Integer / Floating-point Reconfigurable ALU

LA-UR #99-5535

November 1999

Dominique LAVENIER - Yan SOLIHIN - Kirk CAMERON

Los Alamos National Laboratory
Computer Research and Application
Los Alamos, NM 85545, USA

Abstract

This report describes a functional unit able to perform both usual integer operations and floating-point additions. Basically, the architecture extends the structure of a floating-point adder, so that hardware is re-used for both operations. The time for reconfiguring this unit depends on the operations of the instruction flow. On average, the penalty is estimated as one cycle. This Reconfigurable ALU (R-ALU) uses roughly the same die space as a floating-point adder.

1 Motivation

Current microprocessor architectures house separated functional units for executing integer and floating-point operations. Depending on the microarchitecture organization (type and number of functional units) the Instruction-Level Parallelism (ILP) loss can be significant due to the mismatch between the instruction stream and the available hardware resources.

In [1] a method for quantifying the frequency of instructions with respect to the limit of functional unit allocation is developed. The committed instruction sequence are viewed as a sequential stream of instruction types and the inter-arrival distances between instructions can be measured and profiled.

A first utilization of this technique is to provide a list of the most frequently occurring clusters of instructions. For floating-point intensive applications, namely Swim, Wave5 and Su2cor, the list of significant instructions is similar to the integer intensive codes Compress95,

ijpeg, li, and kmeans. These are the "important" instructions for these particular codes. A reconfigurable unit that provides support for these types of instructions is likely to achieve performance gain provided the switching penalty is minimal. These instructions are given in annex.

For all the observed codes, loads and stores occurred with high cluster frequency. This is intuitive since they are typically necessary for many types of operations. While the characterization does not consider the influence on performance due to memory latency, it is certainly the case that improvements in latency or overall memory performance would certainly impact the overall performance. But in this context, only on-chip performance is considered without the influence of memory since the goal of a reconfigurable unit is to improve the service rate of instructions without consideration to memory latency. Loads and stores will achieve a performance boost from a reconfigurable unit that gives additional bandwidth to memory address calculation due to the sheer number of loads and stores taking place.

Integer-add operations also are quite clustered among all the codes. This is expected in integer-intensive codes, but perhaps the magnitude of their presence in floating-point intensive codes is not so intuitive. Nonetheless, the floating-point codes Swim, Wave5, and Su2cor each show frequency distributions that outweigh their floating-point add counterparts significantly. This shows that a reconfigurable unit providing extra bandwidth to integer-add operations should provide a performance boost. Also, the penalty incurred by switching from integer-add to floating-point add resulting in cycle delay could be canceled out by the gain in integer performance afforded by a reconfigurable unit. In other words, a tradeoff is possible between switching penalty and integer bandwidth performance gain since the quantity of clustered integer operations is typically two or three times larger than the quantity of floating-point add operations.

This discussion provides the motivation behind our choices of including and excluding functionality in the defined reconfigurable unit. Particularly, the goal is to provide extended integer execution bandwidth while maintaining the power provided from reconfiguring as a floating-point unit.

The rest of this report is organized as follows: section 2 details the R-ALU architecture. Section 3 discusses some timing considerations. Section 4 deals with the reconfiguration issue of the R-ALU. Section 5 concludes this report.

2 The R-ALU Architecture

This section presents the architecture of a functional unit which can perform both integer or floating-point operations. We refer to it as a reconfigurable ALU (R-ALU) since it requires some delay for switching from integer mode to floating-point mode or from floating-point mode to integer mode. The idea behind the proposed architecture is to adapt the hardware

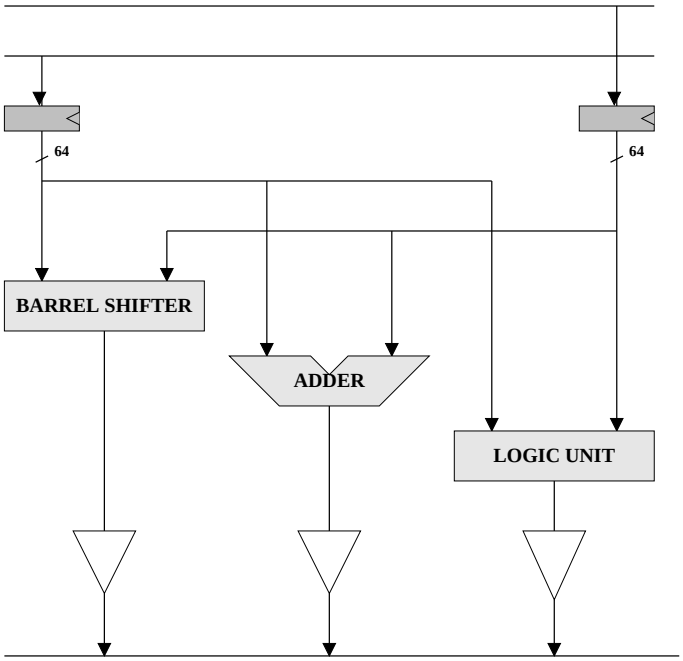


Figure 1: *64-bit integer ALU performing addition (subtraction), shift (left and right) and logic operations*

by analyzing the instruction stream and mutating the R-ALU to integer ALU if the integer operations exceed the floating-point operations or mutating it to a floating-point ALU in the opposite case.

Based on the profiling results, as explained in the previous section, addition, shift and logic operations are the most frequent integer operations. At the same time, floating-point additions represent the major operations in scientific codes. Consequently, the R-ALU is restricted to those operations and can be reconfigured either as:

- an **an integer ALU**

  in this mode the R-ALU performs 64-bit integer operations: addition (`ADD`), subtraction (`SUB`), shift left (`SLL`), shift right (`SRL`) and usual logic operations (`OR`, `AND`, `XOR`, ...).

- a **a floating-point ADDER**

  in this mode, the R-ALU performs double precision IEEE standard 754 floating-point addition (`FP-ADD`).

The architecture of the R-ALU is a hybrid between integer and floating-point architectures. Before describing the principle of the architecture, we first review the architecture of integer and floating-point operators which are currently implemented in the modern microprocessors. Figures 1 and 2 represent respectively two simplified architectures of integer and floating-point units.

The integer ALU has three main components: a 64-bit adder, a barrel shifter and a logic unit. The critical path comes obviously from the 64-bit adder – for example, a 64-bit addition must be performed in 2 ns, if a 500 Mz frequency is expected.

The floating-point adder has a more complex structure. It requires the following (sequential) steps:

1. align the decimal point of the operand that has the smallest exponent;
2. add the significants;
3. normalize the result;

Roughly, each of these operations can be done in one cycle, leading to the 3 stage pipeline adder drawn figure 2. The LOP box computes the leading one prediction in parallel to the addition of the two significants. This is needed to normalize the result. The adder performs a 54-bit addition. Actually, two additions are done concurrently: the sum of the two significants and the sum plus one. This is also needed to speed-up the next stage. This extra operation does not slow down the addition: only an extra carry propagation mechanism is required. For sake of clarity, the rounding mechanism has not been represented.

From the architectures of these two units, one can observe that both require a large adder together with right and left shift capabilities.

