# Implementation Trade-offs in Using a Restricted Data Flow Architecture in a High Performance RISC Microprocessor

**M. Simone, A. Essen, A. Ike*, A. Krishnamoorthy, T. Maruyama*, N. Patkar, M. Ramaswami, M. Shebanow**, V. Thirumalaiswamy, D. Tovey**

**HaL Computer Systems, Inc.**

**1315 Dell Avenue, Campbell, CA 95008**

*** Fujitsu Limited, Kawasaki, Japan**

**** Currently at Cyrix Corp., Richardson, Texas**

## Abstract

The implementation of a superscalar, speculative execution SPARC-V9 microprocessor incorporating Restricted Data Flow principles required many design trade-offs. Consideration was given to both performance and cost. Performance is largely a function of cycle time and instructions executed per cycle while cost is primarily a function of die area. Here we describe our Restricted Data Flow implementation and the means with which we arrived at its configuration. Future semiconductor technology advances will allow these trade-offs to be relaxed and higher performance Restricted Data Flow machines to be built.

## 1.0. Introduction

We have implemented a superscalar 64-bit RISC microprocessor based on the SPARC V9 architecture[1] that executes instructions both out of order and speculatively[2]. Many theoretical concepts were incorporated into the design. This paper describes the various techniques we used and how we arrived at our design choices. In a way, this paper explores the feasibility of applying theoretical concepts in the implementation of a high performance Restricted Data Flow machine.

The goal of the project was to produce a manufacturable, high performance superscalar microprocessor. This paper describes three performance trade-offs. First, we resized the register files. Second, we optimized the number of entries in our reservation stations. Finally, we modified the execution selection mechanism to meet our cycle time goal. These three trade-offs reduced the number of instructions that our machine executes per cycle (IPC). But the decrease in cycle time and reduced die area compensated for the losses in IPC.

This paper is organized into five more sections. The second section discusses the background of Restricted Data Flow (RDF) architectures. It provides a definition of RDF, an explanation of our parameters, and a rationale for the implementation of this type of machine. The third section gives an overview of the Data Flow Unit. It describes the register files and the individual reservation stations. The fourth section describes the method we used for performance evaluation. The fifth section discusses three large trade-offs made during implementation. The sixth section presents some concluding remarks and shows one way in which we can improve upon this implementation in future projects.

## 2.0. Restricted Data Flow Background

The advantages of data flow architectures over conventional Von Neumann architectures have been their ability to exploit fine grain instruction level parallelism [3]. It was found that in excess of 17 instructions could be executed per cycle removing all constraints except the program semantics [4]. By taking advantage of some of this parallelism, high performance machines can be built.

Obviously, removing all constraints is currently impractical. Even removing a large number of constraints is extremely difficult to implement in a single microprocessor. Not only is the number of execution units too great, but the issue and completion mechanisms become too difficult to manage. Also, usage of the execution units will not be efficient.

We used a Restricted Data Flow paradigm for our implementation in which the restriction is on the issue rate with the execution latencies specified [4]. The idea is to dynamically schedule a window of instructions, moving this window through the entire program [5].

## 2.1. HaL's Restricted Data Flow definition

RDF is defined "by three parameters: window size, issue rate and instruction class latencies" [4]. A window size of 64 and an issue rate of 4 instructions per cycle were chosen. The latencies of our instructions can be found below in Table 1. There are 4 fixed point instruction units and one pipelined floating point multiplier/adder unit. The integer multiplier/divider and the floating point divider units are non-blocking and thus are allowed to execute in parallel with other execution units. The Load/Store Unit sends commands to two interleaved Data Caches.

**TABLE 1.** Instruction Latencies.

| Instruction | Latency |
|---|---|
| ALU | 1 cycle |
| Integer Multiply (32 bits) | 3 cycles |
| Integer Multiply (64 bits) | 5 cycles |
| Integer Divide | 2-37 cycles |
| | |
| Load | 3 cycles |
| Store | 5 cycles |
| | |
| Floating Point | 4 cycles |
| Floating Point Moves | 1 cycle |
| Floating Point Divide (single) | 5 cycles |
| Floating Point Divide (double) | 9 cycles |

## 2.2. Why use a Restricted Data Flow algorithm?

A restricted data flow algorithm is used to take advantage of the natural parallelism that is present in any program. It reduces pipeline stalls by issuing instructions in order, executing them out of order, and finally committing them in order. The following code segment is an example. This loop repeats until %g1 is equal to zero.

```
loop_start:   Sub %g1, 0x1 --> %g1
              Load [%g2] --> %f0
              Load [%g2 + 0x8] --> %f1
              Fmadd %f0, %f1, %f2 --> %f3
              Store %f3 --> [%g3]
              Add %g2, 0x10 --> %g2
              Add %g3, 0x8 --> %g3
              Brnz %g1 --> loop_start
```

In a sequential superscalar machine, the pipeline would stall for four cycles while the floating point multiply/add (Fmadd) instruction executes. The store following the Fmadd instruction needs to wait for %f3 to be generated before it can issue. Meanwhile, the instructions after the store cannot be issued, even if these instructions are not dependent on the results of the Fmadd instruction.

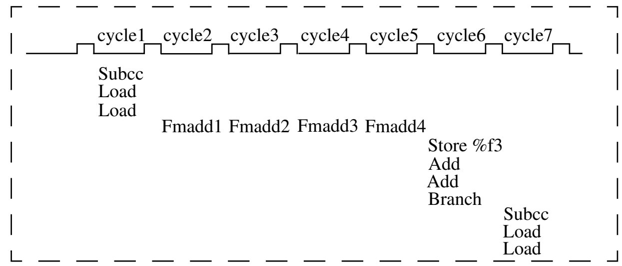


**Figure 1.** Sequential Pipeline.

In a Restricted Data Flow machine, the pipeline does not stall while the Fmadd instruction executes. With an issue window of four, up to four instructions can be issued each cycle. These instructions would execute as soon as their source dependencies are met. This would allow the instructions after the store to execute in parallel with the Fmadd instruction. This would further speed up execution by allowing the next few loops to start executing before the first loop has completed.

