The *fun* Instruction-set Architecture Manual
v.0.1

Contributors and affiliations (in alphabetical order) : Cecil Accetti (SJTU), Peilin Liu (SJTU), Eshton Robateau, Da Shi (SJTU), Rendong Ying (SJTU).

Contents

1 About this Manual **3**

2 Introduction **4**
2.1 Features 4

3 A Timeline of Functional Programming and Machine-Support for Functional Programming **5**
3.1 Foundations 5
3.2 Technology Trigger 5
3.3 Peak of Inflated Expectations 6
3.4 Disillusionment 7
3.5 Enlightment 7
3.6 Reattempts 7

4 Combinator Graph Reduction **8**

5 Instruction Formats **9**

6 Combinators **11**

7 Integer Instructions **12**

8 Floating-Point Instructions **13**

9 Input-Output **14**

10 Floating-Point Instructions **15**

11 Instruction Listings **16**

Chapter 1

About this Manual

fun is a purely-functional instruction set architecture that defines a language based on structured combinators, for applications where isolation, purity and the way computation is actually performed are the central concerns.

This is the first draft of a document that describes the *fun* instruction-set architecture (ISA). This manual does not describe any implementation-specific details such as reduction model, evaluation order or hardware structures such as registers, caches, memories, bus interfaces, garbage collectors and other memory management units.

This document is open to contributions from anyone interested to participate in the *fun* project, and as such, it is a *wok in progress.* The information contained in this manual may change as the architecture and its implementations evolve.

Other relevant material about *fun* should be also available on the project `wiki`, at `http://wiki.fun-arch.org`.

If you wish to contribute on this manual or the wiki, send a request to the email `join@fun-arch.org`.

Chapter 2

Introduction

2.1 Features

fun is a purely-functional instruction set architecture that defines a language based on structured combinators, for applications where isolation, purity and the way computation is actually performed are the central concerns. *fun* is:

- An open, free and community-driven instruction-set architecture;
- The first purely-functional instruction set based on combinators to follow a modern, proven path of other RISC architectures;
- A purely-functional instruction set for which memory handling, control flow and other stateful and effectful behaviors is *unrepresentable*.
- An instruction set for efficient implementation of high-level purely-functional programming languages.

Chapter 3

A Timeline of Functional Programming and Machine-Support for Functional Programming

3.1 Foundations

- 1924 – M.Schonfinkel: Uber die Bausteine der Mathematischen Logik
- 1930 – H.B.Curry: Grundlagen der Kombinatorischen Logik
- 1934 – H.B.Curry: Functionality in Combinatory Logic
- 1958 – H.B.Curry, R.Feys: Combinatory Logic (Book)

3.2 Technology Trigger

- 1971 – C.P.Wadsworth: Semantics and Pragmatics of the λ-calculus
- 1977 – J.Backus: Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs
- 1979 – D.Turner: A new Implementation Technique for Applicative Languages
- 1979 – D.Turner: Another Algorithm for Bracket Abstraction
- 1982 – D.Turner: Miranda