The architecture of the R-ALU follows the architecture of the floating-point adder. However, the hardware has been modified for allowing integer operations. The modifications over the architecture of a floating-point adder are:

- extension of the adder from 54 bits to 64 bits;
- substitution of the 54-bit right shifter by a 64-bit barrel;
- insertion of some programmable switches along the data-path.

Figure 3 shows the architecture of the R-ALU. The circles represent the programmable switches. The R-ALU takes two operands as inputs, and has two outputs dedicated respectively to integer and floating-point results.

When configured as an integer ALU, the swap unit is disabled (switch `RS1a`). Hence, the input of the barrel shifter is `B1`; it is controlled by the instruction decoder through the switch `RS1b`. The two inputs of the adder are respectively connected to `A1` and `B1` by the two switches `RS2a` and `RS2b`.

When configured as a floating-point adder, the inputs of the adder come from the first stage of the pipeline. The barrel shifter is controlled by the operations performed on the exponents. In that scheme, only the 54 least significant bits of both the adder and the barrel are used.

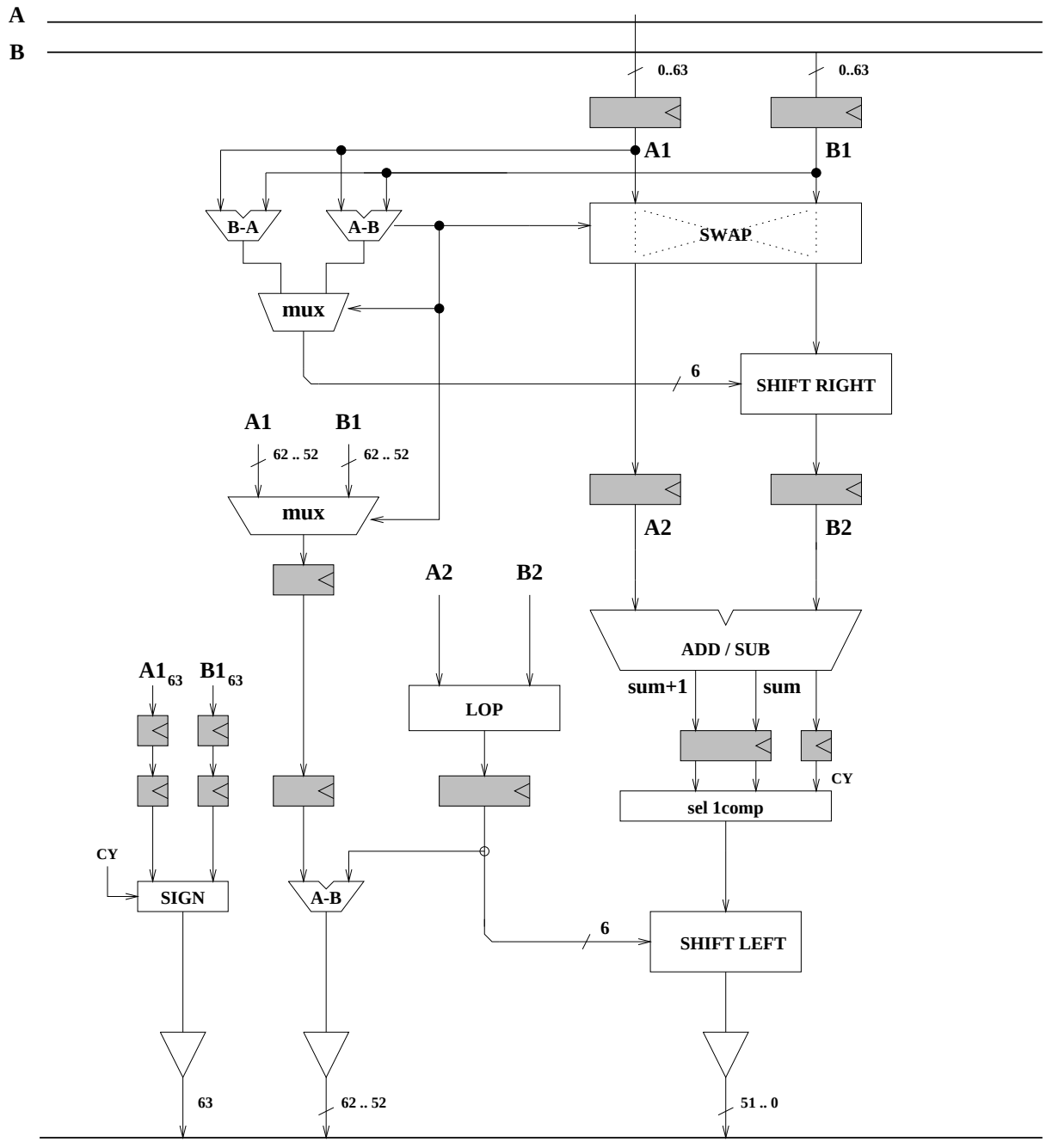


Figure 2: *double precision IEEE 754 floating-point adder*

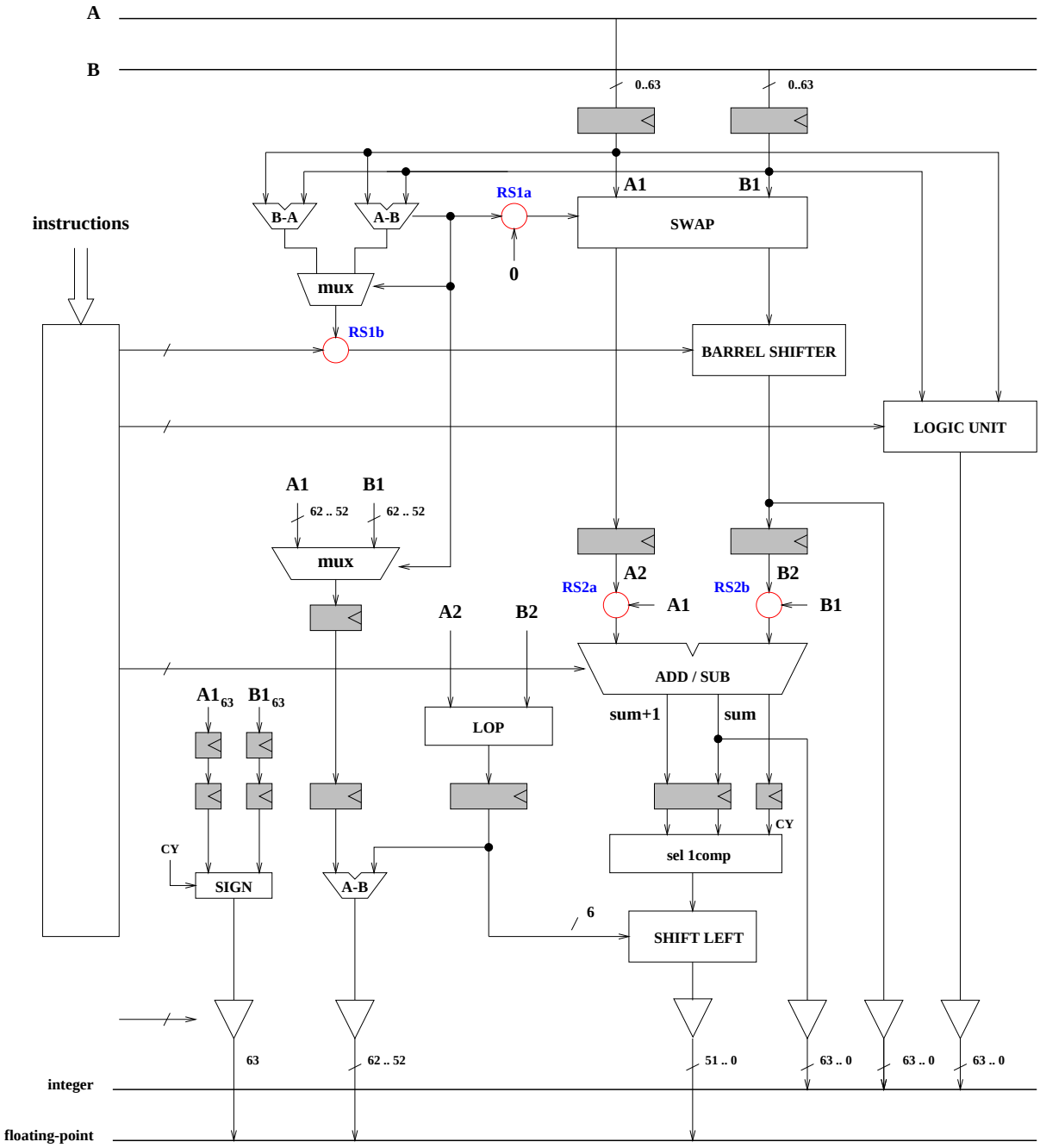


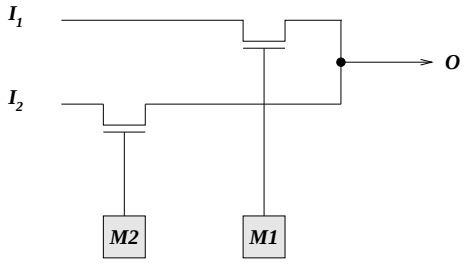
Figure 3: *Reconfigurable ALU*

3 Timing consideration

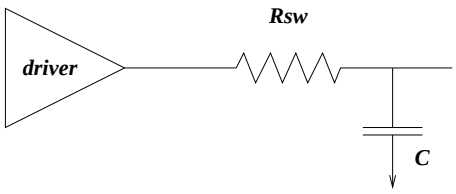
The first assertion we make is that, whatever the technology used, the 64-bit addition is the critical path. When using the Carry Lookahead technique, an addition can be performed in time $O(log\ n)$, in our case, approximatively equal to $6 \times \delta$ where δ is the switching time of an elementary gate.

In the R-ALU, the critical path is the carry propagation of the 64-bit adder, plus the propagation time through the programmable switches `RS2x` on the input operands. The insertion of only one extra gate on this critical path has an immediate effect: it decreases the frequency by more than 15%. Actually, the delay of the programmable switches is less than an elementary gate if we consider that the connection has been set previously, that is during the reconfiguration step. Furthermore, delays induced by the programmable switches can be compensated by resizing a few MOS transistors, as explain below.

Basically, a programmable switch is designed as follows:



For example, to make a connection between $I1$ and O the bit-memory $M1$ must be set to one, making the pass-transistor $T1$ saturated and equivalent to a resistance. In that state, the equivalent electronic diagram of the connection becomes:



C is the load capacitance and Rsw is the resistance of the pass-transistor.

The delay (Δ) is given by:

$$\Delta = 0.7 \times (Rsw + Rd) \times C$$

Rd is the output resistance of the driver.

The delay introduced by the insertion of the switch is thus equal to $Rsw \times C$. The better way to suppress this delay is to have a stronger driver. This is equivalent to resizing the

MOS transistors of the output stage, and thus reducing the output resistance. Suppose a new driver with a resistance equal to $Rd/2$ and a pass-transistor also with a resistance equal to $Rd/2$, then the new delay Δ_{new} becomes:

$$\Delta_{new} = 0.7 \times (Rd/2 + Rd/2) \times C = 0.7 \times Rd \times C$$

This delay is equivalent to a connection without a switch.

To summarize, inserting switches along the data-path does not introduce extra delay if one takes care of resizing correctly the couple (driver/switch). The only effect will be an increase in the power consumption since the drivers are more powerful.

4 Reconfiguration issue

In the previous section, we assume the switches are stabilized for a maximum frequency. In other words, switches cannot be set at the beginning of any working cycle: this would introduce the switching time of the pass-transistor into the critical data-path, and decreasing the frequency. Hence, the R-ALU configuration must be set before the beginning of any operation execution.

The number of configurations of the R-ALU being limited to 2, no specific memory hardware is required. Only one bit is needed for switching from a configuration to another one. Consequently, the minimum reconfiguration time is one cycle.

One must also wonder about the latency of the R-ALU. In integer mode, the R-ALU latency is one cycle, in the floating-point mode, the latency is three cycles. This must be taken into consideration for managing the R-ALU switching.

In the following, we suppose that the reconfiguration of the R-ALU is pipelined: if the reconfiguration starts at cycle i, then the first stage (switches `RS1a` and `RS1b`) is reconfigured during this cycle and the second stage (switches `RS2a` and `RS2b`) is reconfigured during cycle $i + 1$.

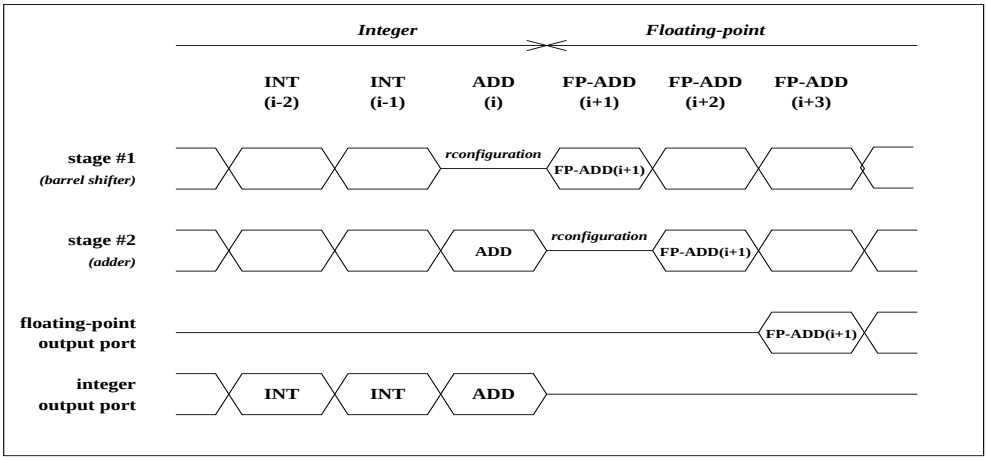
Note that the hardware for executing logical operations is independent: it does not contribute to floating-point addition. Thus, no reconfiguration is required for executing a logic operation.

The next two sub-sections detail the mutation penalty when switching first from integer to floating-point mode and, second, from floating-point to integer mode. Depending of the integer instruction flow, the time for reconfiguring the R-ALU varies. In the following, a floating-point addition is denoted `FP-ADD`, an integer addition (or subtraction) `ADD`, a shift operation `SHIFT`, and a logical operation `LOG`. An integer operation is denoted `INT` (it represents either an `ADD`, `SHIFT`, or `LOG` operation.

