# A Configurable Multi-Ported Register File Architecture for Soft Processor Cores

Mazen A. R. Saghir and Rawan Naous

Department of Electrical and Computer Engineering
American University of Beirut
P.O. Box 11-0236 Riad El-Solh, Beirut 1107 2020, Lebanon
{mazen,rsn11}@aub.edu.lb

**Abstract.** This paper describes the architecture of a configurable, multi-ported register file for soft processor cores. The register file is designed using the low-latency block RAMs found in high-density FPGAs like the Xilinx Virtex-4. The latency of the register file and its utilization of FPGA resources are evaluated with respect to design parameters that include word length, register file size, and number of read and write ports. Experimental results demonstrate the flexibility, performance, and area efficiency of our proposed register file architecture.

## 1 Introduction

Growing interest in configurable and reconfigurable computing has increased the proportion of field programmable gate arrays (FPGAs) being used to implement computational platforms. High logic densities and rich on-chip features, such as hardware multipliers, memory blocks, DSP blocks, peripheral controllers, and microprocessor cores, make it possible to quickly implement highly tuned system designs that couple software programmable processor cores with custom hardware blocks that can be implemented inside the logic fabric of an FPGA.

The past few years have also witnessed the emergence of *soft* processor cores [1, 2]. These processors are specified in a hardware description language, such as VHDL or Verilog, and implemented in the logic fabric of an FPGA. The main advantages of soft processors are that their datapaths can be easily reconfigured and their instruction set architectures extended to support user-defined machine instructions. To overcome the performance and area inefficiencies of implementing soft processors in FPGAs, the microarchitectures of these processors are typically optimized to make use of available on-chip resources [3]. For example, a common optimization is to use embedded memory blocks to implement register files [4]. Given the simplicity of current soft processor architectures, these register files typically feature two read ports and one write port. However, as the complexity of soft processor cores increases to support wider instruction issue [5, 12], multithreading [6], and customized datapaths [7] flexible register file architectures capable of supporting multiple read and write ports are needed.

In this paper we describe the architecture of a configurable, multi-ported, register file for soft processor cores. We also evaluate the performance of our

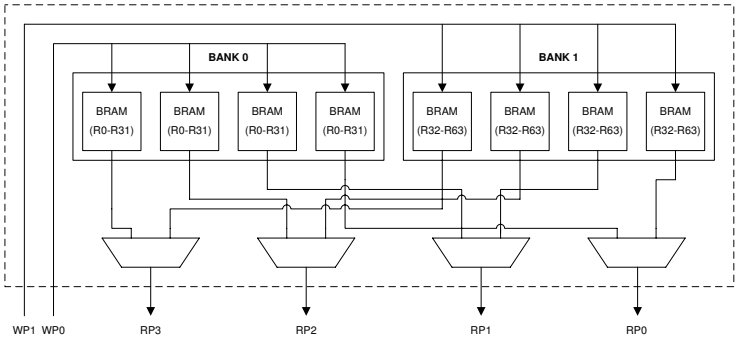


**Fig. 1.** A register file with four read and two write ports

design and measure its utilization of FPGA resources. In Sect. 2, we describe the architecture of our register file and highlight the characteristics and limitations of its design. Then, in Sect. 3 we describe how we evaluate the performance and area of the register file. Next, in Sect. 4, we analyze the effects of various register file design parameters such as data word length, register size, and number of read/write ports on both the latency and area of the register file. Then, in Sect. 5, we show how our register file can be used to customize the datapath of a soft VLIW processor and how this affects the overall performance and area of the processor. In Sect. 6, we describe related work and compare it to our own. Finally, in Sect. 7, we provide our conclusions and describe future work.

## 2 Multi-Ported Register File Architecture

Our register file is designed around the embedded block RAMs (BRAMs) found in contemporary, high-density, FPGAs such as the Xilinx Virtex-4 [8]. The BRAMs on the Virtex-4 are dual-ported, synchronous, memory blocks capable of storing 16,384 data and 2,048 parity bits that can be organized in various aspect ratios. The two ports are independent and can each be configured as either a read or a write port. In our design, each BRAM is configured as a $512 \times 36$-bit memory block with one read port and one write port, and this configuration forms the basic building block in our register file. Figure 1 shows how eight such BRAMs can be used to implement a $64 \times 32$-bit register file with four read and two write ports.