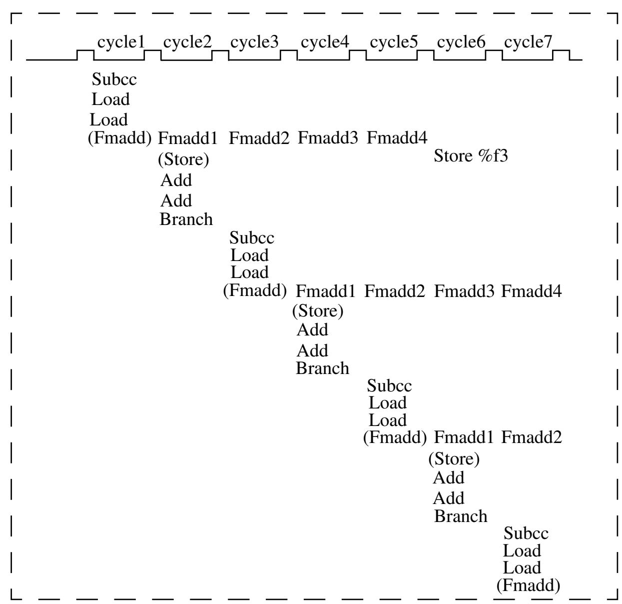


**Figure 2.** Restricted Data Flow Pipeline.

## 3.0. Data Flow Unit Overview

The Data Flow Unit in our microprocessor contains two register files and four reservation stations. The SPARC V9 architecture supports both fixed point and floating point data types. Instructions issue in order, execute out of order, and commit in order. As soon as an instruction has executed, it is removed from the reservation station. New instructions are then able to be issued to the newly freed slot in the reservation station, allowing the window size of 64 instructions to be greater than the total number of slots in the reservation stations.

On issue, instruction packets are dispatched from the Issue Unit to one of the reservation stations, depending on the type of instruction. If an instruction's operands are available in the register file, they are read during the issue cycle. If the operands have not yet been written into the register file, they are captured in the cycle that they are generated. Each reservation station monitors the execution units' results, looking for operands that it needs among the instructions that are currently executing. When an instruction packet's operand is generated, the data is captured in the reservation station. Once all of an instruction's operands are captured, the instruction is ready to be sent to an execution unit. Figure 3 is a block diagram of the Data Flow Unit.

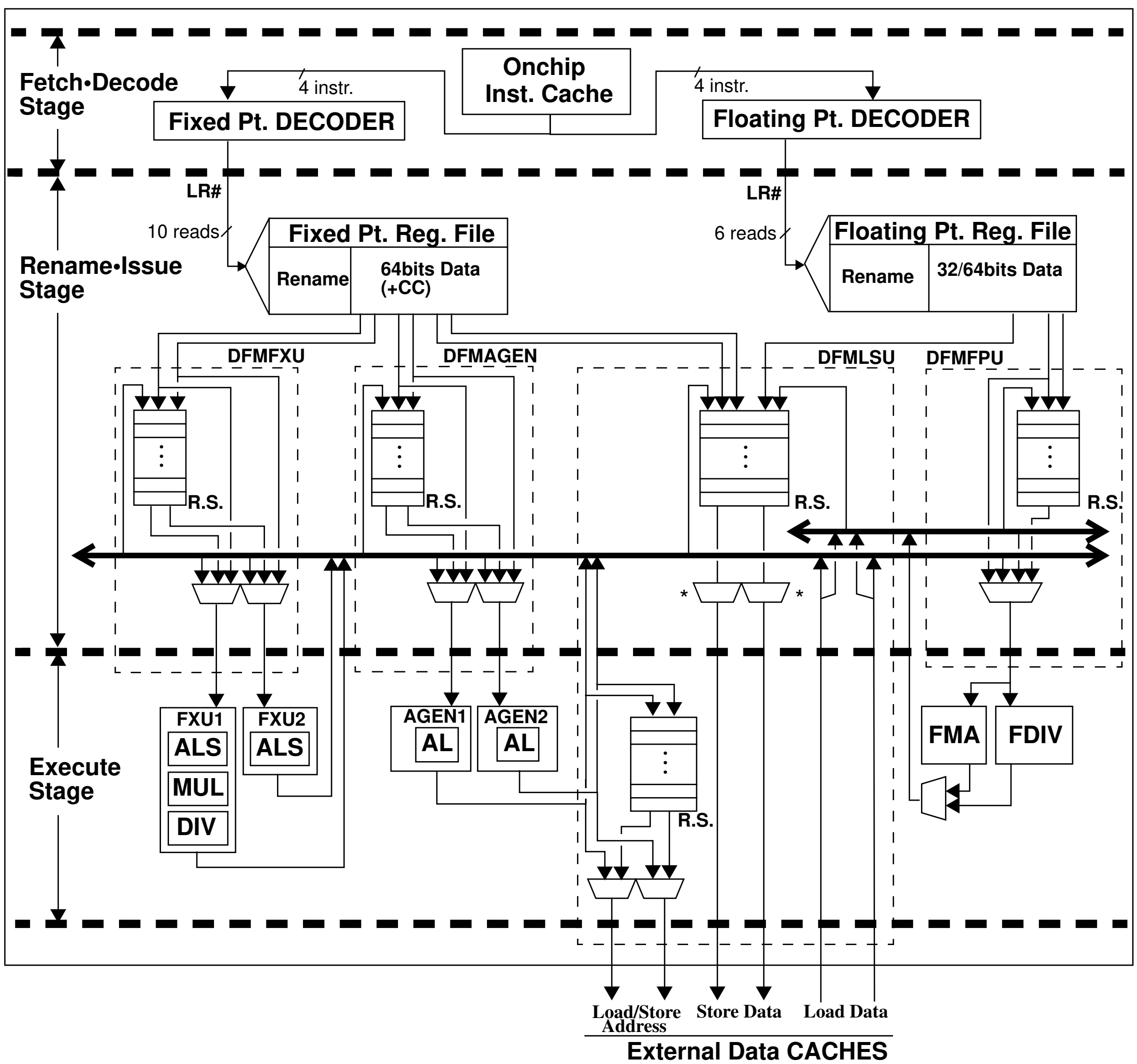


**Figure 3.** Block Diagram of the Data Flow Unit.

### 3.1. Register Files

In order to be able to issue multiple instructions that write the same destination register, logical registers are renamed to physical registers during the issue cycle. This involves assigning an existing free entry, or physical register, in the register file to the logical destination of an instruction [6]. All future instructions which use this logical destination as a source will use the contents of the new renamed physical register. Thus, register dependencies are removed and instructions can execute correctly out of order.