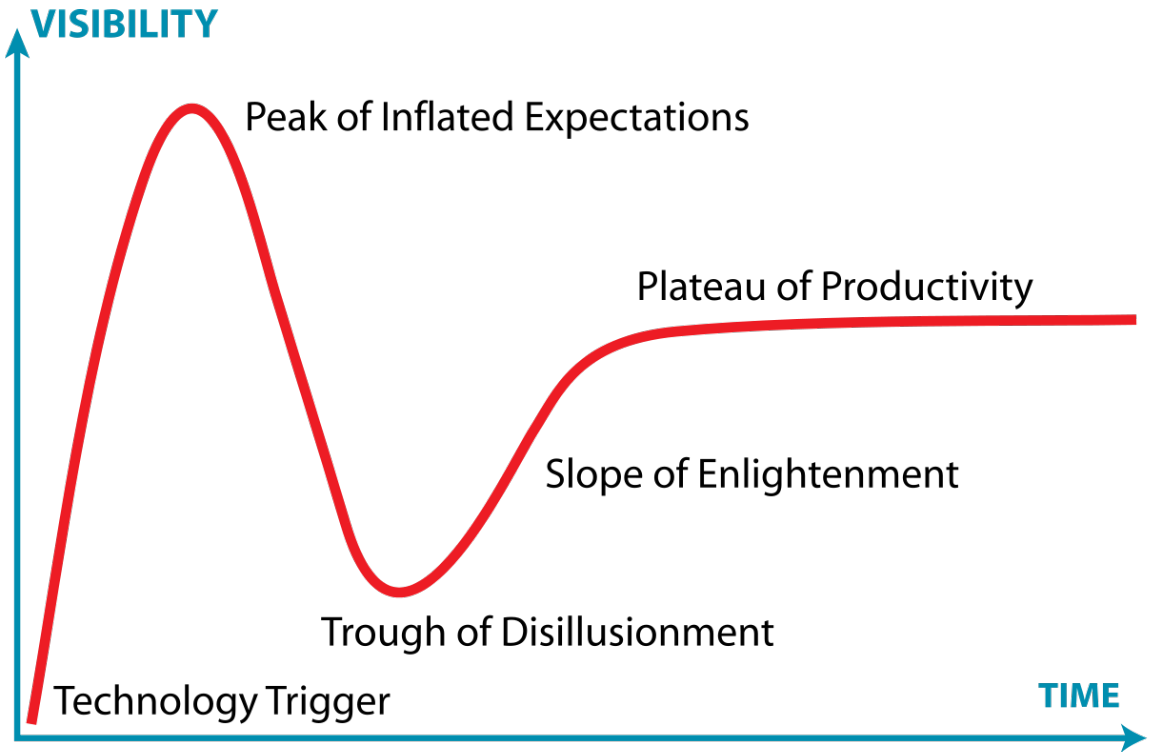

Figure 3.1: Hype cycle

3.3 Peak of Inflated Expectations

LISP Machines

Combinator Architectures

- 1980 – T.J.W.Clarke: SKIM – The S, K, I Reduction Machine
- 1984 – W.R.Stoye: Some Practical Methods for Rapid Combinator Reduction
- 1985 – W.Stoye: Message-based Functional Operating Systems
- 1986 – M.Scheevel: NORMA, A Graph Reduction Processor
- 1986 – J.Ramsdell: The CURRY chip
- 1990 – P.J.Koopman: Implementation of the TIGRE Machine

Parallel & Dataflow

- 197? – Arvind: Dataflow
- 1986 – C.Clack, SPJ: The Four-stroke Reduction Engine
- 1988 – SPJ,et al: GRIP

3.4 Disillusionment

Functional Programming in Stock Hardware

- 1982 – J.Hughes: Supercombinators- A new implementation method for applicative languages
- 1984 – G-Machine
- 1987 – Spineless Tagless G-machine
- 1987 – Haskell

3.5 Enlightment

- Category Theory: Morphisms
- Haskell, OCaml, Erlang, Elm: real-world functional programming languages
- Mainstream FP: Immutable data, higher-order functions on Python, JavaScript, Java, C++
- Industrial use of FP: Specification, theorem proofing, finance, safety

3.6 Reattempts

- 2007 – Naylor: Reduceron
- 2010 – Naylor: Reduceron Reconfigured
- 2017 – McMahan: An Architecture Supporting Formal and Compositional Binary Analysis (Zarf)
- 2020 – Pope: Cephalopode
- 2020 – Coelho: ACQuA

Chapter 4

Combinator Graph Reduction

This chapter is a stub for a discussion on Combinator Graph Reduction.

Chapter 5

Instruction Formats

This chapter describes the instruction encoding of the *fun* ISA, its formats and types.

fun instructions are divided into 4 types (C/I^2/A/V), as depicted in Fig.5.1. Each instruction fits a 36-bit word, with a 4-bit *tag*, and a 36-bit *payload*. Each *tag* is composed by an *eval bit* EV and a type identifier (OP/LIT/LINK/HLINK/ROOT). Table5.1 lists the base types for *fun* , as of version 0.1.

A *fun* instruction can be seen as a graph node for evaluation following a graph reduction strategy.

