RISC-V Dataflow Extension

Martin Cowley and Lina Sawalha
{martin.j.cowley,lina.sawalha}@wmich.edu
Western Michigan University
Kalamazoo, Michigan, USA

Abstract

Dataflow architecture is an alternative to the Von Neumann (control flow) architecture that can improve performance and lower energy consumption. Existing dataflow architectures are either explicit or hybrid. While explicit dataflow architectures provide higher performance for irregular workloads, control flow architectures provide effective control speculation and precise interrupts/exceptions. Hybrid dataflow/Von Neumann architectures combine both dataflow architecture models with control flow architectures.

Most of the existing hybrid architectures integrate elements of dataflow in the Von Neumann computing model. In this work, we designed an Instruction Set Architecture (ISA) extension for the RISC-V architecture that is capable of executing explicit dataflow instructions. We also implemented a heterogeneous dataflow/Von Neumann CPU model in the Gem5 simulator capable of running both Von Neumann and dataflow instructions in a single executable. We ran few microbenchmarks; our results show up to 7.5% improvement in performance, demonstrating the potential for the dataflow ISA extension and the heterogeneous dataflow/Von Neumann architecture.

CCS Concepts: **• Computer systems organization → Data flow architectures**; **• Theory of computation → Timed and hybrid models**.

Keywords: RISC-V extension, Dataflow architecture

ACM Reference Format:
Martin Cowley and Lina Sawalha. 2018. RISC-V Dataflow Extension. In *Proceedings of CARRV '21: Fifth Workshop on Computer Architecture Research with RISC-V (CARRV '21).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

1 Introduction

Traditional Von Neumann architectures rely on the control flow model of computation. In this model, the processor uses a program counter (PC) to sequentially fetch instructions from memory. However, even with the out-of-order execution model, which integrates some dataflow aspects, parallelism can still be limited by the window size. In addition, Von Neumann architectures are not very efficient for workloads with irregular data and control behaviors.

Dataflow is an alternative model of computation. Instructions are fetched and executed based solely on the availability of data. Existing work shows that dataflow architectures have the potential to improve performance and reduce energy consumption [2–4, 6, 8, 9]. Explicit dataflow architectures have been effective for irregular workloads, but they struggle with control speculation, debugging, and implementing dynamic data structures. Combining explicit dataflow and control flow computing models in a heterogeneous processor architecture allows for exploiting the benefits of both.

In this paper, we propose a dataflow (DF) extension to the RISC-V Instruction Set Architecture (ISA). To test the performance of the extension we created a dataflow CPU model and integrated it with the Gem5 simulator. We also designed a heterogeneous dataflow/Von Neumann (DF/VN) architecture.

The rest of the paper is organized as follows. Section 2 mentions background information about DF and hybrid DF/VN models. Section 3 describes our DF extension to the RISC-V ISA. Section 4 shows our heterogeneous architecture, and section 6 describes the experimental setup and shows the results. Finally, section 7 concludes the paper and describes future work.

2 Background

Dataflow architectures have been studied for decades. Because processors are susceptible to instructions with long latency, computer architects have been searching for ways to increase parallelism. For example, a simple in-order processor must block during a cache miss, causing the CPU to be idle for long periods of time. This is inefficient, since independent instructions that are ready to be executed has to wait in the pipeline. Researchers have turned towards the dataflow model of computation to help alleviate the problem.

In the dataflow model, an algorithm is not represented as a sequence of instructions. Rather, dataflow programs are written as a graph. Each node of the graph is an instruction and arcs between nodes represent true data dependencies between instructions. Instead of referencing source and destination registers, dataflow instructions contain pointers to destination instructions. In a dataflow program, data is sent from one instruction to the next. Data drives instruction scheduling and program order is ignored. Figure 1 shows a simple arithmetic expression represented as a dataflow graph.

