Forward-Temporal Referencing for Instruction Set Architecture Data Flow

Emad Jacob Maroun
ejama@dtu.dk
Technical University of Denmark, Department of Applied Mathematics and Computer Science
Kongens Lyngby, Denmark

Abstract

Most instruction set architectures use registers as a means of data flow. Register-based data flow suffers from having to define a fixed number of accessible registers in the instruction set. This results in inefficiencies: There are often too few registers available, necessitating the addition of code for spilling and reloading them. False dependencies also complicate hardware implementations and increase power consumption by necessitating register renaming.

This paper proposes forward-temporal referencing as a novel means of data flow. Instructions only refer to when their outputs are consumed. This data flow method does not exhibit false dependencies and negates the need for any spilling or reloading. We discuss the hardware implication of forward-temporal referencing and show that it can be incorporated into an efficient and flexible instruction set architecture, Scry. Scry uses five orders of magnitude less encoding space than RISC-V's RV64IMC, enabling the use of only 2-byte instructions. Hand-compiled static code density is comparable to RV64IMC for small functions and improves as functions grow in size.

CCS Concepts

• **Computer systems organization** → **Other architectures**; • **Hardware**;

Keywords

scry, risc-v, instruction set architecture, code density, data flow

1 Introduction

An instruction set architecture (ISA) defines the basic instructions processors understand and execute. Instructions accept inputs, perform an operation, and produce outputs used in subsequent instructions. *Data flow* describes how data moves from one instruction to the next within a program. Managing data flow is one of the most fundamental aspects of the design of ISAs. Combined with other elements, like managing control flow, the ISA must support efficient and performant program execution.

Almost all modern ISAs follow a similar design. They use registers for data flow, where instructions refer to which registers hold their input data and which should hold their outputs. Modern processor implementations use sophisticated techniques to maximize performance while maintaining the semantics of the program. They try to execute multiple instructions in parallel and reorder them so instructions with high latencies do not cause bottlenecks [4]. These efforts are hindered by the decades-old design of contemporary ISAs. The limited number of available registers in the ISA (*architectural registers*) means *false dependencies* arise when more operands are live than there are registers in the ISA. Otherwise-independent instruction must use the same registers, creating a dependency between them that could be avoided if more registers were available. Register renaming allows processors to use more physical registers than defined in the ISA [16]. Renaming stores outputs in different physical registers even when their instructions refer to the same architectural register. This allows the processor to increase parallel execution of instructions. However, it is a complicated process, which consumes a significant portion of a processor's power [10], limits a processor's frequency [13], and limits the number of instructions that can be simultaneously checked for parallelism—reducing its exploitation [15]. Limited architectural registers also necessitate adding additional code to most functions to manage the architectural registers. Register allocation increases code size and reduces its density while making register renaming more cumbersome.

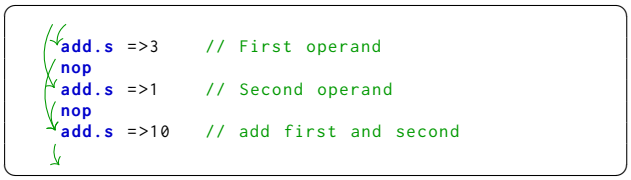
This paper presents *forward-temporal referencing* (FTR) as a novel means of data flow. Data is passed from producers to consumers without explicit registers, instead using temporal references that specify when outputs are consumed. Neither register allocation nor register renaming are needed when using FTR, as false dependencies cannot occur. Instead, the processor can manage data storage and retrieval in the background, using the temporal information to ensure data is ready and consumed when needed.

We first discuss the classification of data flow references. We then present the design of FTR and how it integrates into a novel ISA called *Scry*. We discuss the hardware implications of this method of data flow and how it can affect ISA design. We show that the resulting ISA is *efficient*—uses few code points to encode instructions—and *dense*—uses fewer bytes of instruction data—compared to RISC-V's RV64IMC [19].

2 Reference Types

Data flow in ISAs is achieved using implicit or explicit references. Traditional ISAs like RISC-V or ARM use spatial references to finite resources—registers—to achieve data flow. The reference space is the register space. "`add r1, r2, r3`" has three explicit spatial references "`r1`", "`r2`", and "`r3`". Some ISAs also have implicit references. For example, ARM updates the NZCV condition flags as a side-effect of arithmetic operation [1].