4.1 Switching from integer to floating-point mode

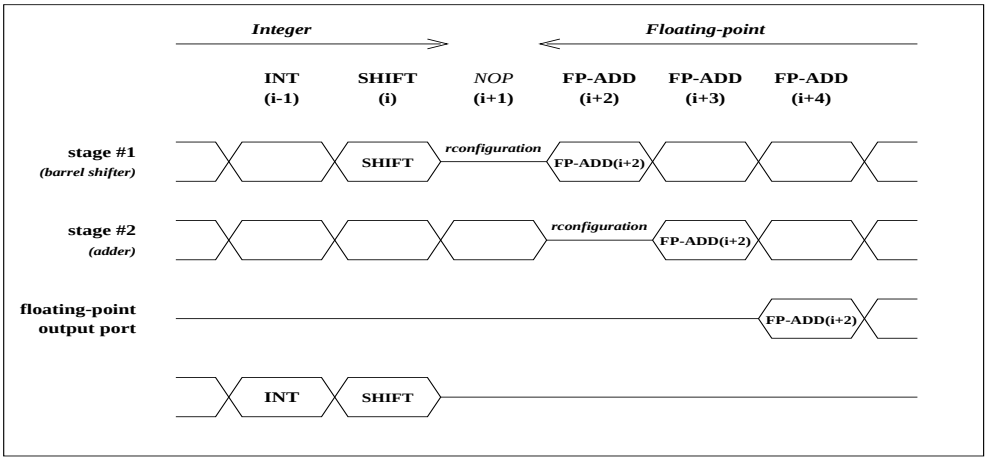
When switching from integer to floating-point mode, two situations might occur:

1. the last integer instruction is a ADD/SUB or a LOGICAL operation.



   Since these operations do not use hardware required by the first pipeline stage of a floating-point addition, this stage can be reconfigured when executing an `ADD` or a `LOG` operation. If such an integer operation is executed during cycle i, the reconfiguration process can start at the beginning of cycle i. **This situation does not require an extra cycle for reconfiguring the R-ALU.**

2. the last integer instruction is a shift operation.

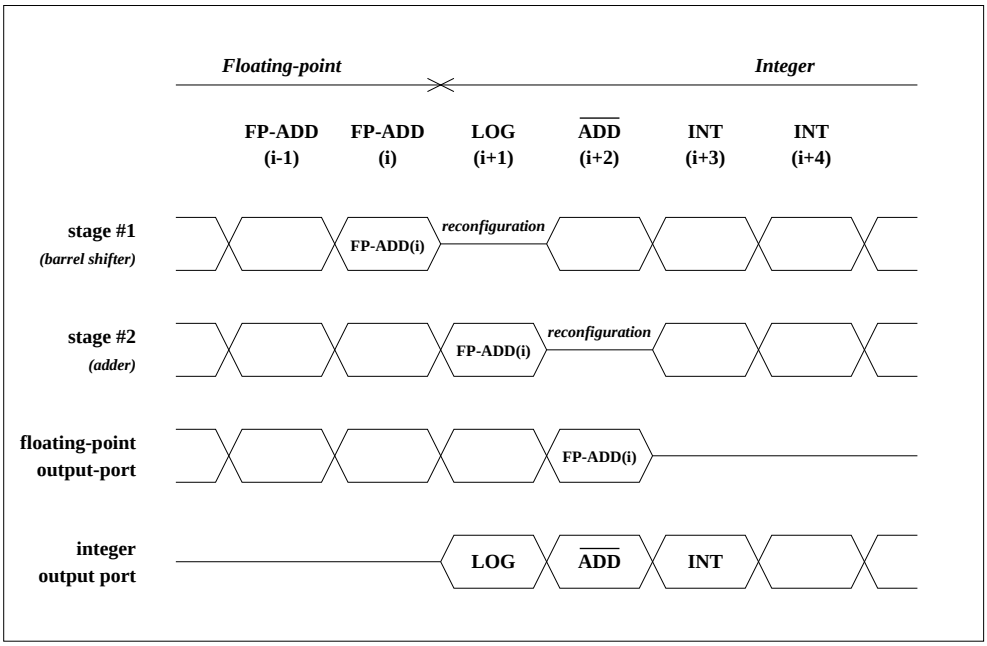


   Since this operation uses the barrel shifter, which is also used in the first step of the floating-point addition, the reconfiguration cannot overlap with computation. **In this situation, one cycle is lost for reconfiguration purpose.**

4.2 Switching from floating-point to integer mode

When switching from floating-point to integer mode, four cases must be considered according to the next two integer instructions:

1. the next instruction is a logical operation **not followed** by an addition.

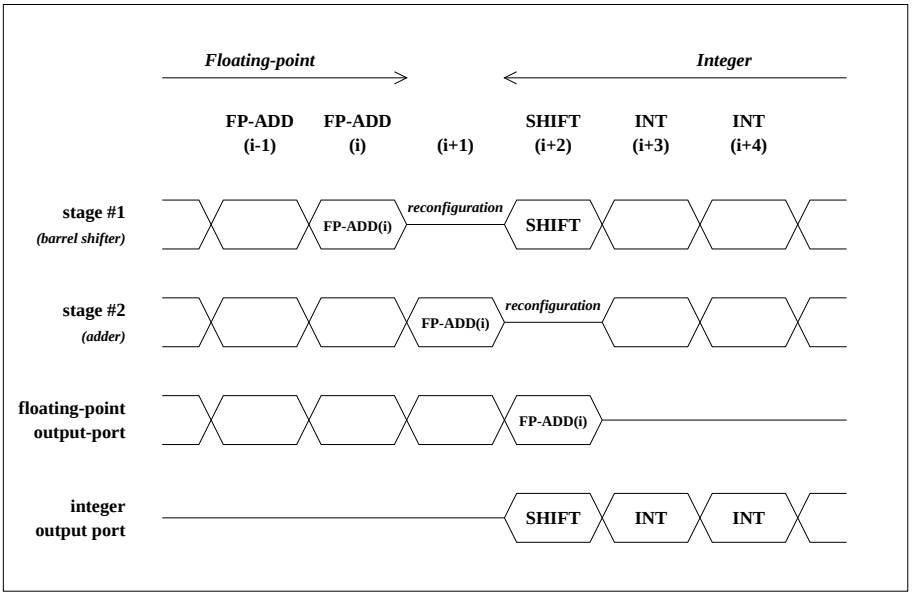


   Since a logical operation does not use the hardware required by a floating-point operation, it can immediately follow a floating-point operation. During execution of it first stage reconfiguration can be accomplished. Note that the results of the logical operation are available one cycle before the result of a floating-point operation. Remember that the R-ALU has two output ports, one connected to the integer register file, and the other to the floating-point register file. **This situation does not require an extra cycle for reconfiguring the R-ALU.**

2. the next instruction is a logical operation **followed** by an addition.