The microprocessor has two register files: a fixed point register file (FXRF) and a floating point register file (FPRF). The FXRF contains 116 64-bit registers. To support the issue of two fixed point instructions (each requires two source registers) and two store instructions (each requires three source registers) in the same issue window, the FXRF reads 10 different source registers at once [7]. Since there are four integer result busses (the two Fixed Point instructions that execute in the AGEN Reservation Station share result busses with the load data that is returning from the two Caches), the FXRF must be able to write four different entries per cycle.

There are four register windows in our SPARC-V9 implementation. This means that of the 116 registers, 80 are always bound to logical registers:

**4*8 in/out registers**
**4*8 local registers**
**8 global registers**
**8 alternate global registers**

Of these 80 registers, two are always set to zero and are handled outside of the register file. This gives the FXRF 38 free fixed point registers to be renamed.

The FPRF contains 56 32-bit registers. Due to architectural compatibility, there are two placements of this unit, used for a combination of both single and double precision registers. Floating point instructions require at most three source registers. Two floating point instructions can be issued per cycle. Due to die area constraints, only one floating point store can be issued per cycle. Floating point stores can be issued with one other Floating point instruction in the same instruction window. Floating point store instructions require one floating point source for the data to be written. Therefore, the FPRF supports the reading of six different source registers at once. There are three result busses, two for floating point loads and one for the floating point multiply/add unit, so the FPRF must be able to write three different entries per cycle.

Of the 112 32-bit registers in the FPRF, SPARC V9 binds 32 registers to either single or double precision registers. An additional 32 are bound to double precision registers. The remaining 48 32-bit registers are used as free registers which can be renamed to any combination of single or double precision registers. Double precision registers result in the renaming of two 32-bit physical registers.

### 3.2. Fixed Point Reservation Station

The Fixed Point Data Flow Machine (DFMFXU) contains an eight entry reservation station. Up to two instructions can be issued to this reservation station each cycle. It contains three *Register Caches* for the instruction's operands: source1 register, source2 register, and condition code register. *Register Caches* are used to capture and store operands' data as it is generated. The DFMFXU also contains an opcode and a physical destination register array. The DFMFXU can select two instructions each cycle to send to the two fixed point execution units that are attached to it.

The first fixed point unit, FX1, contains an Arithmetic/ Logic/ Shift (ALS) unit and integer multiply (MUL) and divide (DIV) units. Multiply and divide instructions are multi-cycle instructions that are allowed to execute in parallel with other instructions. One cycle before multiply or divide instructions are ready to complete, a "done next" signal is sent to the selection logic in the DFMFXU which inhibits an instruction from being sent to this execution unit in the next cycle. This allows the multiply and divide units to share the FX1 result bus.

The second fixed point unit, FX2, contains an Arithmetic/ Logic/Shift unit. This execution unit is also responsible for computing the targets for some control transfer instructions.

### 3.3. Address Generation/Fixed Point Reservation Station

The Address Generation Data Flow Machine (DFMAGEN) is very similar to the DFMFXU. It has an eight entry reservation station that can accept two instructions issued each cycle. It generates the address of load/store instructions. The DFMAGEN also accepts the most common fixed point instructions. It can select two instructions each cycle to send to the two fixed point units that are attached to it.

The two fixed point units associated with the DFMAGEN, AGEN1 and AGEN2, have an extra connection to the address *Register Cache* in the Load/Store Data Flow Machine. When addresses are calculated, they are written into this *Register Cache*. The addresses can be sent directly to the Data Caches if no other Load/Store instructions are executing. The fixed point results computed in these units are muxed with the results of loads from the Data Caches. Due to this, in the cycle that data is returning from a Data

Cache, a fixed point instruction is not allowed to execute in that unit.

The DFMAGEN puts a special constraint on the order in which load and store addresses are calculated. It helps the Load/Store Data Flow Machine maintain precise state by generating load addresses in program order, with respect to store addresses. Older store addresses *must* be generated before younger load addresses. Otherwise, younger loads to the same address of a store might execute before the store writes the data to the Data Cache, resulting in a read of stale data.

### 3.4. Floating Point Reservation Station

The Floating Point Data Flow Machine (DFMFPU) is also similar to the DFMFXU. It has an eight entry reservation station. It contains an extra operand *Register Cache* for source3 of floating point multiply/add instructions.

The DFMFPU is connected to one floating-point execution unit which consists of a floating point multiply-add unit and a non-blocking self-timed floating point divide unit. It can select one floating point instruction per cycle. Most floating point instructions are multi-cycle. Some Floating point instructions, like moves, are executed in a single cycle. The floating point multiply/add unit is a fully pipelined four stage unit. It shares a result bus with the floating point divide.

### 3.5. Load/Store Reservation Station

The Load/Store Data Flow Machine (DFMLSU) interfaces to the external Data Caches. It has a twelve entry reservation station which contains *Register Caches* for the store data and the virtual address. It also contains register arrays for opcode and destination tags. The DFMLSU is able to receive two issued instructions, send two 64 bit load/store instructions to the Data Caches, and forward results of two instructions per cycle. It is able to send load instructions to the Data Caches both out of order and speculatively. The DFMLSU does this while maintaining precise state.

The DFMLSU has both fixed and floating point interfaces. It captures both types of source data for stores. A special destination register match array compares store source registers to all the results being broadcasted each cycle. An extra bit written on issue determines whether the register array matches on a fixed or floating point result.