Dataflow architectures also use spatial references [6]. The reference space is the addresses of instructions, with producer instructions referring to the location of the instructions consuming their outputs. When an instruction has all its inputs ready, it will be executed. Dataflow architectures only use forward references; their instructions only refer to who will consume their outputs. Inputs may come from any preceding instruction. In comparison,



Listing 1: Example Scry assembly with reference target highlight.

traditional architectures use both forward and backward references. In "`add r1, r2, r3`", "`r1`" is a forward reference (output) and "`r2`" and "`r3`" are backward references to the location of input values.

Spatial referencing produces false-dependencies when otherwise-independent instructions use the same resource for data flow. In traditional architectures, this results in the need for register renaming, while dataflow architectures instead have problems with operand management and figuring out when instructions are ready for execution [14]. Temporal referencing instead uses time to facilitate data flow. In the STRAIGHT and Clockhands ISAs, instructions refer to when their inputs were produced [7, 9]. The references use instruction execution as a measure of time, where the reference value in the consumer specifies how many instructions have been executed since the producer. This referencing is also exclusively backwards, since outputs may be consumed by any or none of the succeeding instructions.

Backwards-temporal referencing solves the issue of register renaming, since it does not exhibit false-dependencies. However, it does not provide the processor with lifetime awareness. Because outputs may be consumed by any succeeding instruction, they must be stored for the maximum references distance, which must be defined in the ISA. This is resource intensive and inefficient if the maximum distance is far but loses performance if the maximum distance is close, as long-lived values would more often have to be explicitly spilled to memory.

Forward-temporal referencing solves the lifetime issues by making it explicit when a value will be consumed. Processors can then optimize for storing different lifetime classes and immediately drop values when they are no longer needed. We use instruction execution as a measure of time. Explicit forward references indicate how many instructions will be executed before the consumer instruction. Implicit forward references are also used, with the following instruction being the consumer.

3 Forward-Temporal Referencing

In FTR, initial instructions specify when their outputs are consumed. When execution reaches an instruction, the inputs it needs to operate on are already specified by preceding instructions. References are temporal, describing when the operands will be consumed and not the position of the consuming instruction. For example, a reference value of 0 means the next instruction in the instruction stream will consume the output, a 1-reference means the second instruction will consume it, etc. If execution branches at runtime, different instructions will be executed, and the outputs will therefore automatically flow to the executed instruction. Each instruction only refers to when its output(s) will be consumed, with its inputs being implicit. The processor is responsible for managing the lifetime of values so that they arrive at the functional units when needed.

Listing 1 shows an example assembly program using three addition instructions with highlighted reference targets (green arrows). The first and second addition (lines 2 and 4) reference the third addition as the consumer of their outputs. The references use offsets (`=>3` and `=>1`) equal to the number of instructions between each producer and consumer in the instruction stream: 3 and 1, respectively. I.e. `=>3` means whatever instruction is the fourth to be executed after the current one, will be the consumer of the output. In Listing 1, had there been a branch instruction before the third addition, whatever instruction was first in the new instruction stream would have consumed the first two additions' outputs. The order of value production is important. Each instruction's inputs are ordered, with the inputs produced by earlier instructions being first in the order. This could be important for instructions like `sub`, where the second input (i.e., produced later) is subtracted from the first. It is the programmer's/compiler's responsibility to order instructions such that their outputs reach the consuming instruction in the correct order. The dedicated data flow instructions can be leveraged to ensure a correct order.

3.1 Data Flow Instructions

Fine-grained control of values is necessary to ensure FTR can support existing coding patterns. Therefore, dedicated data flow instructions must be present, whose sole purpose is managing values so that they reach their desired consumer.

The authors of [3] show that a vast majority of values are used only once, with only very few being used more than four times. The authors of [17] present similar statistics on the SPEC CPU2000 benchmark suite. There, 70% of values are only used once, while over 90% are used only twice. It is therefore most efficient for instructions to only have one output reference (e.g., `=>3`) and then use dedicated duplication instructions when a value must be consumed multiple times.

The authors of [18] show that 90% of values live for less than 11 clock cycles. The authors of [2] and [5] both show that the active lifetime of values (the time from being written to a register to the last read) is in the low single digits. Lastly, [11] shows that up to 95% of all values will be dead within the 32 instructions following their write. Therefore, for the vast majority of output values, a reference reach of 32 should be enough (requiring a 5-bit encoding field). For values that are longer lived, a dedicated instruction to extend their reach should be included in an ISA. We call this an *echo* instruction. Additionally, easy spilling and reloading to/from the stack is necessary for values whose lifetime is not statically known. This could happen, e.g., for values that are produced before a variably-iterating loop but only consumed after the loop exits. However, [11] shows that this is like rare, as most values do not cross basic-block boundaries.