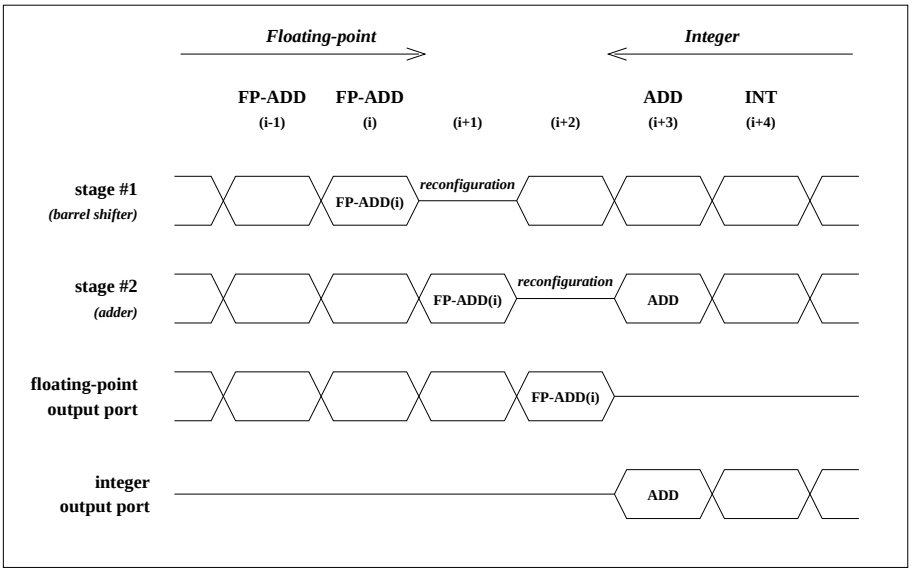
   In this case, the integer addition cannot be performed since the adder is used for executing the floating-point addition. As a matter of fact, if a `FP_ADD` begins at cycle i, it uses the adder at cycle $i+1$, and the reconfiguration cannot be done before cycle $i+2$. Thus an integer addition can begin to execute only on the next cycle, that is cycle $i+3$. **This situation needs one extra cycle for reconfiguration.**

3. the next instruction is a shift operation.



Since a shift operation uses the barrel shifter which is also used in the first stage of a floating-point addition, **one reconfiguration cycle is needed between the execution of these two operations.**

4. the next instruction is a ADD/SUB operation.



In this situation, the adder is the bottleneck. If the instruction `FP-ADD` is executed at cycle i, then the adder is used at cycle $i+1$, allowing the second stage to be reconfigured at cycle $i+2$, and instruction `ADD` to start at cycle $i+3$. **In this situation, two cycles are needed for switching from floating-point to integer**.

4.3 Average number of cycles dedicated to reconfiguration

The following table summarizes the various situations:

	inst i	inst i+1	inst i+2	cycles
int to float	LOG or ADD	FP-ADD	FP-ADD	0
	SHIFT	FP-ADD	FP-ADD	1
float to int	FP-ADD	LOG	not ADD	0
	FP-ADD	LOG	ADD	1
	FP-ADD	SHIFT	INT	1
	FP-ADD	ADD	INT	2

The columns inst i, inst i+1 and inst i+2 represent the instruction flow. The R-ALU is switched between instruction i and instruction i+1. The last column indicates the number of cycles required for switching the R-ALU.

The average number of cycles dedicated for reconfiguring the R-ALU depends both on the instruction distribution and on the strategy to determine when to switch from FP-to-INT or INT-to-FP. If we assume that most of the integer instructions are ADD/SUB instructions, then the average cycle to reconfigure the R-ALU is equal to one (no cycle for the ADD/FP-ADD switch, two cycles for the FP-ADD/ADD switch).

5 Conclusion

The hardware cost of the R-ALU is roughly the cost of a floating-point adder. Extending the adder from 54 bits (current implementation of the IEEE standard 754 for binary floating-point arithmetic) to 64 bits does not provide a frequency limitation since the integer unit already perform such operation in one cycle.

The re-programmability is provided by a few programmable switches. The delay introduced by these devices can be canceled thanks to a clever transistor resizing. The overall effect will be a small (probably insignificant) increase in power consumption.

The average number of cycles dedicated to reconfigure the R-ALU can roughly be estimated to one. Due to the pipeline organization of the floating-point adder, the switching from floating-point to integer needs two cycles, while switching from integer to floating-point does not require extra cycles.

References

[1] K.W. Cameron, Y. Luo, Instruction-Level Microprocessor Modeling of Scientist Application, Lecture Note on Computer Science 1615, Proceedings of the Second International Symposium on High Performance Computing, pp. 29-40, 1999.

[2] H. A. Al-twaijry, S.F. Oberman, S. T. Fu, M. J. Flynn, The Snap project: Building Validated Floating Point Units, Journal of Computer Science, vol 4, no 2, pp. 99-109, 1998.

[3] J.D. Bruguera, T. Lang, Leading-One Prediction Scheme for Latency Improvement in Single Data-path Floating Point Adders, *Proceedings of the International Conference on Computer Design (ICCD'98)*, 1998.

[4] J. L. Hennessy, D. A. Patterson, Computer Architecture, A Quantitative Approach, *Prentice Hall*, 1998.

[5] J. L. Hennessy, D. A. Patterson, Computer Organization and Design, The Hardware/Software Interface, *Morgan, Kaufmann*, 1998.

[6] M. J. S. Smith, Application-Specific Integrated Circuit, *Addison Wesley*, VLSI Systems series, 1997.

[7] *MIPS R10000 Microprocessor User's Manual*, MIPS, 1996.

[8] A. Ahi et al., R10000 Superscalar Microprocessor, *Hotchip'95*, 1995.

[9] A. Seznec, F. Lloansi, Etude des Architectures des Microprocesseurs MIPS R10000, UltraSparc et Pentium Pro, *Rapport de recherche IRISA*, no 1024, 1996.

