SympA'6
Besançon, 19 - 22 juin 2000
6ème Symposium sur les Architectures Nouvelles de Machines

Reconfigurable Arithmetic and Logic Unit
Unité Arithmetique et Logique Reconfigurable

D. Lavenier, K. Cameron, Y. Solihin
Los Alamos National Laboratory
Los Alamos, NM 87545, USA

Abstract

A microprocessor architecture based on a reconfigurable arithmetic and logic unit (R-ALU) capable of executing either integer or floating-point operations is presented. Simulation comparisons between a MIPS R1000-like architecture and its counterpart augmented with a R-ALU show a speed-up ranging from 8% to 14% for integer codes.

Résumé

Une architecture de micro-processeur basée sur une unité arithmétique et logique reconfigurable (R-ALU) capable d'exécuter soit des opérations entières, soit des opérations flottantes est présentée. Des comparaisons de performances à partir de simulation montrent un gain variant de 8% à 14% sur des codes entiers.

1 Introduction

Current super-scalar microprocessor architectures house several separated functional units to exploit instruction level parallelism. In these architectures, the mismatch of the instruction stream mix with the functional unit configuration can result in a significant factor of performance loss, due to under-use of certain functional units.

In [11], a PowerPC 620 study identified this instruction mismatch as the highest contributor to the loss of IPC (Instruction per Cycle) in the Spec92 benchmark, roughly ranging from 0.4 to 1.8. Figure 1 repeats this experiment on an architecture similar to the MIPS R10000 [17] [18] using the spec95 benchmark suite [1] augmented with the k-means iterative clustering algorithm used in typical image processing and computer vision applications [15]. The figure shows the loss of IPC due to unavailable functional units. Note that the IPC loss for integer applications is generally higher than that of floating-point applications. This indicates that the R10000 lacks integer execution bandwidth much more than floating-point execution bandwidth.

By providing as many copies of each functional unit as the microprocessor can issue, one can expect to provide the best IPC by avoiding any stalls due to unavailable units. However, this approach increases the die area and, more concerning, significantly increases the bypass network delay. As pointed out by Subarao et al. [16] this delay is directly proportional to the quadratic value of the number of functional units, and as technology

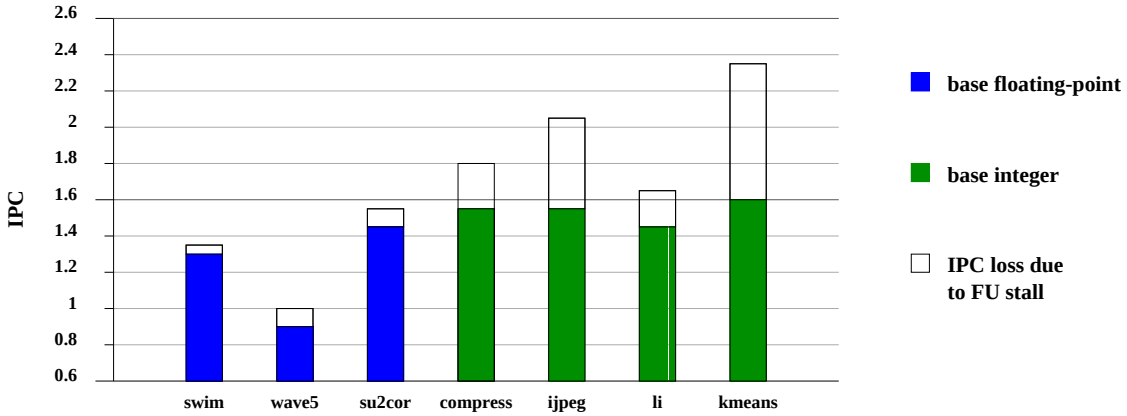

Figure 1: IPC loss due to unavailable functional units

scales down, it becomes one of the most important factors limiting the clock frequency.

Thus, if wider instruction issue is a possible approach to reduce the IPC loss, it also adversely contributes to damage the clock frequency. The solution we investigate is based on the idea that a unit can be configured on-the-fly as either a floating-point or an integer unit. In this way, we minimize the number of hardware units and we reduce the IPC loss by matching the hardware with the instruction stream.