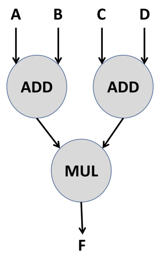

Figure 1. Dataflow Graph implementing the expression: f = (a+b)*(c+d)

2.1 Tagged-Token Architectures

Dataflow architectures send instruction operands directly to each instruction instead of loading them from a register file. There are different ways to implement this in hardware. Tagged-token machines represent each operand as a data token [1, 6]. Each piece of data is stored in a token, which contains the data and an identification tag. A cache is used to store operand tokens until the corresponding instruction is ready to be executed. This cache is also used as the instruction scheduling mechanism. An instruction becomes ready when both operands have reached the cache. When this event is detected, the two operand tokens are dispatched along with the instruction and executed.

One option for tag matching is to use a fully associative memory to store the tokens. The instruction address is used as the tag/key to access the cache. All operands of an instruction will have identical tags and will be mapped to the same location. A hash function has also been used to implement this feature [1, 6]. The Monsoon architecture uses similar tag-matching hardware, but simplified the matching system by replacing the associative memory with a simple effective address calculation [8]. These architectures use the instruction address to detect when an instruction is ready to be executed; when two operands of an instruction arrive, both operand tokens will be matched to the same location in memory, and the instruction is ready to be executed.

2.2 Hybrid DF/VN Models

Modern out-of-order processing cores borrow from the dataflow model to exploit Instruction Level Parallelism (ILP). They use sophisticated hardware to detect instructions that are ready and execute them out of program order. The main limitation is that these cores are fundamentally control flow and must maintain program order by committing instructions in-order. This complicates the hardware–increasing cost, latency, and energy consumption–and limits ILP because of the limited window of instructions that can be executed out-of-order.

Other hybrid dataflow/Von Neumann (DF/VN) architectures combines the control flow and dataflow aspects differently. Some examples include TRIPS [3], WaveScalar [9] and DySER [5]. While these architectures relax the restrictions of VN architectures, they still limit the amount of parallelism that can be achieved. On the other hand, explicit dataflow architectures can schedule instructions from anywhere in the program as long as their operands are ready. This allows for a much wider window of instructions to choose from.

2.3 Heterogeneous DF/VN Architectures

Heterogeneous DF/VN architectures combine different DF and VN cores in one processor, for example the SEED architecture [7]. The SEED architecture consists of two heterogeneous cores: a standard VN based core and a DF engine. Nowatzki et al. analyzed different benchmarks to study the potential improvements for a heterogeneous architecture capable of switching between the DF and VN cores within a single application. They used a high-level modeling technique to estimate performance instead of a cycle-level simulation. They did not take into consideration the switching cost between the DF and the OoO cores and assumed a different instruction set for the DF engine.

In this paper, we propose a DF extension of the RISC-V ISA. Our work considers a heterogeneous DF core and an OoO core that share the same cache. Our work considers the switching cost between the DF] and the OoO cores. We used a cycle-level architectural simulator, *gem5* to simulate our DF extension and heterogeneous architecture. We also used the llvm compiler infrastructure and implemented a simple compiler that translates the llvm intermediate representation (IR) to the new dataflow RISC-V sub-ISA instructions.

3 The RISC-V Dataflow Extension

Dataflow (DF) instructions require a completely different structure from standard control flow instructions. To simplify the hardware, each instruction has a maximum of two operands and up to three output arcs, as shown in Figure 2. The result of an instruction is sent along its arcs to each dependent instruction. Figure 3 compares the bit fields between a normal RISC-V instruction and a dataflow instruction. The instructions are encoded with opcode and function bits to specify the operation, similar to regular RISC-V instructions. However, instead of using the remaining bits to specify the source operands, the dataflow instructions encode pointers to dependent instructions: Destination0, Destination1, and Destination2– representing the arcs in a DF graph. The remaining bits D2, D1, and D0 are used to distinguish between left or right operands.