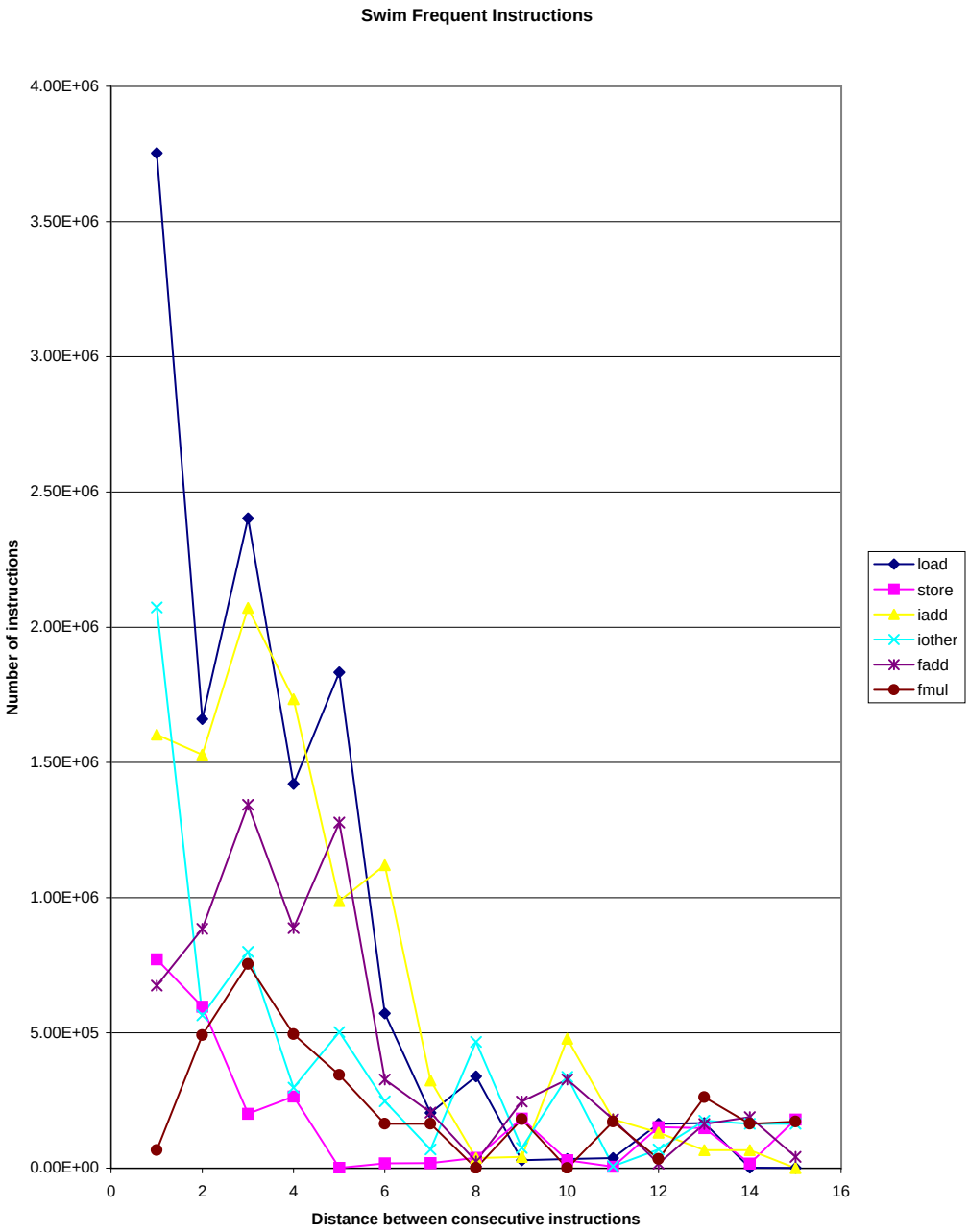
[10] N. Quach, J. Flynn, Design and implementation of the SNAP floating-pont adder, *technical report CSL-TR-91-501*, Stanford University, 1991.

[11] N. Quach, J. Flynn, Leading One Prediction: implementation, generalization, and application, *technical report CSL-TR-91-463*, Stanford University, 1991.

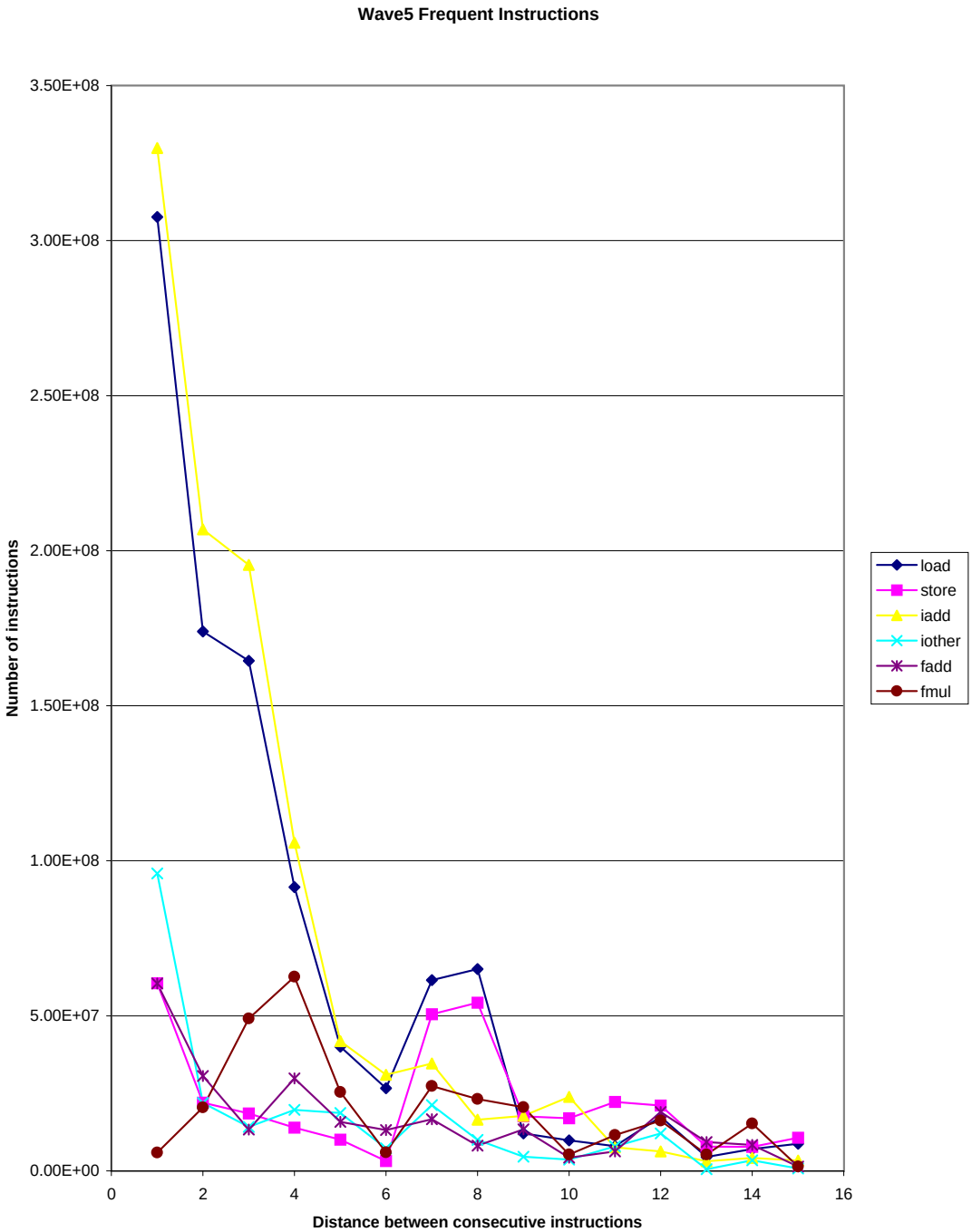
[12] V. G. Oklobdzija, Architecture For Single-Chip ASIC Processor With Integrated Floating Point Unit, *Proceedings of the 21st Hawaii International Conference on System Sciences*, pp. 1-9, 1988.