The DFMLSU forwards both types of data for loads. Since the same destination register array is shared between fixed point and floating point, depending on the data type of the load instruction, two different valid bits for the results are broadcast to all of the reservations stations. In this manner, the same physical register numbers do not falsely match between fixed and floating point instructions.

The DFMLSU is able to handle both Big and Little Endian data alignment. For loads, it aligns the data as it returns from the Data Caches. This data is then forwarded to the other reservation stations in the same cycle. For stores, the DFMLSU captures the data in the internal Big Endian format and changes it to Little Endian format, if required, before the store data is sent to the Data Caches. The DFMLSU can handle both types of alignment in the same cycle.

## 4.0. Estimating the Performance Impacts of Microarchitectural Trade-offs

A microprocessor design is composed of various trade-offs between items such as cycle time, number of cycles per instruction, and die area. In order to make reasonable trade-offs, the effects of these trade-offs must be estimated. The performance on a given application can be determined by the following [8]:

**execution time = cycle time * instruction count * CPI**

Cycle time is determined by physical characteristics of the microprocessor such as the maximum gate delays per clock and the semiconductor technology used. The instruction count is determined by the application and its associated input. The instruction count is also dependent on compiler efficiency. The CPI (cycles per instruction) is a function of the microarchitecture of the microprocessor and the compilers used.

In order to estimate the number of instructions executed per cycle (IPC = 1/CPI) of proposed microarchitectures, we developed a trace driven simulator called *TIMER*. To determine the performance of a given benchmark on a specific design, the instruction count is determined by the benchmark and the cycle time is estimated by the design groups. With these three pieces of data, the performance of the system can be estimated.

### 4.1. Trace Driven Simulation

Trace driven simulation is a means of simulating the execution of code without having to fully model all of the execution resources of the actual hardware. [9] This simplification eases performance estimates in numerous ways. The simplification also allows the model to execute at much higher speeds than the detailed logic models. The higher level of abstraction also allows the model to be modified quickly.

An instruction trace consists of a sequence of instruction records. Each record contains an instruction's virtual address, instruction word, effective address, if required, and some flags. The effective address is used for memory accesses and branch targets. The flags are used for items such as the direction of conditional branches. This trace is then fed into a pipeline simulator in order to make performance estimates.

A major pitfall of trace driven simulation is the inability to model the contamination caused by speculative execution. The instruction trace consists only of those instructions which will be issued and committed. In a speculative execution machine, there will be a large number of instructions which are issued but cancelled before they are committed. These instructions will perform operations such as loading unwanted data into the Data Caches and updating the branch prediction bits incorrectly. However, sometimes the unwanted data is used later, and the speculative accesses are beneficial prefetches. These effects cannot be modelled with a trace driven simulator and must be modelled by an execution driven simulator.

## 4.2. Estimating Performance with *TIMER*

In order to estimate the performance of an application, the application must first be traced. We accomplished this by running the application on a SPARC-V9 architectural model called Halsim [10] with tracing enabled. An actual 64-bit Unix kernel is booted on Halsim and the application is run underneath producing a full trace which includes both user and kernel code. The instruction count is also determined with Halsim. The trace is then fed into some filters to reformat the data.

*TIMER* models all major units in the microprocessor, cache, and main memory system. Available options include peak issue rate, maximum instruction window size, cache sizes and latencies, physical register file sizes, reservation station sizes, number of function units and latencies, variable pipeline lengths, and various branch prediction strategies. Once a desired configuration is chosen, the trace file is read and instructions are simulated as they flow through the execution pipeline. *TIMER* can output cycle-by-cycle detailed execution information or summarize the information into relatively brief reports. The reports include information such as issue stall conditions, cache miss rates, branch prediction accuracy, and IPC.

For long running programs, the traces become unmanageably large. In this case the traces are sampled down to a reasonable size. Sample sizes are determined by the number of instructions to "warm up" the microprocessor tables such as the caches. A typical sampled trace will consist of 30 samples of 200,000 instructions each. The number of samples may be adjusted upward or downward depending on the variance of the IPC's.

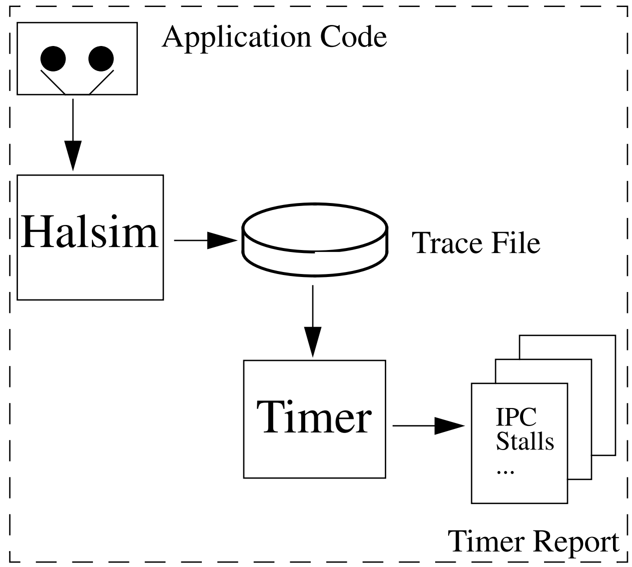


**Figure 4.** Performance Evaluation Flow.

The absolute error margin of *TIMER* vs. gate level simulations on long running codes has been measured to be typically much less than 10%. This indicates the combined error due to higher abstraction levels and speculative contamination is tolerable. It must also be noted that the relative performance estimates are much more precise. An example is the relative performance boost of doubling the instruction cache size. While the absolute IPC may be off by up to 10%, the relative boost is more accurate.

## 4.3. Benchmarks

As a benchmark, we have chosen to use the SPEC92 benchmark suite [11]. This popular suite consists of 6 integer programs (SPECint92) and 14 floating point programs (SPECfp92).