However, this approach raises many questions: what is the penalty (in terms of number of cycles) for reconfiguring the functional unit and what is the impact on the overall performance ? How often must the unit be reconfigured ? Can a reconfigurable unit work at the same speed as other functional units ? This paper attempts to answer these questions.

The next section is an analysis of the instruction stream. It aims to determine what instructions are most interesting for an implementation of the reconfigurable unit. Section 3 details the architecture of the reconfigurable unit. Section 4 presents some performance provided by simulation. Section 5 concludes the paper.

2 Instruction Stream Analysis

In [7], an attempt to model on-chip processor performance at the instruction-level is presented. While somewhat limiting in its assumptions, this method provided qualitative conclusions regarding the loss of performance due to instruction and functional unit mismatch. A major conclusion was significant performance gain was possible using an architecture that could change its functional unit allocation dynamically. In developing the hardware logic necessary to implement such a dynamic architecture, it is necessary to minimize the subset of "important" instructions. "Important" in this context means those instructions that will have primary influence on the performance of applications of interest.

A processor services instructions. Instructions enter the processor, and are eventually committed to program-state. But the processor has a limited amount of resources available on-chip. The functional units themselves are typically hardwired allowing only a finite number and type of instructions to be executed per cycle resulting in stalls if they are overwhelmed. Furthermore, stalls resulting from this mismatch and of course memory