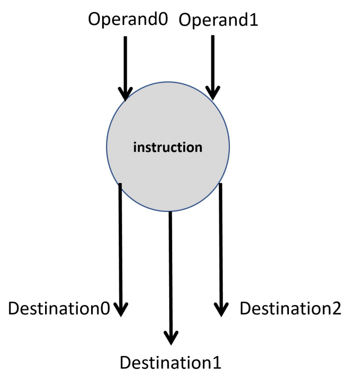

Figure 2. Instruction Node

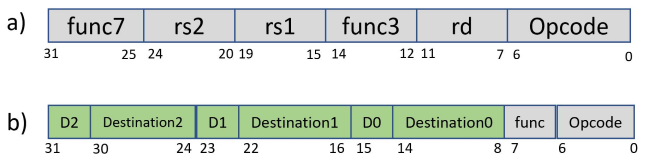

Figure 3. RISC-V Instruction Bitfields: (a) regular RISC-V instruction (R-type), (b) RISC-V DF extension instruction

One benefit of a DF architecture is that it moves data dependencies to the ISA level. Instructions have explicit pointers to their dependent instructions lowering the complexity of the hardware needed to keep track of data dependencies and to detect ready instructions. To simplify the hardware, each instruction has at most two operands and up to three instruction pointers. The instruction pointers are encoded as an offset from the current instruction. When an instruction executes, the data is sent to all dependent instructions. The effective address of the dependencies are calculated using the following equation:

$$EA = currInst.Addr + destinationOffset$$

Our DF ISA extension consists of DF instructions in four different main categories: arithmetic and logical instructions, memory instructions, control instructions and DF/VN communication instructions.

Arithmetic and Logical Instructions. Table 1 shows the arithmetic and logical instructions that the dataflow extension currently supports. It reflects the common operations found in most ISAs: addition, subtraction, multiplication, bitwise operations, and compare instructions.

Memory instructions. Table 2 lists all memory instructions in the DF extension. The dataflow execution model accesses memory through shared caches with the OoO model.

Even though true data dependencies are explicit in the dataflow extension, memory aliasing is still a problem. We take a conservative approach and force memory instructions

Table 1. Arithmetic and Logic Instructions

Name	Description
df_add	addition
df_sub	subtraction
df_sl	shift left
df_and	bitwise AND
df_not	bitwise NOT
df_mul	multiplication
df_cmpLT	Compare Less Than
df_cmpEQ	Compare Equal
df_cmpGT	Compare Greater Than

Table 2. Memory Instructions

Name	Description
df_lw	load 64-bit
df_sw	store 64-bit
df_sd	store 32-bit
df_ld	load 32-bit
df_lb	load single byte
df_sb	store single byte

to execute in-order any time an aliasing problem can occur. This memory ordering is done by the compiler: the compiler inserts additional data dependencies between memory instructions to force them to execute in-order. As such, our architecture does not introduce new memory consistency problems.

Control Instructions. Control instructions (Table 3) are used to implement conditional branches and looping. Dataflow branch instructions are similar to control flow branches in that they allow the program to execute conditional code (e.g. if-else statements). Control flow branches modify the program counter to dictate which part of the algorithm should be executed. However, the DF extension converts control dependencies to data dependencies, which controls the flow of data between different paths in the dataflow graph.

Figure 4 shows the structure of a branch instruction. There are two operands: a piece of data to be sent, and a boolean. The output of the branch has two arcs, but only one path is taken per branch. The boolean input determines which path should be taken. If the condition is 0, the false path will be taken, otherwise the true path will be taken.

Branch instructions can be used to implement loops, and the compiler unrolls loops four times. However, loops introduce a problem with re-entrant code: how do we execute the same instruction multiple times in a DF architecture? A unique number is given to each loop iteration in the form of a context tag. The context tag isolates each loop iteration to prevent any conflicts. The *df_loop_br* instruction behaves

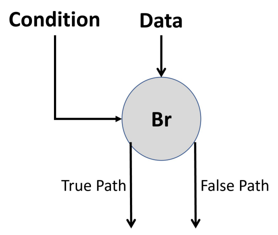

Figure 4. Dataflow Branch Instruction

Table 3. Control Instructions

Name	Description
df_br	branch instruction
df_loop_br	branch+modify context
df_modContext	instruction modifies the context tag

Table 4. Communication Instructions

Name	Description
df_switch	toggle between Von Neumann and dataflow mode
df_tok	generates a dataflow token
df_mov	loads token data into a register

like the *df_br* branch instruction except it also increments the context. This is used to handle simple for/while loops. If more complex loop structures need to be implemented, the *df_modContext* allows the compiler to directly change the context.