## 5.0. Trade-offs

After choosing an instruction window of 64 and an issue rate of 4, the size and dimensions of the Data Flow Unit changed dramatically over the course of our project. We started out with an approach of trying to maintain a high IPC number while balancing several trade-offs to account for technology limitations. The main focus was on:

**Maintaining IPC while**

**Reducing Cycle Time and**

**Reducing Die Area**

The initial fixed point register file had 128 entries. This allowed for 50 free fixed point registers to be renamed. The floating point register file also had 128 entries which allotted 64/32 free floating point single/double precision registers. The initial number of entries in the different reservation stations was 16 for the DFMFXU, DFMAGEN, and DFMFPU and 24 for the DFMLSU. And the selection mechanism in these reservation stations was aggressive in its choice of which instructions were to be executed in the next cycle.

## 5.1. Register File Sizes

The register file size turned out to be one of the most debated aspects of our project. Adding more physical registers increases performance by reducing issue stalls due to lack of free registers. But adding registers decreases performance by slowing the cycle time. The more registers there are in the register file, the longer it takes to read them. Also, the more registers added, the larger the die area.

Our performance evaluation group found, that on average, there is a branch every sixth instruction. If so, out of the 64 instructions in our instruction window, about 10 will be branches, which do not require a destination register. That leaves at most 54 of 64 instructions which require a fixed point destination register. Out of these 54 instructions, not all of them will have destination registers. Stores write to memory and thus do not use a destination register. And a mixture of fixed point and floating point code would ease the number of free registers required for each data type inside an instruction window.

The results of the benchmarks run on *TIMER* is seen in the next two graphs. From our original baseline of 50 free fixed point registers we decided that we could reduce this number to 38 free fixed point registers at only a slight performance loss. Similar results were found with floating point registers. We reduced the number of free single precision floating point registers from 64 to 48.

This reduced the size of the FXRF from 128 registers to 116 registers. And both instances of the FPRF were reduced from 64 registers to 56 registers. To ease the physical design change, registers were removed in groups of four. With these changes, we were able to shrink the FXRF by 8% and shrink the FPRF by 10%. This change also helped us meet our cycle time goal.

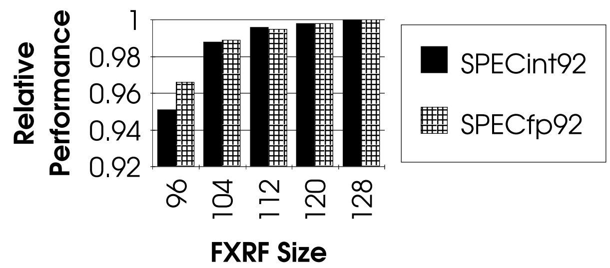


**Figure 5.** FXRF relative performance.

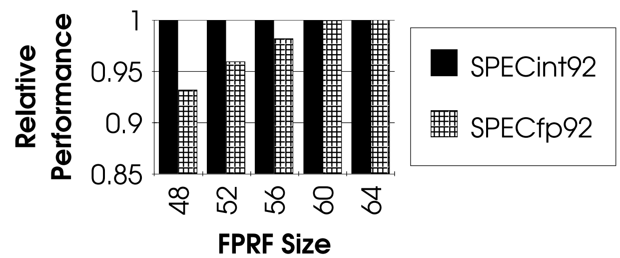


**Figure 6.** FPRF relative performance.

## 5.2. Reservation Station Sizes

The specific size of the reservation stations was chosen for several reasons. The smaller the number of entries in a reservation station, the faster cycle time it will be able to achieve. The physical design size is also directly related to the reservation station size. But, the smaller the number of entries, the fewer the number of instructions that can be issued to it before it fills up. When a reservation station fills up, issue of instructions to that reservation station stalls and IPC is reduced.

**DFMFXU, DFMAGEN, DFMFPU Sizes**

Die area and cycle time dictated the reservation station sizes for the DFMFXU, DFMAGEN, and DFMFPU. The number of entries was reduced from 16 to 8. In doing so, the die area for these regions was reduced by over 50%. And the cycle time decreased by 5-7% by removing two stages of logic from the critical path. As seen by the following performance graphs, the IPC decreased with these changes by less than 1%. The relative performance in these graphs is based on the initial IPC number of the reservation station sizes. Overall performance increased since the cycle time improvements were greater than the IPC loss with the added advantage of a die area reduction.