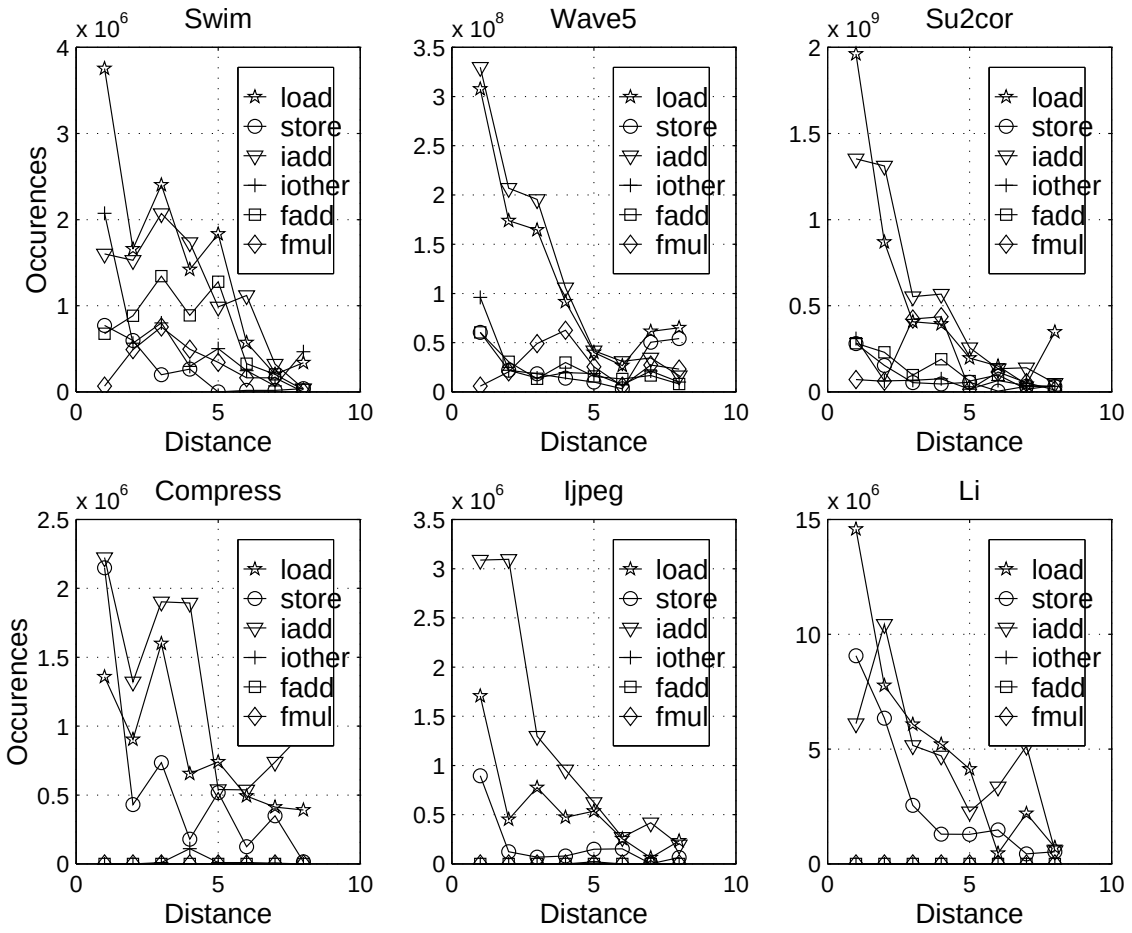

Figure 2: Profiling of the instruction streams: if an ADD instruction is encountered, the number of other instructions prior to the next occurrence of an ADD is counted. The x-axis represents the number of times distances of this length occur, and the y-axis occurrence

latency cause instructions to be backed up to the fetch/decode stage. This again results in stalls on-chip since only a finite number of instructions can be active at any one time due to limits in registers, queue sizes, etc. For any particular processor, these limitations vary.

We profile the instruction stream as follows: if we encounter an integer-add instruction, we count the number of other instructions that occur prior to the next occurrence of an integer-add instruction. We keep track of the number of times distances of this length occur and plot length on the x-axis and number of occurrences on the y-axis of our graphs. Figures 2 shows the resulting sets of most frequent instructions for all the codes of interest.

So, contemplating the qualities of a typical processor, we measure the distance between consecutive instructions of each type. In other words, we directly count the number of instructions between two identical instruction types. Why is this interesting? The frequencies within the instruction stream itself determine the number of times a certain distance between a certain type of instruction occurs, thus giving a good representation of the original application. Furthermore, assuming an architecture uses the same instruction

set and compiler, such a profiling scheme is comparable across architectural improvements to the physical limitations previously described. Lastly, this approach directly quantifies the qualities that affect this instruction and functional unit mismatch. Particularly, we want to be able to focus on instructions that exhibit high frequencies of small distances between like instructions. Such instructions will inevitably have an adverse effect on performance since static functional unit allocation will result in on-chip stalls. By providing quantitative comparisons between each instruction type measured, we can directly compare all instruction types for a particular code. Also, we can highlight the most "important" instructions for the codes measured and compare the codes themselves. Not surprisingly, the same instructions tend to be "important" across codes while magnitudes will vary.

Based on some of the principals discussed in [7], we developed a method of quantifying the frequency of instructions with respect to the limit of functional unit allocations. By augmenting the profiling capabilities of the Simplescalar tool set [13], which simulates a MIPS R10000like architecture, we are able to measure inter-arrival distances between instructions. We view the committed instruction sequence as a sequential stream of instruction types that are executed by the processor. This stream is the entity we analyze.

We profile the instruction stream as follows: if we encounter an integer-add instruction, we count the number of other instructions that occur prior to the next occurrence of an integer-add instruction. We keep track of the number of times distances of this length occur and plot length on the x-axis and number of occurrences on the y-axis of our graphs. Figures 2 shows the resulting sets of most frequent instructions for all the codes of interest.

We first utilize this technique 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 k-means. 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 listed in each of the figures.

Integer-add operations are quite common 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 these 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 for the reconfigurable unit. Particularly, the goal is to provide extended integer execution bandwidth while maintaining the power provided from reconfiguring as a floating-point unit.