35	34 32	31 29	28 26	25 23	22 20	19 17	16 14	13 11	10 5	4 0	
EV	OP	a_6	a_5	a_4	a_3	a_2	a_1	arity	type	opcode	C-type
EV	OP	imm[31:11]							type	opcode	I^2-type
EV	OP	0							type	opcode	A-type
EV	LIT	Integer/ Float[31:0]/ Complex{Im[31:16],Re[15:0]}									V-type
EV	LINK	Address[31:0]									V-type
EV	HLINK	Address[31:0]									V-type

Figure 5.1: Instruction formats for Fun, extended with RISC-V base instructions. A tag[34:32] indicates the type of the instruction.

Type `LINK` and `HLINK` identify references to other nodes or strings of combinators.

Table 5.1: Basic types

Type	**Encoding**
LINK	011
HLINK	110
LIT	001
OP	010
ROOT	111

Type `LINK` is reserved for references made a compile time, by a programmer or compiler, for references to other nodes on the program graph (subgraphs, functions or bus locations).

Type `HLINK` is reserved for references made on runtime by a *fun* CPU, and should not be declared on the initial program graph.

Type `OP` is reserved for *combinators* and other operations for integer arithmetic, floating-point arithmetic, bitwise boolean logic, input-output, and other types defined in the base ISA specification.

Type `LIT` is reserved for pure integer or single-precision floating-point values.

Type `ROOT` is reserved to identify the root node of a program graph.

Chapter 6

Combinators

Combinators are instructions of type C that operate on the program graph changing its structure, following the rules specified in the instruction body.

A reduction rule is specified by the *reduction pattern*, *arity* and the contents of each new graph node after reduction.

The base specification for *fun* supports up to 64 reduction patterns, listed on Appendix A.

35	34 32	31 0	34 32	34 32	34 32	34 32	34 32	34 32	10 5	4 0
ev	type	a6	a5	a4	a3	a2	a1	arity	pattern	opcode
1	3	3	3	3	3	3	3	3	6	5
EV	OP	a6	a5	a4	a3	a2	a1	num	T0-T63	COMBI

Chapter 7

Integer Instructions

This chapter is a stub for a detailed description of the instructions for integer arithmetic and logic of the *fun* ISA.

Chapter 8

Floating-Point Instructions

This chapter is a stub for a detailed description of the instructions for floating-point arithmetic of the *fun* ISA.

Chapter 9

Input-Output

This chapter is a stub for a detailed description of the instructions and mechanisms for I/O in *fun* .

Chapter 10

Floating-Point Instructions

This chapter is a stub for a detailed description of the instructions for floating-point arithmetic of the *fun* ISA.

Chapter 11

Instruction Listings

The listing for the base set of *fun* instructions is shown on Table 11.1.

35	34 32	31 29	28 26	25 23	22 20	19 17	16 14	13 11	10 5	4 0	
EV	OP	a_6	a_5	a_4	a_3	a_2	a_1	arity	type	opcode	C-type
EV	OP	imm[31:11]							type	opcode	I^2-type
EV	OP	0							type	opcode	A-type

Base Instruction Set

EV	OP	a_6	a_5	a_4	a_3	a_2	a_1	arity	type	11000	combi

EV	010	0	000000	10001	add
EV	010	0	000001	10001	sub
EV	010	0	000011	10001	sll
EV	010	0	000111	10001	srl
EV	010	0	001000	10001	sra
EV	010	0	001011	10001	mul
EV	010	0	001111	10001	div
EV	010	0	010001	10001	rem
EV	010	0	010101	10001	eq
EV	010	0	010110	10001	gt
EV	010	0	010111	10001	lt
EV	010	0	001010	10001	and
EV	010	0	001011	10001	or
EV	010	0	001100	10001	xor

EV	010	imm[31:11]	000000	10011	addi
EV	010	imm[31:11]	000011	10011	slli
EV	010	imm[31:11]	000111	10011	srli
EV	010	imm[31:11]	001000	10011	srai
EV	010	imm[31:11]	001011	10011	muli
EV	010	imm[31:11]	001111	10011	divi
EV	010	imm[31:11]	010001	10011	remi
EV	010	imm[31:11]	010101	10011	eqi
EV	010	imm[31:11]	010110	10011	gti
EV	010	imm[31:11]	010111	10011	lti
EV	010	imm[31:11]	001010	10011	andi
EV	010	imm[31:11]	001011	10011	ori
EV	010	imm[31:11]	001100	10011	xori

EV	010	0	000001	11110	fix
EV	010	0	000100	11110	seq
EV	010	0	010000	11110	out
EV	010	0	001010	11110	break

Table 11.1: Instruction listing for *fun*

Bibliography

[1] The unlambda programming language. http://www.madore.org/~david/programs/unlambda/. Accessed: 2018-8-5.