6 ANNEX

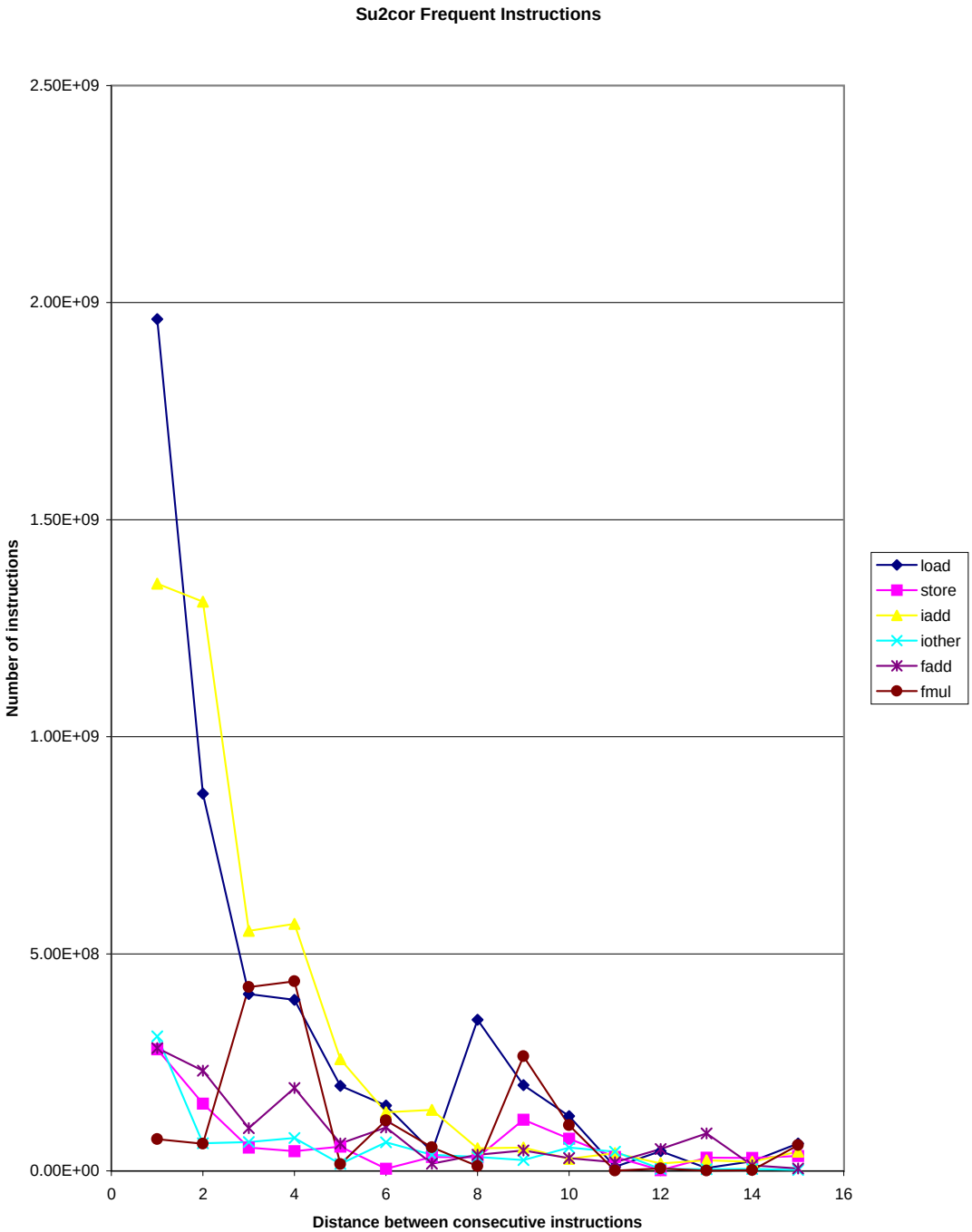
6.1 Instruction distribution for swim



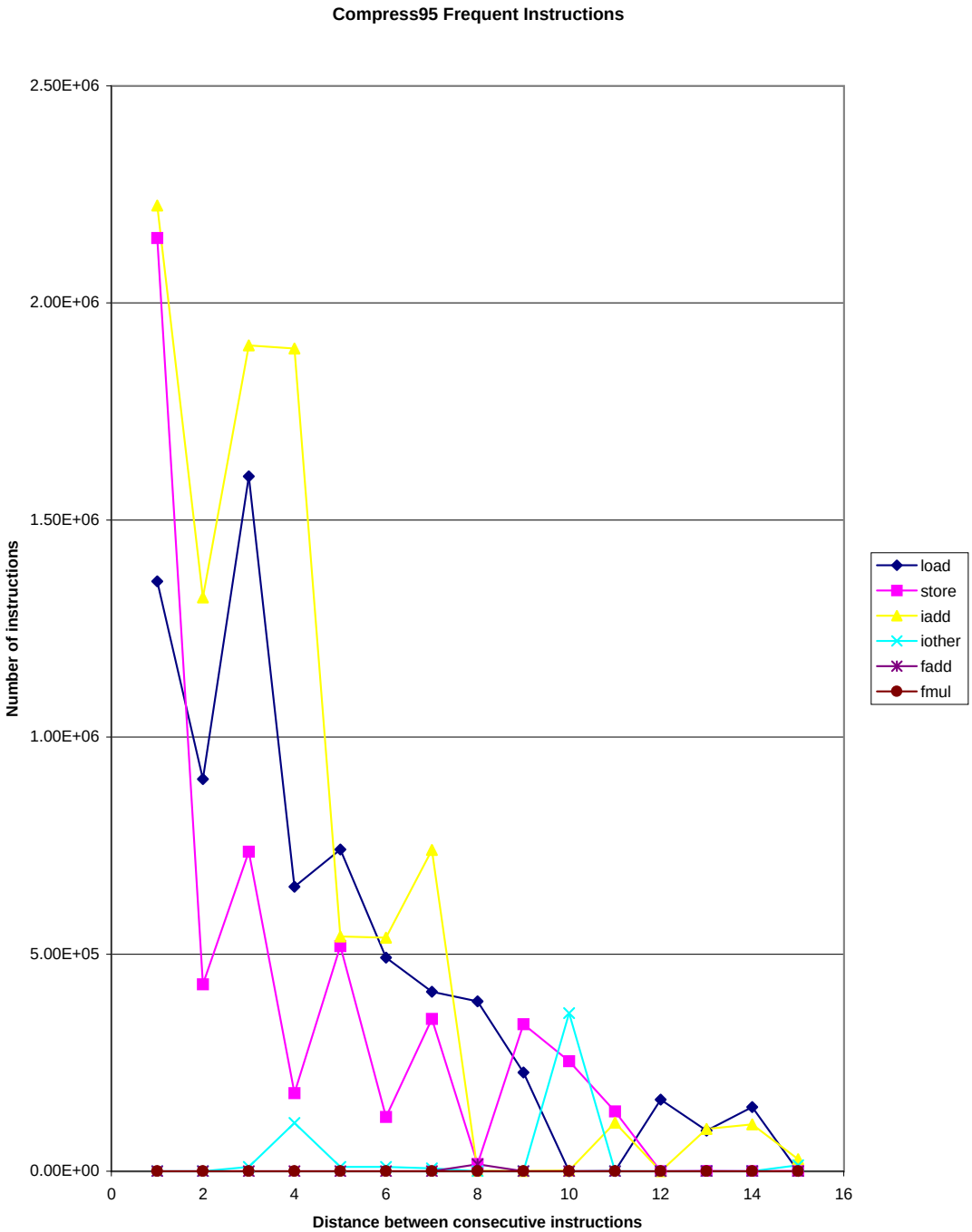
6.2 Instruction distribution for wave



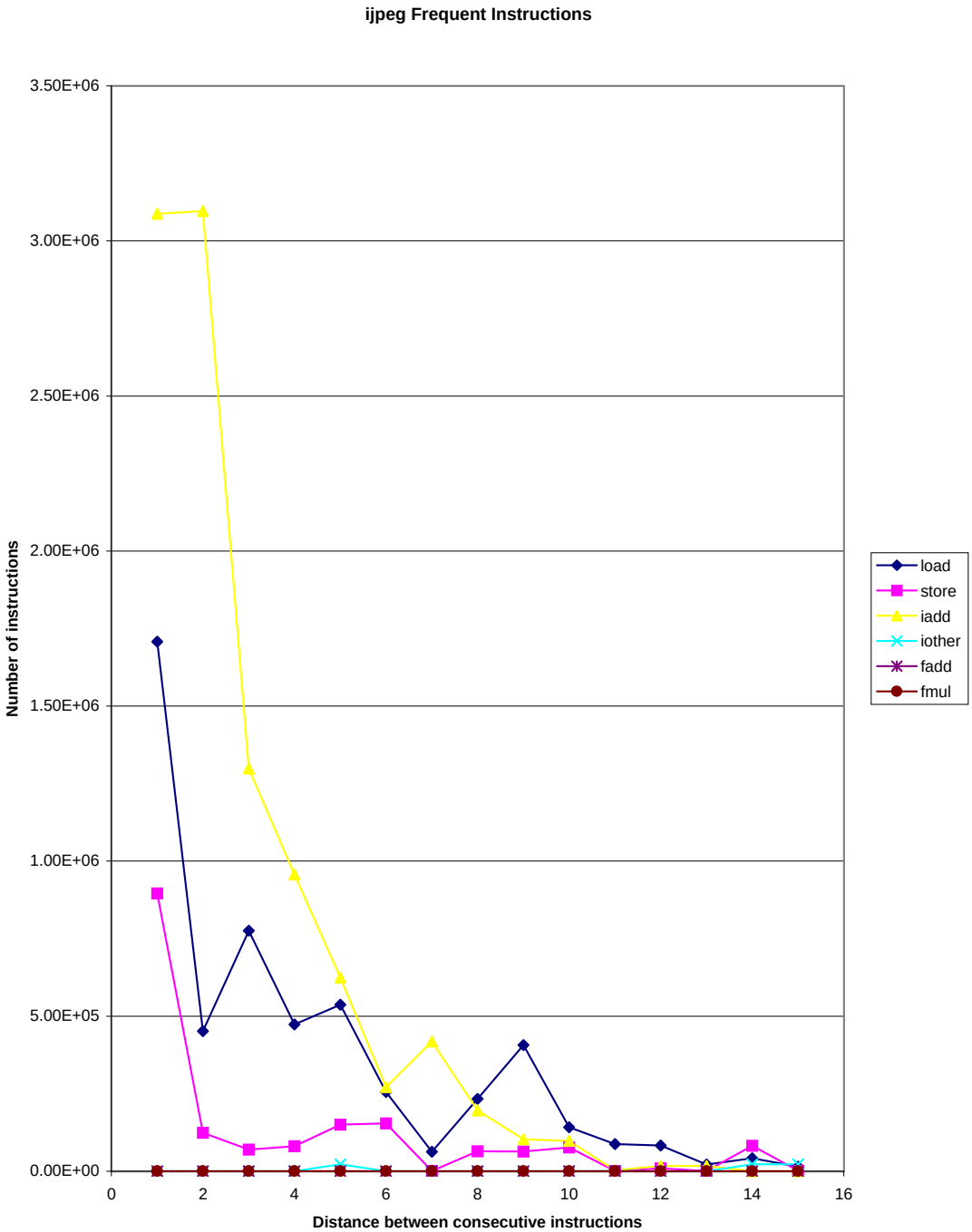