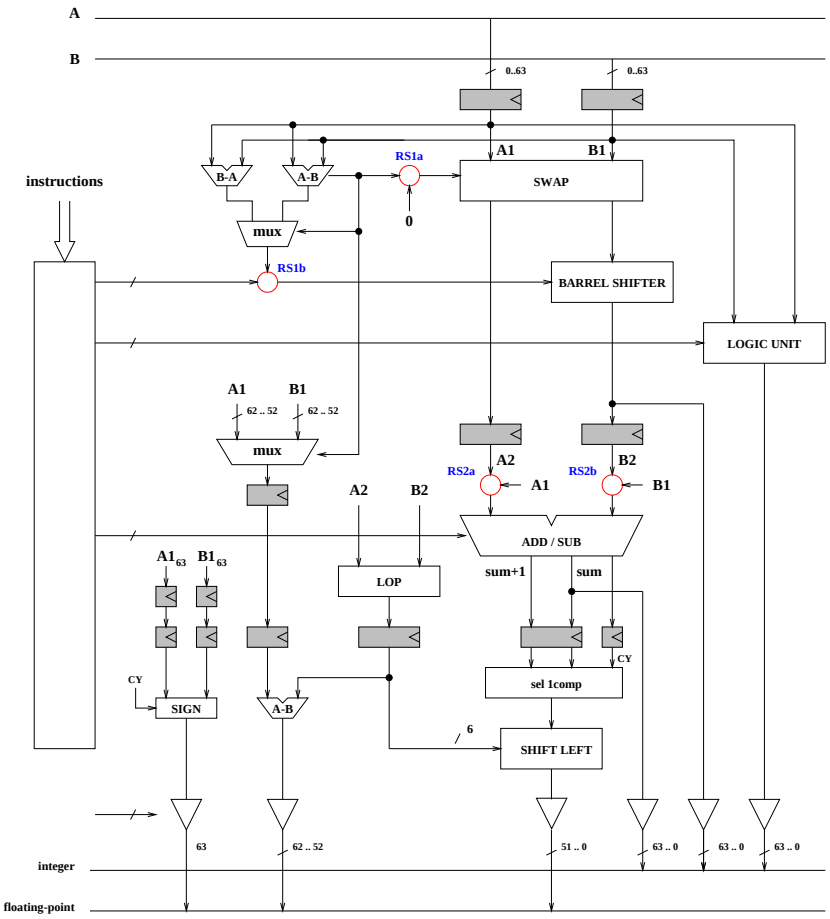

Figure 3: Reconfigurable Arithmetic and Logic Unit (R-ALU): The architecture of a floating-point adder is augmented as follows: extension of the adder from 54 to 64 bits; insertion of a 64-bit barrel shifter, a few programmable switches, and a logic unit.

3 Reconfigurable ALU

The idea of a functional unit that can serve both integer and floating-point instruction was proposed by Subromanya and Smith [8] [19]. By augmenting integer execution capability to floating-point units, they show unit speed-up of integer intensive applications by off-loading the integer instructions to the floating-point units. Their approach focuses on compiler transformation of source codes to identify instructions that can be executed by the augmented floating-point unit. This is done by adding 22 new opcodes to identify the partition of integer instructions that are to be executed on the floating-point unit.

Our approach is opposite in direction. We detect when integer instructions can be executed by the reconfigurable unit with minimal hardware cost: steering logic is added to the dispatch stage to decide to which reservation station instructions are sent. So we do not sacrificed binary compatibility, and we avoid re-compilation and additional compiler instruction analysis.

3.1 Architecture

Based on the instruction stream analysis, addition, shift and logical operations are the most important integer operations. At the same time, floating-point addition represents the major operation of scientific codes. Thus, the reconfigurable arithmetic and logic unit (R-ALU) is restricted to these operations and can be reconfigured either as:

- **an integer ALU:** in this mode, the R-ALU performs 64-bit integer operations: addition (ADD), subtraction (SUB), shift (SLL, SRL) and usual logic operations (OR, AND, XOR, ...);
- **a floating-point Adder:** in this mode, the R-ALU performs double precision standard 754-floating-point addition (FP-ADD).

The architecture of the R-ALU, as shown figure 3 is a hybrid between an integer and floating-point architecture. Basically, it follows the architecture of a floating-point adder but is augmented with the following features:

- extension of the adder from 54 to 64 bits;
- substitution of the 54-bit right shifter by a 64-bit barrel shifter;
- insertion of 4 programmable switches along the data-path;
- addition of a logic unit.

The circles in Figure 3 represent the programmable switches. The R-ALU takes two operands as input, 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.