The last necessary data flow instruction must take multiple input values and then output them through each their output reference. We call this a *splitting echo*. Uses for this instruction could be at loop exits to reorder values to hit the correction instructions following the loop. It is also necessary at the start of functions, where the

function arguments are likely to target only the first instruction, where a splitting echo may reroute them to other instructions.

3.2 Hardware Considerations

FTR puts most of the responsibility for managing values on the processors themselves. One challenge for the hardware is managing in-flight operands in long functions and across function calls. Without limits, the number of inputs to an instruction might be unknown. It would be difficult to implement a processor that can handle any number of instruction inputs. A reasonable limit on the number of inputs can be three or four, as most modern instructions take three inputs or less, leaving some headroom for irregular instructions in the future.

The total number of in-flight operands must also be handled with care. On a function call, the caller's in-flight values must be managed until the call returns, at which point their original targets must still be reached. The callee function might also have many in-flight values when it performs another call. The call stack might, therefore, have an unknown number of total in-flight values needing to be managed. ISAs should not limit the total number of in-flight values, nor the number of in-flight values in a single function. Instead, the processor should store long-lived values in specially-designed caches; which are in turn backed by main memory. This can be done using a combination of a priority queue to managed short-lived values [8] and dedicated caches that also store the remaining distances a value has to be stored in its function scope.

4 The Scry ISA

Scry is a new ISA that aims to support modern processor implementations in their quest for performance. Scry is designed to give the processor as much information about the future as possible to enable it to optimize execution on-the-fly. Output references using FRT provide the processor with the lifetime of values, allowing it to optimize their storage and simplify parallel execution; An instruction can be executed in parallel with any succeeding instruction that it does not refer to. Scry also includes features for control flow that gives the processor early warnings about branching. The details of control flow are out of the scope of this paper.

Scry's secondary goal is to be as efficient of an encoding as possible, as that is one of the few metrics that is mostly affected by the ISA design. Efficient encoding also result in less memory being used and moved, improving performance and efficiency for the whole system. Internal tagging and instruction polymorphism is used to reduce the amount of encoding space needed for its instructions [12], which are exclusively 16-bit. FTR provides further encoding efficiencies by enabling operand-count polymorphism: depending on the number of inputs, instructions may change their semantics. For example, if the `add` instruction is only provided with one input, it will perform an increment operation on the single input. Instructions also most often have fewer outputs than inputs, meaning fewer fields are needed in each instruction, improving encoding efficiency further.

As an example, Listing 2 shows the C language function `strcpy` written in Scry assembly. It starts with a splitting echo instruction that splits the source and destination pointers. Notice how Scry

```
            echo =>dup_dst, =>dup_src
lp_start:
dup_src:    dup =>load, =>inc_src
load:       ld  u8, =>0
            dup =>lp_cond, =>store
lp_cond:    jmp lp_start, lp_end
dup_dst:    dup =>store, =>0
            add.s =>lp_end=>lp_start=>dup_dst
inc_src:    add.s =>lp_end=>lp_start=>dup_src
store:      st
lp_end:     ret return_at
return_at:
```

Listing 2: `strcpy` function in Scry assembly.

assembly can also use labels to calculate output references automatically. Here, `dup_dst` refers to the sixth instruction and `dup_src` to the second. Therefore, the first instruction is equivalent to `echo =>4, =>0`. The main loop then runs between the `lp_start` and `lp_end` labels. It loads (`ld`) a byte, checking it against zero using the conditional jump instruction (`jmp`), and then storing (`st`). The addition instructions (`add.s`) handles incrementing the two pointers and uses a multi-step label reference that automatically calculates reference reach across loop iterations. Lastly, `ret` is used to return from the function.

We have implemented an ISA simulator and an assembler supporting Scry. However, because of Scry's significant fundamental difference from traditional ISAs, implementing a compiler and a processor is a significant challenge we have yet to undertake. Scry has feature parity with RV64IMC and will be compared to it in the following sections.