[2] Andrea Asperti, Cecilia Giovanetti, and Andrea Naletto. The bologna optimal higher-order machine. *J. Funct. Program.*, 6(6):763–810, 1996.

[3] Lex Augusteijn. Sorting morphisms. In *Advanced Functional Programming*. Springer, 1999.

[4] Lennart Augustsson. Bwm: A concrete machine for graph reduction. In *Proceedings of the 1991 Glasgow Workshop on Functional Programming*, pages 36–50, Berlin, Heidelberg, 1992. Springer-Verlag.

[5] John Backus. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 1978.

[6] R. S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.

[7] Arjan Boeijink, Philip K. F. Hölzenspies, and Jan Kuper. Introducing the pilgrim: A processor for executing lazy functional languages. In *Implementation and Application of Functional Languages*. Springer, 2011.

[8] U. Boquist and T. Johnsson. The grin project: A highly optimising back end for lazy functional languages. In *Implementation of Functional Languages*. Springer, 1997.

[9] Sabine Broda and Luís Damas. Compact bracket abstraction in combinatory logic. *The Journal of Symbolic Logic*, 62, 1997.

[10] T. J.W. Clarke, P. J.S. Gladstone, C. D. MacLean, and A. C. Norman. Skim - the s, k, i reduction machine. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, 1980.

[11] Antoni Diller. Efficient multi-variate abstraction using an array representation for combinators. *Inf. Process. Lett.*, 84:311–317, 2002.

[12] Antoni Diller. Efficient bracket abstraction using iconic representations for combinators. 2011.

[13] Conal Elliott. Compiling to categories. *Proc. ACM Program. Lang.*, 2017.

[14] Ricardo Coelho et al. Acqua:a parallel accelerator architecture for pure functional programs. In *Proc.IEEE Comp.Soc.Annual Symp.on VLSI*, 2020.

[15] Federico Flaviani and Elias Tahhan-Bittar. Criteria for bracket abstractions design. In *CLEI Selected Papers*, 2020.

[16] Donald I. Good. Provable programming. In *Proceedings of the International Conference on Reliable Software*, pages 411–419, New York, NY, USA, 1975. ACM.

[17] E. Goto, T. Soma, N. Inada, T. Ida, M. Idesawa, K. Hiraki, M. Suzuki, K. Shimizu, and B. Philipov. Design of a lisp machine - flats. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, LFP '82, pages 208–215, New York, NY, USA, 1982. ACM.

[18] A. J. Harris and J. R. Hayes. Functional programming on a stack-based embedded processor. In *2nd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'06)*, pages 7 pp.–424, July 2006.

[19] Pieter H. Hartel and Arthur H. Veen. Statistics on graph reduction of sasl programs. *Softw. Pract. Exper.*, 18(3), 1988.

[20] Nevin Heintze and Jon G. Riecke. The slam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM SIGPLAN-SIGACT Symp. POPL*. ACM, 1998.

[21] Paul Hudak and Benjamin Goldberg. Serial combinators: "optimal" grains of parallelism. In *Proc. of a Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag, 1985.

[22] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2), 1989.

[23] R. J. M. Hughes. Super-combinators a new implementation method for applicative languages. In *Proc.1982 ACM Symp.LISP and Functional Programming*. ACM, 1982.

[24] Thomas Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '84. ACM, 1984.

[25] Neil D. Jones and Steven S. Muchnick. A fixed-program machine for combinator expression evaluation. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, LFP '82, pages 11–20, New York, NY, USA, 1982. ACM.

[26] Simon L Peyton Jones. An investigation of the relative efficiencies of combinators and lambda expressions. In *Proc.1982 ACM Symp.LISP and Functional Programming*, page 150–158. ACM, 1982.