3.2 Timing consideration

The first assertion we make is that, whatever the technology used, the 64-bit addition is the critical path. When using Carry Lookahead technique, an addition can be performed in time $O(log\ n)$, that is a time, 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 delay if we consider that the connection has been set previously, that is during the reconfiguration

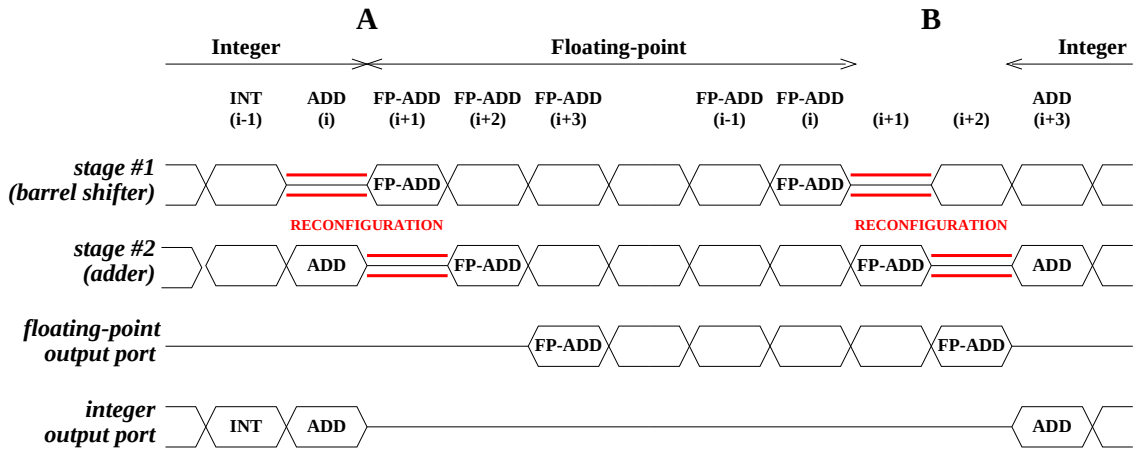

Figure 4: Switching penalty: (A) from integer to floating-point; (B) from floating-point to integer

step. Furthermore, delays induced by the programmable switches can be compensated by resizing a few MOS transistors. In [6] we show that the data-path delay is not affected if switches are carefully driven by appropriated drivers. The only effect will be a very small increase of the power consumption.

3.3 Reconfiguration issue

As the R-ALU may be asked to switch frequently between integer or floating-point ALU, we must analyze the reconfiguration penalty cost and its impact on the overall performance. Actually, due to the 3-stage pipeline when configured as a floating-point unit, and the 1-stage pipeline when configurated as an integer ALU, the time for switching from integer to floating-point, or floating-point to integer, is not constant: it depends on the last or the next operation to perform.

We present two extreme situations. The first switches from integer to floating-point and requires no extra reconfiguration cycles. The second switches from floating-point to integer and requires two extra reconfiguration cycles. An exhaustive analysis can be found in [6].

1. **Switching from Integer to Floating-point (fig. 4-A):** The last integer operation to perform before switching is an `ADD/SUB` or a `LOGICAL` operation (not a shift 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 `LOGICAL` operation. If such integer operation is executed during cycle i, the reconfiguration process can start at the begining of cycle i. **This situation does not require an extra cycle for reconfiguring the R-ALU.**

2. **Switching from Floating-point to Integer (fig. 4-B):** The first integer operation to perform after switching is an `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**.

The following table summarizes all the switching 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 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).

4 Performance Analysis

Performance analysis for evaluating the R-ALU benefit has been performed on the MIPS R10000 architecture. Basically, This architecture (Figure 5-A) comprises 2 integer ALUs, 2 floating-point units and one address generation unit embedded in the Load/Store unit (LSU). Both ALUs are capable of performing basic operations. In addition ALU1 performs branch and shift operations, and ALU2 multiplication and division operations. One floating-point unit (FPU1) is dedicated to floating-point addition, the other one (FPU2) performs multiplication, division and square-root operations. There are 3 16-entry reservation stations issuing instructions in an out-of-order manner.

