 series (ACM monograph series), Academic Press, Inc., Orlando, FL, USA, 1973.
- [Pat85] David A. Patterson, Reduced instruction set computers, Commun. ACM **28** (1985), no. 1, 8–21.
- [Pat86] \_\_\_\_\_, Reduced instruction set computers, pp. 76–89, in [FL86], 1986.
- [Pay96] Bernd Paysan, Implementation of the 4stack processor using Verilog, Diploma thesis, Technische Universitat Munchen, Institut fur Informatik, August 1996, <http://www.jwtdt.com/~paysan/4stack.html>.
- [Pay02] Berndt Paysan, b16–A Forth processor in an FPGA, Forth-Tagung 2002 (2002), <http://www.b16-cpu.de/>.
- [PD80] David A. Patterson and David R. Ditzel, The case for the reduced instruction set computer, SIGARCH Comput. Archit. News **8** (1980), no. 6, 25–33.
- [PH90] David A. Patterson and John L. Hennessy, Computer architecture: a quantitative approach, Morgan Kaufmann Publishers Inc., 1990.
- [PH98] \_\_\_\_\_, Computer organization and design: the hardware/software interface, 2nd ed., Morgan Kaufmann Publishers Inc., 1998.
- [PS81] David A. Patterson and Carlo H. Sequin, RISC I: A reduced instruction set VLSI computer, ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture (Los Alamitos, CA, USA), IEEE Computer Society Press, 1981, pp. 443–457.
- [PS98a] David A. Patterson and Carlo H. Sequin, Retrospective: RISC I: a reduced instruction set computer, ISCA '98: 25 years of the international symposia on Computer architecture (selected papers) (New York, NY, USA), ACM Press, 1998, pp. 24–26.
- [PS98b] David A. Patterson and Carlo H. Sequin, RISC I: a reduced instruction set VLSI computer, ISCA '98: 25 years of the international symposia on Computer architecture (selected papers) (New York, NY, USA), ACM Press, 1998, pp. 216–230.

- [Ras03] James Rash, Space-related applications of forth, webpage: <http://forth.gsfc.nasa.gov/>, April 2003, Presents space-related applications of Forth microprocessors and the Forth programming language at NASA. Accessed on Nov. 2006.
- [RCM93] Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore, The evolution of Forth, The second ACM SIGPLAN conference on History of programming languages (Cambridge, Massachusetts, United States), ACM Press, 1993, pp. 177–199.
- [RCM96] ———, The evolution of Forth, History of programming languages—II (New York, NY, USA), ACM Press, 1996, pp. 625–670.
- [Ros87] Robert F. Rosin, Prologue: The Burroughs B5000, Annals of the History of Computing **9** (1987), no. 1, 6–7.
- [SB60] K. Samelson and F. L. Bauer, Sequential formula translation, Commun. ACM **3** (1960), no. 2, 76–83.
- [SB04] Huibin Shi and Chris Bailey, Investigating available instruction level parallelism for stack based machine architectures, DSD '04: Proceedings of the Digital System Design, EUROMICRO Systems on (DSD'04) (Washington, DC, USA), IEEE Computer Society, 2004, pp. 112–120.
- [Sha99] George William Shaw, PSC1000 microprocessor reference manual, Patriot Scientific Corporation, San Diego, CA, March 1999, Ref. No. 99-0370001.
- [Sha02] ———, IGNITE intellectual property reference manual, Patriot Scientific Corporation, San Diego, CA, March 2002, Revision 1.0.
- [Sit78] Richard L. Sites, A combined register-stack architecture, SIGARCH Comput. Archit. News **6** (1978), no. 8, 19–19.
- [SSK97] Dezso Sima, D. Sima, and Peter Kacsuk, Advanced computer architectures, Addison-Wesley Longman Publishing Co., Inc., 1997.
- [Sta90] William Stallings, Computer organization and architecture, 2nd ed., Prentice Hall PTR, 1990.
- [Sta93] ———, Computer organization and architecture: principles of structure and function, 3rd ed., Macmillan Publishing Co., Inc., 1993.
- [Sta02] ———, Computer organization and architecture, Prentice Hall Professional Technical Reference, 2002.
- [Sto80] Harold S. Stone, Introduction to computer architecture, Science Research Associates, 1980.
- [Sto92] ———, High-performance computer architecture, Addison-Wesley Longman Publishing Co., Inc., 1992.

- [TCCLL99] P.K. Tsang, K.H. C.C. Cheung, T.K. Lee Leung, and P.H.W. Leong, MSL16A: an asynchronous Forth microprocessor, TENCON 99. Proceedings of the IEEE Region 10 Conference, vol. 2, September 1999, pp. 1079–1082.
- [Tin97a] C H Ting, The P series of microprocessors, More On Forth Engines **22** (1997), 1–17.
- [Tin97b] ———, P16 microprocessor design in VHDL, More On Forth Engines **22** (1997), 44–51.
- [Wil91] Barry Wilkinson, Computer architecture: Design and performance, 1st ed., Prentice-Hall, Inc., 1991.
- [Wil96] ———, Computer architecture: Design and performance, 2nd ed., Prentice-Hall, Inc., 1996.
- [Wil01] Rob Williams, Computer systems architecture with CDROM, Addison-Wesley Longman Publishing Co., Inc., 2001.
- [Yue] C K Yuen, Superscalar execution of stack programs using reorder buffer, <http://www.comp.nus.edu.sg/~yuenck/stack>.