VN/DF Communication. Because DF instructions deviate from the VN model significantly, some glue instructions are needed to integrate the DF sub-ISA and allow for fast switching between the DF and VN models. The *df_switch* instruction toggles the execution mode between the VN and DF cores. This is needed because executing DF instructions requires a complete shift in the way the processor fetches, schedules, and executes instructions. The *df_tok* and *df_mov* instructions allow the VN and DF cores to communicate. The *df_tok* instruction is used to take data from a register and create a DF token. The *df_mov* instruction performs the opposite action; a data token is moved into the register file.

4 The DF Extension's Microarchitecture

Our DF microarchitecture is based on earlier work of tagged-token machines [1, 6] with modifications to reduce energy consumption and also support mainstream DF execution. Tagged-token machines process data in the form of tokens (Figure 5). Each token has two components: a piece of data and a tag. The tag includes an instruction pointer (to the instruction that uses the operand) and a context value that is used for looping.

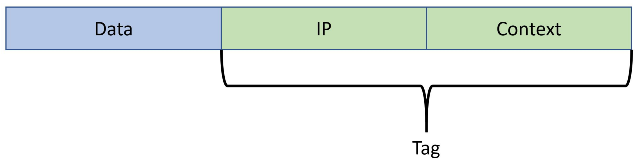

Figure 5. Dataflow Token

The tag is used to match operands that belong to the same instruction; if an instruction has two operands then both operands will contain the same tag. The DF core knows that an instruction is ready to execute when it detects two tokens with the same tag. The instruction and the two operands are then sent to the execute stage. Single operand instructions do not have to wait and can be executed as soon as the first token is generated.

The DF core consists of a simple pipeline. All dependencies among instructions are determined statically by the compiler; however, instructions are being dispatch to the functional units dynamically. The DF core can support predication and can be used for limited regions or larger regions of code, with restrictions on the number of tokens that can be handled as once.

4.1 Pipeline

The DF pipeline is a circular pipeline; tokens that are generated in the execute stage are routed back to the token queue to be processed. Figure 6 shows the four main steps of the pipeline: fetch, decode, match, and execute. Each step is more than one stage and takes multiple cycles. The length of the DF pipeline is 12. Both the DF and the OoO piplines run under the same clock frequency.

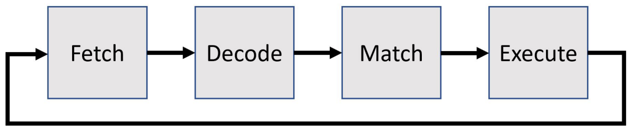

Figure 6. Pipeline

Fetch. The fetch unit fetches instructions from memory. The only deviation from VN cores is that there is no PC. Instead, the fetch address comes from the input token sent from the previous stage or the OoO core.

Decode. This stage decodes instructions, but does not have to fetch operands from a register file. Instead, the operands are contained in the tokens.

Match Unit. The match unit is used to detect which instructions are available for execution. This module is significantly different from any other in the VN architecture. Instead of a register file, the match unit has a small cache of memory for storing data tokens, called token cache. It is a 450-entry content addressable memory with the token's tag as the key. When both operand tokens of an instruction have been generated and have reached the match stage, the instruction is "fired" and sent to the next stage to be executed. Here are two scenarios that result in an instruction being executed:

- Single-operand instruction: it bypasses the match unit and is sent directly to the execute stage.
- Dual-operand instruction: the first operand token is stored in the cache when it arrives. When the second operand token is generated, a match is detected because both tokens have the same tag and are mapped to the same spot in the cache. Then the first token is removed from the cache and both operands are sent to the next stage for execution.