Figure 5-B shows the modification we made to the MIPS R10000 architecture for inserting the R-ALU. The floating-point unit (FP1) is substituted by the R-ALU and a new 8-entry reservation unit is added. The FPU reservation station is reduced to 8 entries because now it does not need to store FP-ADD operation. Analysis of instruction is done at the dispatch stage right after the fetched instruction are decoded for operands. Then register renaming is performed in parallel with a steering logic that selects a subset of instruction to be executed by the R-ALU. Finally, the selected instructions are dispatched to the new reservation unit. If the R-ALU detects that the new instruction stream is of different type compared to the one it is currently serving, it switches accordingly. The steering logic is performed in parallel with register renaming to avoid any effect on the clock frequency.

Simulations have been performed on the same set of applications introduced in section 2, again with the Simplescalar Toolset simulator [13]. However, a modified simulator version has been done to insert the R-ALU unit, its reservation station, the steering logic, and to take into account the latency cost due to the reconfiguration of the R-ALU.

Figure 6 shows the IPC simulation results for the MIPS R10000 base architecture (first bar) and for the modified architecture (second bar). A third architecture (third bar) has also been simulated for comparison purpose: it adds to the base architecture an extra

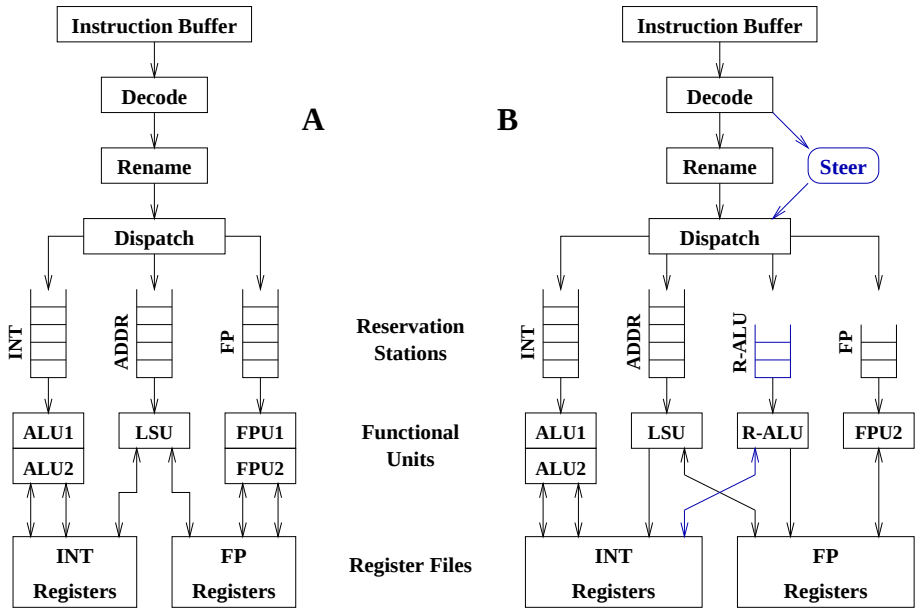

Figure 5: (A) base MIPS R10000 architecture. (B) modified architecture: the FPU1 is subsituted with the R-ALU, an 8-entry reservation station is added as weel as the steering logic

integer ALU that also calculates addresses for the memory instruction. It is provided for comparison to assess how effective the R-ALU is to a less-scalable brute-force approach of adding an extra ALU.

First, for all schemes, the IPC for floating-point applications does not change. By comparing the base architecture with the additional ALU scheme we know that floating-point operations do not need additional integer execution or address generation bandwidth. Consequently, adding a R-ALU which adds integer capability will have little impact on these codes. The IPC is either increased or decreased by a very small amount. This is due to the fact that the R-ALU does not add any additional floating-point execution bandwidth. The R-ALU is only beneficial when there is no floating-point addition, for example, during initialization phase. This extra bandwidth gives the swim and the su2cor applications a little bit of IPC improvement. However, for the wave5 application, the additional IPC is offset by the cost of the reconfiguration. The reconfiguration frequency for the wave5 application is the highest, causing high total cost penalty.