[27] Richard B. Kieburtz. The g-machine: A fast, graph-reduction evaluator. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 400–413, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[28] Richard B. Kieburtz. A risc architecture for symbolic computation. In *Proc.2nd Int.Conf.Architectural Support for Programming Languages and Operating Systems*. IEEE Comp.Soc.Press, 1987.

[29] Oleg Kiselyov. λ to ski, semantically - declarative pearl. In *Proc.14th Int.Symp.Functional and Logic Programming*, 2018.

[30] Philip John Koopman. Implementation of the tigre machine. In *An Architecture for Combinator Graph Reduction*. Academic Press, 1990.

[31] Lukasz Lachowski. On the complexity of the standard translation of lambda calculus into combinatory logic. *Reports Math. Log.*, 53:19–42, 2018.

[32] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, March 1966.

[33] Rafael D. Lins. Categorical multi-combinators. In *Proc. of a Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag, 1987.

[34] R. Machado. An introduction to lambda calculus and functional programming. In *2013 2nd Workshop-School on Theoretical Computer Science*, pages 26–33, Oct 2013.

[35] Joseph McMahan, Michael Christensen, Lawton Nichols, Jared Roesch, Sung-Yee Guo, Ben Hardekopf, and Timothy Sherwood. An architecture supporting formal and compositional binary analysis. In *Proc. 22nd ASPLOS*. ACM, 2017.

[36] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*. Springer, 1991.

[37] D. Moon. Symbolics architecture. *Computer*, 20(1):43–52, Jan 1987.

[38] M.W.Bunder. Some improvements to turner's algorithm for bracket abstraction. *J.Symb.Log.*, 55, 1990.

[39] Matthew Naylor and Colin Runciman. The reduceron: Widening the von neumann bottleneck for graph reduction using an fpga. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *Implementation and Application of Functional Languages*, pages 129–146, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[40] Matthew Naylor and Colin Runciman. The reduceron reconfigured. In *Proc.15th ACM SIGPLAN Int.Conf.on Functional Programming*, 2010.

[41] Matthew Naylor and Colin Runciman. The reduceron reconfigured and re-evaluated. *Journal of Functional Programming*, 22(4-5):574–613, 2012.

[42] S. L. Peyton Jones. Grip; a parallel processor for functional languages. *Electronics and Power*, 33(10):633–636, October 1987.

[43] S. L. Peyton Jones and J. Salkild. The spineless tagless g-machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 184–201, New York, NY, USA, 1989. ACM.

[44] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, January 1987.

[45] Jeremy Pope, Jules Saget, and Carl-Johan H. Seger. Cephalopode: A custom processor aimed at functional language execution for iot devices. In *2020 18th ACM-IEEE Int.Conf.Formal Methods and Models for System Design (MEMOCODE)*, 2020.

[46] Alejandro Russo. Functional pearl: Two can keep a secret, if one of them uses haskell. *SIGPLAN Not.*, 2015.

[47] Mark Scheevel. Norma: A graph reduction processor. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming.*

[48] Moses. Schönfinkel. Uber der bausteine der mathematischen logic. *Math. Annalen*, 92(3), 1924.

[49] R. Stewart, E. Belikov, and H.W. Loidl. Graph reduction hardware revisited. In *Trends in Functional Programming 2018*, 2018.

[50] W. R. Stoye, T. J. W. Clarke, and A. C. Norman. Some practical methods for rapid combinator reduction. In *Proc. 1984 ACM Symp.on LISP and Functional Programming*, 1984.

[51] William Stoye. Message-based functional operating systems. *Science of Computer Programming*, 6:291 – 311, 1986.

[52] David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. Safe haskell. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 137–148, New York, NY, USA, 2012. ACM.

[53] D. A. Turner. A new implementation technique for applicative languages. *Software: Practice and Experience*, 9(1), 1979.

[54] D. A. Turner. The semantic elegance of applicative languages. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, FPCA '81, pages 85–92, New York, NY, USA, 1981. ACM.

[55] D. A. Turner. Some history of functional programming languages. In *Trends in Functional Programming*, 2013.

[56] David Turner. Another algorithm for bracket abstraction. *J. Symb. Log.*, 44, 1979.

[57] Philip Wadler. How enterprises use functional languages, and why they don't. 07 2008.