This stage also contains a simple loop predictor. This is used to speed up execution of simple loop structures. However, more complex control speculation is difficult due to the difficulty of recovering from branch mispredictions. Because instructions order is abandoned in dataflow cores, it is difficult to flush all speculative instructions. After a branch misprediction, our dataflow core flushes the entire token cache and returns control back to the OoO core.

Execute. Once a match has been determined, the instruction is dispatched to one of the functional units. When the instruction is executed, instead of writing back to a register file, up to three tokens are generated and sent back to the token queue.

5 The Heterogeneous DF/VN Architecture

Our heterogeneous DF/VN model consists of two cores, an out-of-order (OoO) core and a DF core. A switch instruction will change the mode of operation between the DF and the OoO cores. One advantage of our ISA extension is that the heterogeneous architecture is capable of switching execution models within a single application with reduced overhead compared to switching to a totally different ISA. It also imposes lower overhead on compilation and loading.

The two cores share the same cache, which allows the cores to communicate large amounts of data (arrays) without communication overhead. Additionally, the *df_tok* and *df_mov* instructions allow the two cores to directly send data back and forth (see previous section). Figure 7 shows a block diagram of our DF/VN heterogeneous architecture. Table 5 summarizes the main microarchitecture configurations of both the DF and the OoO pipelines.

6 Experimental Setup and Results

To evaluate the potential for our DF extension, a DF CPU model was added to the gem5 computer architecture simulator [2]. A heterogeneous processor architecture contains

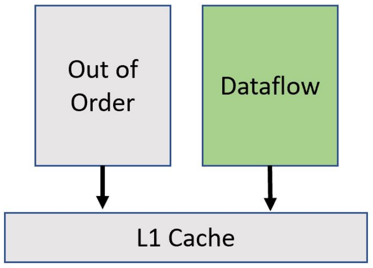

Figure 7. Hybrid Model

Table 5. OoO and DF Microarchitecture Configurations

Name	OoO	DF
Pipeline stages	15	12
fetch Width	8	8
decode Width	6	6
Rename Width	6	-
Match Width	-	8
commit Width	8	8
Branch predictor	loop predictor	loop predictor
Reservation station	60 entries	-
Reorder buffer	220 entries	-
Match unit	-	450 entries
L1 ICache Size	16 KB	
L1 DCache Size	64 KB	
L2 Size	4 MB	

the DF pipeline discussed in the previous section as well as gem5's existing out-of-order core, called O3.

We have implemented a simple compiler written as an LLVM backend that is capable of compiling C/C++ code into a dataflow graph. More work is required to make the compiler more efficient and optimized. As such, we optimized certain parts of the code manually. Because of the manual modifications effort, this work considers only microbenchmarks to study the performance of the DF extension for common programming features (loops, arrays, arithmetic, etc.). We wrote three microbenchmarks: sum array, matrix multiplication and indirect sum.

Sum Array. The first microbenchmark is a simple sum array program. A single for loop is used to loop through and add up all elements of an array. This microbenchmark was chosen to measure the DF core's performance of two features that are ubiquitous to programming: for loops and arrays.

Matrix Multiplication. Multiplies two NxN matrices. Matrix multiplication has regular access patterns, but has a more complex memory and control structures. Only the inner most loop was compiled to dataflow and the rest of the code

Algorithm 1: Sum Array

```
sum = 0;
for i=0; i<N; i++ do
    sum += array[i];
end
```

was run on the OoO core. This also tests the performance of a program that frequently switches between dataflow and Von Neumann execution.

Algorithm 2: Matrix Multiplication

```
sum = 0;
for k=0; k<N; k++ do
    for j=0; j<N; j++ do
        for i=0; i<N; i++ do
            result[k][j] += A[k][i] * B[i][j];
        end
    end
end
```

Indirect Sum. The next program represents a class of algorithms called “irregular applications". These are applications that have irregular data or control behaviors that are difficult for traditional VN processors to handle. The *i*ndirect sum microbenchmark finds the sum of an array but the access to the array is not sequential as shown below. *idx* is an array of random integers, which are used to index the array; the array elements are accessed in a random order. This irregular memory access pattern is unpredictable and performs poorly with the cache, causing large delays. The DF model can improve performance by increasing ILP.