The table below shows the reconfiguration frequency for all the applications. For floating-point applications, the reconfiguration frequency translates directly into IPC gain: the more frequent the reconfiguration occurs, the lower the IPC gain.

Application	swim	wave5	su2cor	compress	ijpeg	li	k-means
Average nb. instructions per reconfiguration	40.5	16.5	21.1	370	7219576	7065704	325

The reconfiguration frequency for the ijpeg and li applications is very low because the

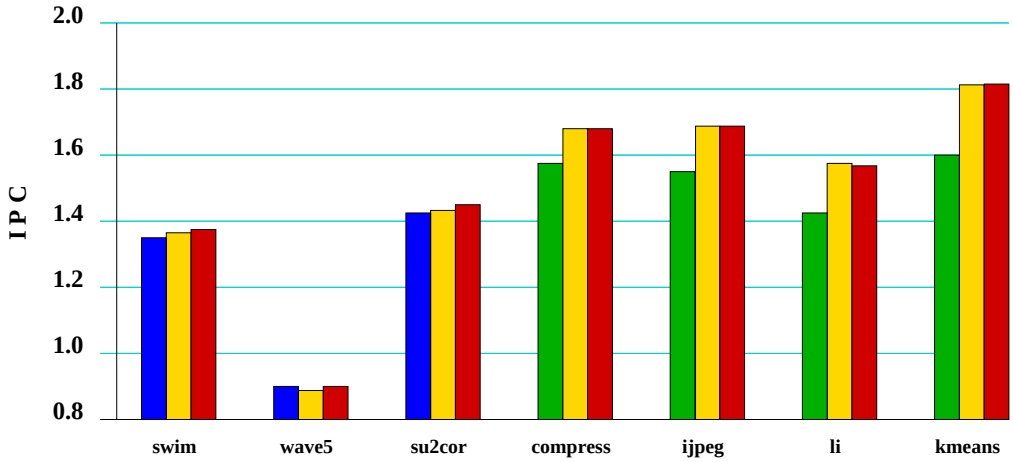

Figure 6: IPC gain: base MIPS R10000 architecture (first bar), modified R-ALU architecture (second bar), extra integer ALU architecture (third bar).

code contains no floating-point addition. The k-means and compress applications, however, respectively have 0.7% and 0.5% floating-point addition instructions, leading to a higher reconfiguration frequency compared to ijpeg and li.

If we now consider integer applications, improvement ranges from 8.3% for the compress application to 14.4% for the k-means application. The steering of integer and memory instructions explains this performance improvement: the R-ALU provides more integer execution bandwidth most of the time. Note that the memory instructions that are sent to the R-ALU are also sent to the address reservation station because the R-ALU only performs address generation and passes the results to the corresponding instruction in the address reservation station which actually holds the memory instruction until the actual load or store operations have been completed and committed. Hence, memory instructions enter the R-ALU at the dispatch stage, and leave the R-ALU at the issue stage.

5 Conclusion

We have presented an architecture that exploits extra bandwidth provided by a reconfigurable ALU having its own reservation station. The performance gain is practically equivalent to adding an additional ALU into a MIPS R10000-like architecture. We have shown from simulation that this architecture can speed-up integer application ranging from 8.3% to 14.3%. For floating-point applications, the architecture has no significant impact on the performance. Because we add a separate 8-entry reservation station while reducing the floating-point reservation station to 8 entries, the only extra hardware cost is due to extra integer register ports, steering logic, and integer extension capability of a floating-point adder. We estimate the cost to be less than 1% of the total die area of the micro-processor.

Actually, with the steadily increasing integration density, minimizing the hardware (in terms of number of transistors) may not be the principal objective. The clock speed is a more critical aspect. Adding an extra integer ALU somehow leads to a simpler architecture which, as we have shown, gives similar IPC improvement. However, a great advantage of

our solution is that we do not extend the bypass network which will be a more and more critical section of future micro-processors as the MOS technology will continue to scale down.