4.1 Encoding Efficiency

We calculate an instruction's encoding space usage by looking at how many *code points* it uses. A Scry instruction like `nop` (discard all inputs, do nothing else) uses only one code point, as it has no fields. However, the `add` instruction has one output reference fields of five bits. Therefore, it uses $2^5 = 32$ code points. RISC-V's compressed instructions occupy space in the same 32-bit encoding space as the non-compressed instructions, meaning 16 bits go unused for every compressed instruction. We therefore treat the unused bits like a field of 16 bits in code point calculations. For example, the `c.ebreak` has no fields and `c.add` has two register fields [19]. Therefore, they occupy $2^{16} = 65\,536$ and $2^{5+5+16} = 67\,108\,864$ code points, respectively.

We compare the Scry encoding efficiency to that of RV64IMC by summing the amount of encoding space they each use. Cumulatively, Scry uses 18 605 code points. This is 28% of the available 16-bit encoding space. In contrast, RV64IMC uses 2 902 171 650 code points, which is 68% of the 32-bit encoding space.[1]

4.2 Assembly Functions

We evaluate the proposed Scry ISA using hand-written assembly programs. We compare their structure to the same programs written in C and compiled using the GCC RISC-V compiler. We choose short functions from the C standard library with open-source C implementations to represent real-world use cases: `strcpy`, `memcpy`,

[1]The RISC-V analysis is based on the instruction encoding data given in: https://github.com/riscv/riscv-opcodes/commit/b30cec9

Table 1: Instruction composition for evaluated functions. Left values are Scry, right are RISC-V.

Function	Instructions		Bytes		Data		Control		Logues	
`strcpy`	10	**7**	20	**18**	4	**1**	2	2	2	**1**
`memcpy`	14	**9**	28	**22**	6	**1**	3	3	3	**1**
`isxdigit`	13	13	**26**	40	6	**4**	**1**	3	**2**	3
`bsearch`	**33**	47	**66**	106	15	**9**	**5**	7	**7**	25
`cmpu8`	5	**4**	**10**	12	1	**0**	1	1	2	**1**
`find_max`	14	**13**	**28**	36	9	**5**	**2**	5	2	2
`hextol`	36	**31**	**72**	96	21	**8**	**2**	6	2	**1**

`isxdigit`, and `bsearch`. We have also implemented three non-standard functions: `cmpu8` implements a comparison of unsigned bytes for use with `bsearch`; `find_max` does a simple iterative search through an array, returning the largest value; and `hextol` simplifies the C standard library `strtol` function by supporting only hexadecimal and not managing whitespace or returning a value.

We compare the static composition of the selected functions. Table 1 shows, for each function, the total number of instructions, the number of instruction bytes, the number of data-flow instructions, the number of control-flow instructions, and the number of instructions comprising the function prologue or epilogue. Data-flow instructions' only purpose is data movement within the processor. Scry's echos and duplicates (among others) count as data-flow instructions. RISC-V's `mov` instruction also counts.

For the smallest functions, we see that Scry and RISC-V require comparable byte numbers, but Scry requires more instructions. `strcpy` and `cmpu8` differ only by two bytes. `memcpy` needs six more bytes in the Scry version than RISC-V. However, `find_max` needs eight more in the RISC-V version than Scry. Scry's static code density shows promise for the rest of the functions, needing 14, 24, and 40 fewer bytes for `isxdigit`, `hextol`, and `bsearch`, respectively. In `bsearch`'s case, the benefit comes from the prologue and epilogues, where Scry only needs seven instructions while the RISC-V implementation needs 25. The savings come mostly from Scry not needing to save and reload registers. Looking at the amount of data-flow instructions, we see that Scry uses many more than RISC-V, as expected. However, this does not come at the cost of the code density.

The trend these numbers show is that small Scry functions are comparable in density to RISC-V, but the larger and more complicated the function becomes, the more likely Scry is to be denser. Note that these functions are so simple that no register spilling is needed in the RISC-V versions. Larger functions with many live values will affect RISC-V's density adversely. Scry will never need to add instructions for spilling, so it will not be affected as much. Lastly, Scry uses instructions analogous to conditional-moves for small-scale decision-making. Their use has successfully reduced the number of control flow instructions without adversely affecting code density.