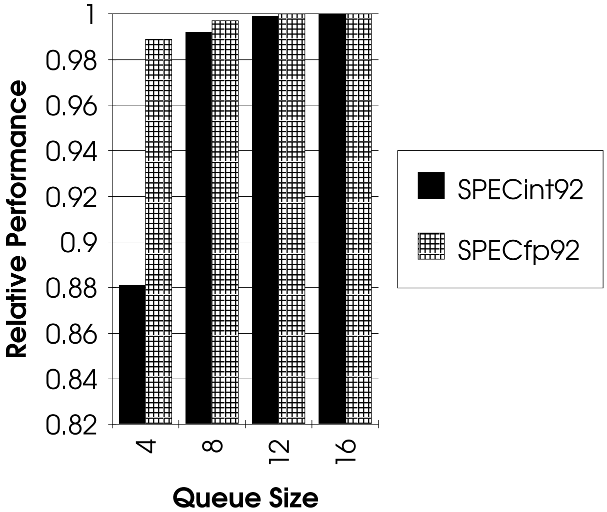


**Figure 7.** DFMFXU relative performance.

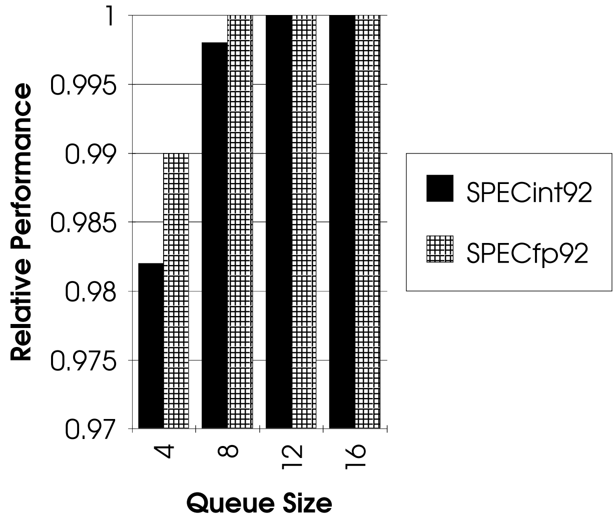


**Figure 8.** DFMAGEN relative performance.

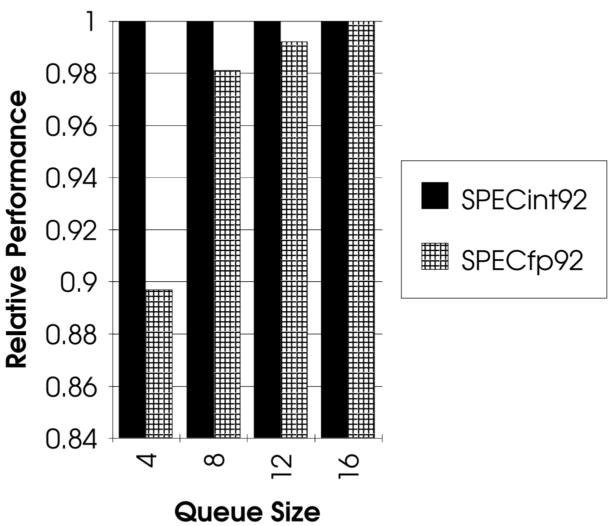


**Figure 9.** DFMFPU relative performance.

### DFMLSU Size

The number of entries in the DFMLSU had one extra constraint compared with the other Data Flow Machines. It had to be able to handle a spill/fill trap (a series of 16 loads or stores) without filling up and stalling the issue rate. Also, due to Data Cache misses, DFMLSU execution time can be much longer than the other Data Flow Machines, so a slightly larger DFMLSU was also desirable. After starting with a 24 entry DFMLSU, a 12 entry reservation station was chosen. This reduced the area required by over 50% while decreasing the cycle time by about 6%.

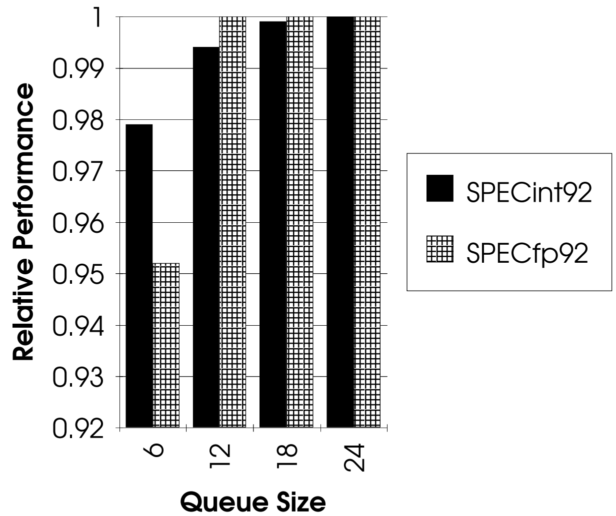


**Figure 10.** DFMLSU relative performance.

## 5.3. Number of free slots in the DFMs.

Logic to track the number of free slots in the reservation stations was moved from the issue unit to the reservations stations to provide a more up-to-date count of the free slots available. This increased IPC by reducing or eliminating issue stalls due to the reservation stations being full. The number of free slots available in the DFMLSU is updated in the same cycle that HIT status returns from the Data Caches. This allows reservation station entries to be reused as soon as they become free, instead of two cycles later when the instruction commits.

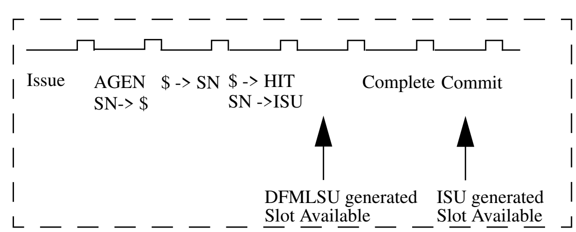


**Figure 11.** DFMLSU generated number of slots available.

The method that the DFMFXU, DFMAGEN and DFMFPU used to compute the number of slots available was modified slightly to reduce cycle time. These reservation stations started out using the DFMLSU's method. If the reservation station was full and two instructions were executing, these Data Flow Machines would report to the Issue Unit that there were two slots available next cycle. When the selection mechanism was modified, this became a timing problem. The compromise made was to not report a slot available for the second instruction selected for execution, if the reservation station was full and two instructions were executing. This reduced our cycle time by about 5%, while decreasing IPC slightly. Since the DFMFPU has only one execution unit, it can only reclaim one reservation station slot at a time, so floating point performance was not impacted. And it was not as critical for the DFMFXU or DFMAGEN to reclaim reservation station slots as fast as the DFMLSU, since they do not stall on Data Cache misses.

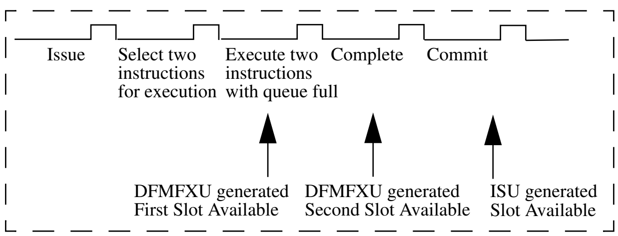


**Figure 12.** Non-DFMLSU generated number of slots available.

## 5.4. Reservation station selection algorithm trade-offs

### DFMFXU and DFMAGEN (before)

There were several trade-offs made before the final execution selection algorithm was chosen. Each Data Flow Machine, except the DFMFPU which only has one execution unit, can select two instructions for execution each cycle. At the beginning of the project, a very aggressive and optimistic approach was implemented. As timing and physical design work was done, the selection algorithm was modified several times to help reduce the cycle time and die area used. The original order of execution selection was:

1. Choose the oldest ready instructions that modified the condition codes.
2. Choose the oldest ready instructions.
3. The first instruction being issued in the current cycle, if ready, could go to either execution unit.
4. The second instruction being issued in the current cycle, if ready, could go to either execution unit.