Although R-ALU is applicable to superscalar microprocessors, it is most applicable to some future architectures. These include Processor In Memory (PIM), trace processors, and VLIW architectures. In terms of performance, R-ALU always helps when the instruction fetch bandwidth of a processor exceeds its instruction execution bandwidth, as demonstrated in [5]. However, for the future architectures mentioned, R-ALU provides more than just performance boost.

In PIM systems [3], there are many processors in a memory chip. DRAM manufacturers are concerned on the die area occupied by logic compared to DRAM banks. Thus, minimizing logic area is of the most importance. The use of R-ALU can help reducing the area occupied by each processor while retaining performance.

In trace processor (e.g. Hal Sparc64 [2]), instructions are bound to ports and units at trace construction stages. Mismatch between trace and functional units requires trace to be broken. R-ALU units provide more flexibility in trace construction, thus denser traces. Furthermore, since data dependences are analyzed at trace construction, integer instructions that are to be sent to the R-ALU should use floating-point registers only, eliminating extra wiring between the two register files. Similar benefits of flexible instruction bundling without extra wiring between register files can be obtained for a VLIW architecture.

Bibliographie

1. Standard Performance Evaluation Corporation. http://www.spec.org.
2. K. Diefendorff, Hal Makes Sparc Fly: Sparc64 Employs Trace Cache and Superspeculation for High ILP, *Microprocessor Report* 13(15), 1999.
3. Yi Kang et. al., FlexRAM: Toward an Advanced Intelligent Memory System, *International Conference on Computer Design*, 1999.
4. Y. Solihin, K.W. Cameron, Y. Luo, D. Lavenier, M.Gokhale, Reservation Station Architecture of Mutable Functional Unit Usage in Superscalar Processors, *Los Alamos Unclassified Report 99-6234*, LANL, 1999
5. Y. Solihin, K. W. Cameron, Y. Luo, D. Lavenier, M. Gokhale, Boosting the Speedup of Future Processor Architectures by Using Mutable Funcational Unit, *Los Alamos Unclassified Report 99-6768*, LANL, 1999.
6. D. Lavenier, Y. Solihin, K. Cameron, Integer/Floating-point Reconfigurable ALU, *Los Alamos Unclassified Report 99-5535*, LANL, 1999.
7. 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*, 1999.
8. S. Subramanya Sastry et. al. Exploiting idle floating-point resources for integer execution. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.
9. 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, 1998.
10. 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.
11. J. L. Hennessy, D. A. Patterson, Computer Architecture, A Quantitative Approach, *Prentice Hall*, 1998.
12. J. L. Hennessy, D. A. Patterson, Computer Organization and Design, The Hardware/Software Interface, *Morgan, Kaufmann*, 1998.
13. D. Burger, T.M. Austin, The Simplescalar Toolset, Version 2.0, *Technical Report 1342*, University of Wisconsin-Madison Computer science Departement, 1997.
14. M. J. S. Smith, Application-Specific Integrated Circuit, *Addison Wesley*, VLSI Systems series, 1997.
15. J. Theiler, G. Gisler. A contiguity-enhanced k-means clustering algorithm for unsupervised multispectral image segmentation, *Proc. SPIE 3159*, 1997.
16. S. Palacharla, J.E. Smith, Complexity-effective superscalar processor, *ISCA*, 1997.
17. *MIPS R10000 Microprocessor User's Manual*, MIPS, 1996.
18. A. Seznec, F. Lloansi, Etude des Architectures des Microprocesseurs MIPS R10000, UltraSparc et Pentium Pro, *Rapport de recherche IRISA*, no 1024, 1996.
19. S. Palacharla, J.E. Smith, Decoupling Integer Execution in superscalar processors, *Proceedings of the 28th Annual International Dymposium on Microarchitecture*, 1995.
20. A. Ahi et al., R10000 Superscalar Microprocessor, *Hotchip'95*, 1995.
21. N. Quach, J. Flynn, Design and implementation of the SNAP floating-point adder, *technical report CSL-TR-91-501*, Stanford University, 1991.
22. N. Quach, J. Flynn, Leading One Prediction: implementation, generalization, and application, *technical report CSL-TR-91-463*, Stanford University, 1991.