Algorithm 3: Indirect Sum

```
sum = 0;
for i=0; i<N; i++ do
    index = idx[i];
    sum += array[index];
end
```

We compiled two versions of each microbenchmark: one using only the standard RISC-V instruction set for the OoO core and one for the heterogeneous architecture including the DF extension. Both OoO and DF cores run at the same frequency. Table 6 shows the speedup of each microbenchmark relative to the OoO core. The *S*um Array benchmark shows a 7.5% improvement of performance using the DF extension due to mainly lower cache misses and lower number of instructions. The *M*atrix Multiplication results in a slowdown using the DF core due to the overhead cost of the frequent switching between the VN and DF cores. Finally, the DF extension shows a 3.2% improvement in performance over the OoO core for the *I*rregular Sum benchmark, executing more instructions in parallel.

Table 6. Percent Improvement of dataflow and Von Neumann CPUs

Benchmark	% Improvement
Sum Array	7.48%
Matrix Multiplication	-2.36%
Indirect Sum	3.21%

7 Conclusion and Future Work

In this work, we designed a new extension for the RISC-V Instruction Set Architecture (ISA) that executes dataflow instructions. We added the dataflow sub-ISA to the RISC-V ISA and simulated it using the gem5 simulator. Our heterogeneous dataflow/Von Neumann architecture runs applications using both the dataflow and out-of-order execution models, considering the communication cost between both pipelines. We used three microbenchmarks to simulate the hybrid architecture. Our results show a performance improvement up to 7.5% for the dataflow core over the OoO core.

In the future, we aim to improve the compiler to generate an optimized dataflow code. We will also run many different benchmarks to find the performance benefits for different types of applications and application domains. In addition, we will estimate the energy consumption of the DF core and the heterogeneous architecture. Finally, we will consider adding more instructions to the dataflow ISA extension and optimize the DF hardware further.

8 Acknowledgement

We thank the reviewers for their valuable feedback. This work was supported in part by the Faculty Research And Creative Activities Award (FRACAA) W2016-037, and the College of Engineering and Applied Sciences at Western Michigan University.

References

[1] Arvind and R. Nikhil. 1990. Executing a program on the mit tagged-token dataflow architecture. *IEEE Trans. Comput.* 39 (1990), 300–318. https://doi.org/10.1109/ISCA.1990.134511

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood and. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (May 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[3] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, and the TRIPS Team. 2004. Scaling to the

End of Silicon with EDGE Architectures. *IEEE Computer* 37, 7 (2004), 44–55. https://doi.org/10.1109/MC.2004.65

[4] Jack B. Dennis and David P. Misunas. 1975. A Preliminary architecture for a Basic Data-Flow Processor. *ISCA '75 Proceedings of the 2nd annual symposium on Computer architecture* 3, 4 (Jan. 1975), 126–132. https://doi.org/10.1145/642089.642111

[5] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. 2011. Dynamically specialized datapaths for energy efficient computing. *HPCA* (Feb. 2011), 300–318. https://doi.org/10.1109/HPCA.2011.5749755

[6] J. R Gurd, C. C Kirkham, and I. Watson. 1985. The Manchester prototype dataflow computer. *Commun. ACM* 28, 1 (Jan. 1985), 34–52. https://doi.org/10.1145/2465.2468

[7] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. 2015. Exploring the potential of heterogeneous von neumann/dataflow execution models. *ISCA '15 Proceedings of the 42nd Annual International Symposium on Computer Architecture* 43, 3 (2015), 298–310. https://doi.org/10.1145/2872887.2750380

[8] G.M. Papadopoulos and D.E. Culler. 1990. Monsoon: an explicit token-store architecture. *Proceedings. The 17th Annual International Symposium on Computer Architecture* (May 1990), 82–91. https://doi.org/10.1109/ISCA.1990.134511

[9] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J. Eggers. 2007. The WaveScalar architecture. *ACM Transactions on Computer Systems (TOCS)* 25, 2 (May 2007).