An instruction is said to be ready if all of its operands have been captured. A precedence matrix was used to choose the oldest ready instruction which modified the condition codes. By choosing these first, branches could be resolved faster and the backup time due to mispredicts would be minimized. If there were no instructions that modified condition codes that were ready to execute in the reservation station, then the oldest ready instruction would be chosen. This first instruction chosen was then removed from the list of executable instructions for the second selection. Again, the oldest ready instruction which modified condition codes had the highest priority, followed by the oldest ready. If nothing was selected from the instructions already in the reservation station, the instructions that were being issued were allowed to be chosen for execution. Since these instructions are

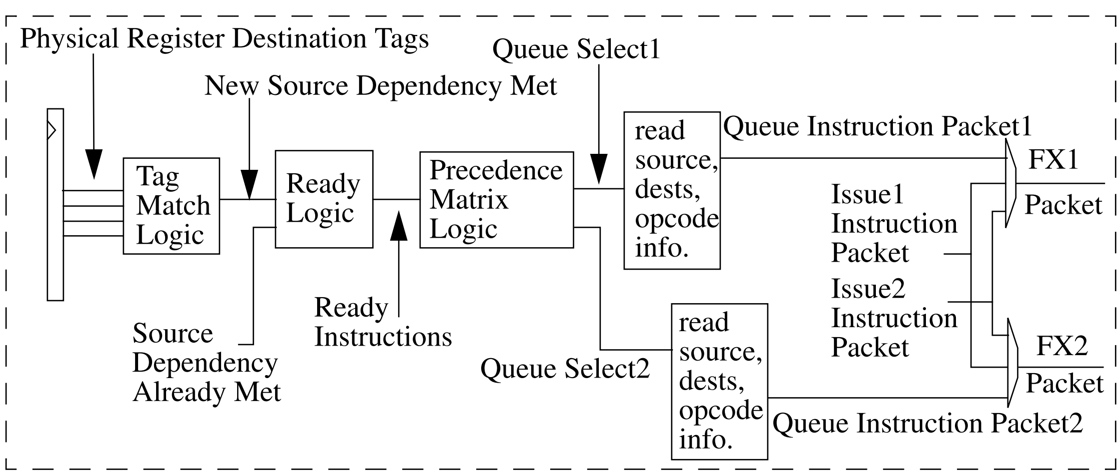


**Figure 13.** Original execution selection algorithm.

issued in order, the first instruction packet received a higher priority. Figure 14 shows the selection algorithm cycle.

### DFMFXU and DFMAGEN (after)

By the end of the project, timing and physical constraints allowed for the following selection order:

1. The two oldest ready instructions whose final source dependency was met last cycle are selected.

This allowed for a timing speedup of 25% in two ways. First, the tag match logic used to generate the ready signal was removed from this timing path. Second, the throughput time of the Precedence Matrix was reduced while reducing the support logic and the size of the Precedence Matrix. This also reduced the size and complexity of the Precedence Matrix by removing the condition code logic.

2. Two "random" instructions whose final source dependencies are being generated this cycle are selected.

An instruction was considered ready if all of its source dependencies were met. In the beginning, this included instructions whose dependencies were being met in the current cycle. Timing constraints in the tag matching logic caused instructions whose dependencies were being met in the current cycle to be selected after instructions who were younger but already had all of their sources captured. This did not cause much performance loss: instructions were chosen for execution (just not in the most optimal order). Additionally, Timer found that on average, there were less than or equal to two instructions ready to execute per cycle.

3. The first instruction being issued in the current cycle is allowed to be executed only in the first execution unit and the second instruction being issued is allowed to be executed only in the second execution unit if nothing is valid in the reservation station.

Physical constraints restricted the first instruction being issued from going to the second execution unit and the second instruction being issued from going to the first execution unit. This simplified the design and eased timing. Also, if there are instructions in the reservation station which are valid, selecting instructions that are being issued in the current cycle is not allowed. This reduced cycle time by about 5% with only a small affect on IPC. Figure 15 shows the current selection algorithm cycle.

## 5.5. DFMLSU selection algorithm

Like the other Data Flow Machines, a precedence matrix in the DFMLSU is used to select the next two instructions for execution. An extra requirement for the DFMLSU execution selection mechanism, besides capturing all source data, is to maintain precise state. Stores and younger loads to the same address as older stores, must execute in program order. Loads that are not to the same address as older stores can execute both speculatively and out of order.

The DFMLSU has the ability to kill the last transaction (KLT) that it sent to the Data Caches. KLT accomplishes two things. It allows the DFMLSU to send virtually any instruction to the Data Cache, whether it is ready to be sent or not. If it is not ready to be sent, then it is killed. This generates a higher memory bandwidth at virtually no cost to performance. The second thing that KLT allows, is a