6.3 it Instruction distribution for su2cor



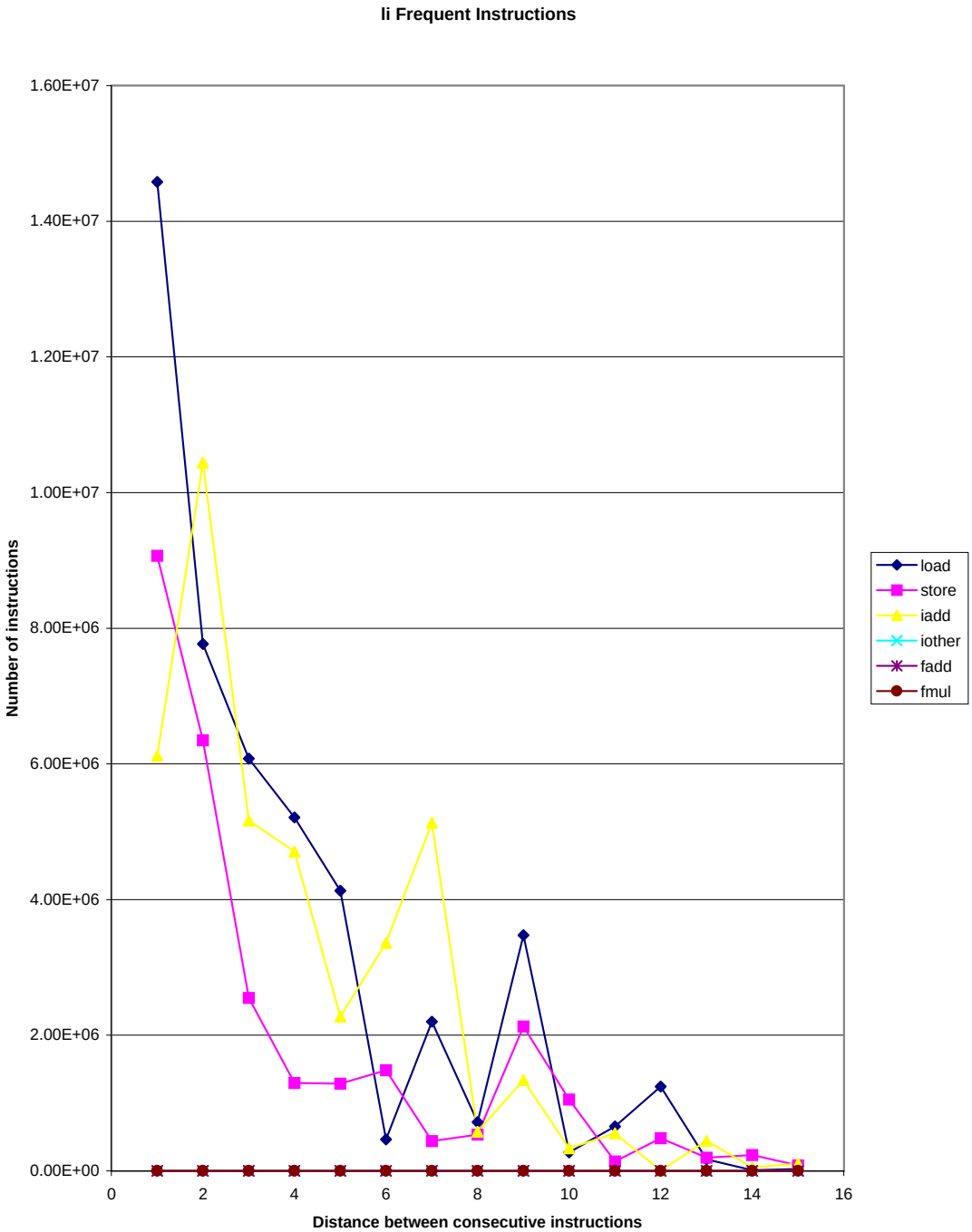
6.4 Instruction distribution for compress



6.5 Instruction distribution for ijpeg



6.6 Instruction distribution for li



6.7 Instruction distribution for kmeans