5 Conclusion

FTR is a novel method of data flow for ISAs. Instructions refer only to when their outputs are consumed, offloading live value management to the processor. FTR does not exhibit false-dependencies, negating the need for the complicated register renaming. It also improves encoding efficiency by enabling operand-count polymorphism and requiring fewer reference fields than other methods. Scry is a new ISA that uses FTR as its means of data flow. Static analysis of hand-coded functions shows that Scry's static code density is comparable for small functions and superior for larger functions than RISC-V's RV64IMC. Encoding efficiency is also superior, using 5 orders of magnitude fewer encoding points for its instructions. This work is still in the early stages. A compiler must still be implemented to get a better sense of the quality of code that may be produced. FTR also greatly affect the implementations of processors. Extensive investigation must be performed to ascertain whether performant and efficient processors can be built supporting FTR as the means of data flow.

References

[1] Arm Ltd. 2018. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition*. DDI 0406. Arm Ltd. https://developer.arm.com/documentation/ddi0406/latest.

[2] J Adam Butts and Gurindar S Sohi. 2004. Use-based register caching with decoupled indexing. *ACM SIGARCH Computer Architecture News*, 32, 2, 302.

[3] M. Franklin and G.S. Sohi. 1992. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *[1992] Proceedings the 25th Annual International Symposium on Microarchitecture MICRO 25*, 236–245. doi:10.1109/MICRO.1992.697025.

[4] John Hennessy and David Patterson. 2006. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers. ISBN: 1-55880-069-8.

[5] Zhigang Hu and Margaret Martonosi. 2000. Reducing register file power consumption by exploiting value lifetime characteristics. In *Workshop on complexity-effective design (WCED)*. Vol. 1, 1829–1841.

[6] Ali R Hurson and Krishna M Kavi. 2007. Dataflow computers: their history and future. *Wiley Encyclopedia of Computer Science and Engineering*.

[7] Hidetsugu Irie et al. 2018. STRAIGHT: hazardless processor architecture without register renaming. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 121–133.

[8] Lukáš Kohutka, Lukáš Nagy, and Viera Stopjaková. 2018. A novel hardware-accelerated priority queue for real-time systems. In *2018 21st Euromicro conference on digital system design (DSD)*. IEEE, 46–53.

[9] Toru Koizumi, Ryota Shioya, Shu Sugita, Taichi Amano, Yuya Degawa, Junichiro Kadomoto, Hidetsugu Irie, and Shuichi Sakai. 2023. Clockhands: rename-free instruction set architecture for out-of-order processors. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 1–16.

[10] Gurhan Kucuk, Oguz Ergin, Dmitry Ponomarev, and Kanad Ghose. 2003. Energy efficient register renaming. In *International Workshop on Power and Timing Modeling, Optimization and Simulation*. Springer, 219–228.

[11] L.A. Lozano and G.R. Gao. 1995. Exploiting short-lived variables in superscalar processors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*. (Nov. 1995), 292–302. doi:10.1109/MICRO.1995.476839.

[12] Emad Jacob Maroun. 2025. On internally tagged instruction set architectures. *IEEE Computer Architecture Letters*, 1–4. doi:10.1109/LCA.2025.3585621.

[13] Subbarao Palacharla, Norman P Jouppi, and James E Smith. 1997. Complexity-effective superscalar processors. In *Proceedings of the 24th annual international symposium on Computer architecture*, 206–218.

[14] Andrew Petersen, Andrew Putnam, Martha Mercaldi, Andrew Schwerin, Susan Eggers, Steve Swanson, and Mark Oskin. 2006. Reducing control overhead in dataflow architectures. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, 182–191.

[15] Salvador Petit, Rafael Ubal, Julio Sahuquillo, and Pedro López. 2013. Efficient register renaming and recovery for high-performance processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22, 7, 1506–1514.

[16] Dezso Sima. 2000. The design space of register renaming techniques. *IEEE micro*, 20, 5, 70–83.

[17] Francis Tseng and Yale N. Patt. 2008. Achieving out-of-order performance with almost in-order complexity. In *2008 International Symposium on Computer Architecture*, 3–12. doi:10.1109/ISCA.2008.23.

[18] Brian Van Essen, Robin Panda, Aaron Wood, Carl Ebeling, and Scott Hauck. 2010. Managing short-lived and long-lived values in coarse-grained reconfigurable arrays. In *2010 International Conference on Field Programmable Logic and Applications*. IEEE, 380–387.

[19] Andrew Waterman, Krste Asanovic, et al. 2019. The RISC-V instruction set manual volume I: unprivileged ISA. *RISC-V Foundation*, 20191213, 1–4.