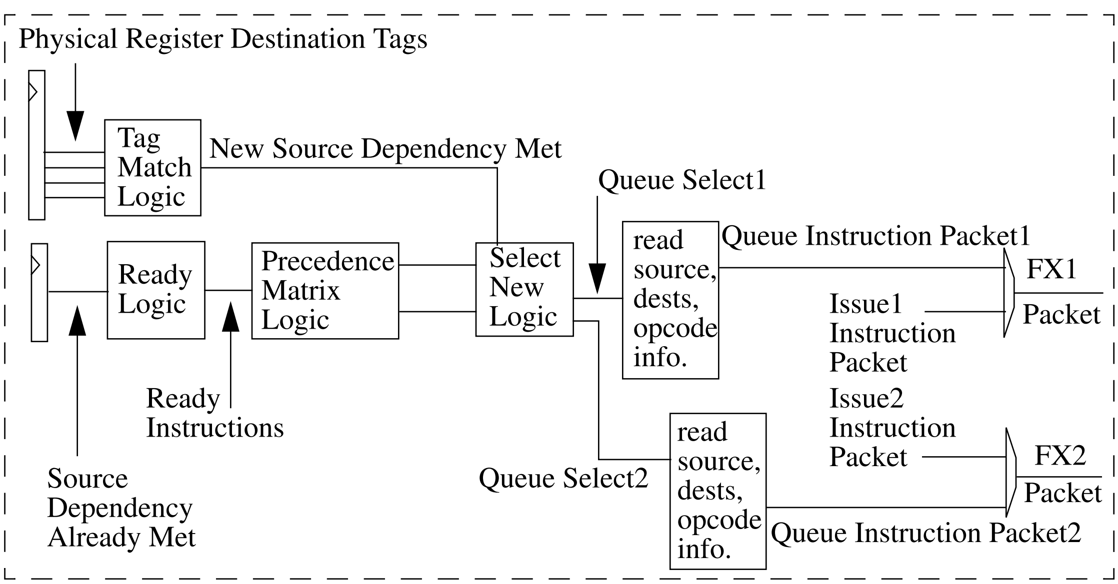


**Figure 14.** Final selection algorithm pipeline stage.

speedup in inter-chip cycle time. The logic to determine whether an instruction can be sent to the Data Caches is quite complex. Instructions would not be allowed to bypass directly to the Data Caches when they generate their addresses if KLT was not present. This allows loads to have a 3 cycle access time to the Data Caches, instead of the 5 cycle access time for store instructions.

If nothing is chosen to be sent to a Data Cache from the DFMLSU reservation station, then instructions which are generating addresses this cycle can be sent to a Data Cache. Due to inter-chip cycle time constraints, a load command is always sent to a Data Cache if nothing is chosen to execute from the reservation station and a load/store instruction generates an address this cycle. These "bypassing" instructions are killed in the next cycle if they were not loads or if they were loads that are to the same address as older store instructions.

The DFMAGEN supports the DFMLSU with speculative execution. It provides key command bits to the Data Caches for instructions that are bypassing to the Data Caches this cycle. This reduces cycle time by not requiring that these bits be read from their register arrays in the DFMLSU.

## 6.0. Conclusions

Due to technological limitations, several trade-offs had to be made in order to implement a Restricted Data Flow algorithm. Cycle time and die area forced us to further restrict the Data Flow ideas. The amount of fixed and floating point registers able to be renamed was limited, the sizes of the reservation stations were reduced, and the selection mechanisms were modified. Even with these trade-offs, we were still able to produce a high performance RISC microprocessor based on a Restricted Data Flow algorithm.

Improvements to technology will allow us to expand on our current design. We will be able to relax some of the trade-offs that we were forced to make. For example, with the next generation of semiconductor technology, the two Register Files will shrink in size. This will allow us to add more free registers and/or more register windows. As seen below in Table 2, this would be a boost to our performance at only a small cost to cycle time.

## 7.0. Acknowledgments

We would like to thank the various teams that made it possible to implement our design. Without help from rest of the design groups, the verification, tool and other support teams, this project couldn't have been completed. Professor Yale Patt consulted with us early in the project. John Gmuender was the original designer of the DFU whose ideas we modified and built on.

**TABLE 2.** Future Work...

| Register Windows | Free Registers (total - needed) | Difference from current | Cycle Time | Performance Change | Notes |
|---|---|---|---|---|---|
| 4 | 116 - 78 = 38 | 0 | base | baseline | Current: fastest cycle time |
| 4 | 128 - 78 = 50 | 12 | base + delta | +0.0% | Handles average case without issue stalls. |
| 4 | 144 - 78 = 66 | 28 | base + 2 * delta | +0.0% | Never a stall due to lack of free registers |
| | | | | | |
| 5 | 116 - 94 = 22 | 0 | base | +0.8% | Fewer spill/fill traps, more issue stalls. |
| 5 | 132 - 94 = 38 | 16 | base + delta | +2.9% | Fewer spill/fill traps, Base number of issue stalls. |
| 5 | 144 - 94 = 50 | 28 | base + 2 * delta | +3.0% | Handles average case without issue stalls. |
| 5 | 160 - 94 = 66 | 44 | base + 3 * delta | +3.0% | Never a stall due to lack of free registers. |
| | | | | | |
| 6 | 116 - 110 = 6 | 0 | base | -41.0% | Fewest spill/fill traps, MANY issue stalls. |
| 6 | 148 - 110 = 38 | 32 | base + 2 * delta | +5.5% | Fewest spill/fill traps, Base number of issue stalls. |
| 6 | 160 - 110 = 50 | 44 | base + 3 * delta | +5.6% | Handles average case without issue stalls. |
| 6 | 176 - 110 = 66 | 60 | base + 4 * delta | +5.6% | Never stalls due to lack of free registers, slowest cycle time. |

## 8.0. References


[1] J. Boney, “SPARC Version 9 Points the Way to Next Generation RISC”, SunWorld, October 1992, pp 100-105.

[2] G. Shen, et. al., “A 64b 4-Issue Out-of-Order Execution RISC Processor”, ISSCC, (Feb. 1995), pp 170-171.

[3] R. Iannucci, “Toward a Dataflow/Von Neumann Hybrid Architecture”, The 15th Annual International Symposium on Computer Architecture, (May 1988), pp 131-140.

[4] M. Shebanow, Y. Patt, et. al. , “Single Instruction Stream Parallelism Is Greater than Two”, Proceedings of the 18th Annual Symposium on Computer Architecture, (May 1991) pp. 276-286.

[5]Y. Patt, W. Hwu, et al, “Experiments with HPS, a Restricted Data Flow Microarchitecture for High Performance Computers”, Digest of Papers, COMPCON 86, (March 1986), pp. 254-258.

[6]N. Patkar, et. al., “Microarchitecture of HaL’s CPU”, COMPCON, (March 1995), pp. 259-266.

[7] C. Asato, et. al., “A 14-Port 3.8ns 116 Word 64b Read-Renaming Register File”, ISSCC, (Feb. 1995), pp. 105-106.

[8] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann, 1990, p. 36.

[9] R. Jain. *The Art of Computer Systems Performance Analysis*, Wiley, 1991, p 404.

[10] D. Barach, et al. “HALSIM - a very fast SPARC V9 Behavioral Model”, International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS95), January 1995.

[11] J. Reilly, “A Summary of the SPEC Benchmark Suites”, SPEC Newsletter, March 1994, p. 3.