The key to supporting multiple register ports is to organize the BRAMs into register banks and duplicate data across the various BRAMs within each bank. The register banks are each used to store a *subset* of the register file. In Fig. 1, the BRAMs are divided into two register banks used to store registers `R0` to `R31` and `R32` to `R63`, respectively. Since register banks hold mutually exclusive sets

of registers, they can be updated independently. Associating each register bank with a separate write port provides support for multiple write ports.

Within each register bank, multiple BRAMs are used to store *duplicate* copies of the corresponding register subset. By duplicating register values across BRAMs, the values of different registers can be read from different BRAMs simultaneously, thus providing support for multiple read ports. Since the register file in Fig. 1 has four read ports, each register bank contains four BRAMs that store duplicate copies of the corresponding register subset. Since each register bank stores a different subset of the register file, multiplexers are used to provide access to the registers stored in each register bank. In Fig. 1, four multiplexers – one for each read port – are used to provide access to all the registers in both register banks. Finally, to maintain data consistency among the BRAMs in a register bank, each BRAM is connected to the corresponding register write port making it possible to update all BRAMs simultaneously.

### 2.1 Design Characteristics and Limitations

A major characteristic of our register file is that it is fully configurable and can be designed with an *arbitrary* word length, register size, and number of read or write ports. Here it is worth noting that the number of register ports does not have to be even or a power of two. This enables customizing the register file at a very fine level to the specific needs of the target processor microarchitecture. However, the size of the register subset associated with a register bank must be a power of two, and this causes some register file configurations to have register banks with different sized register subsets. For example, a register file of size 64 with five read ports and three write ports will consist of three register banks, two of which containing 16 registers and one containing 32 registers. However, such an asymmetric distribution of register subsets does not affect the performance, area, or functionality of the register file.

Another characteristic of our register file is that it influences the way registers and instructions get allocated and scheduled, respectively. Since registers are distributed across several banks associated with different write ports, registers must be allocated, and instructions scheduled, in a manner that avoids contention for the write ports. Thus, instructions cannot be scheduled to execute in parallel if they produce results in registers that belong to the same register bank. Although various software and hardware techniques, such as register renaming, can be applied to solving this problem [9, 10], these techniques and their effects are beyond the scope of this paper.

Finally, our register file is limited by the number and size of the BRAMs available within the FPGA. If $WL$ and $DW$ are the word length and data width of the register file and BRAMs, respectively, implementing a register file with $M$ read ports and $N$ write ports requires $\lceil WL/DW \rceil \times M \times N$ BRAMs distributed across $N$ banks. Since BRAMs typically have a fixed size and can only be configured in a limited number of aspect ratios, there is a limit on the number of registers that can be stored within each BRAM. In the Xilinx Virtex-4 FPGA, the BRAM aspect ratio that corresponds to the maximum word length

is 512 × 36 bits. This limits the size of the register subset that can be stored in a BRAM to 512 registers. Although this limit can be easily extended by using more BRAMs in each register bank, we do not consider this case in this paper.

## 3 Performance and Area Evaluation

To evaluate the performance and area of our register file architecture, we developed a parameterizable register file generator called MPRFGen. MPRFGen reads design parameters such as word length, register size, and number of read and write ports and generates the corresponding VHDL code, which instantiates the appropriate number of BRAMs and organizes them as a multi-ported register file. Once the VHDL code is generated, we use the Xilinx ISE 8.2.03i tools, targeting a Xilinx Virtex-4 LX (XC4VLX160-12FF1148) FPGA, to synthesize and implement the register file. We chose the XC4VLX160 as our target FPGA because of its high logic density, which is representative of the types of FPGAs used for implementing computational platforms designed around soft processor cores, and because it contains 288 BRAMs, which is large enough to implement and evaluate a wide range of register file organizations. Finally, we used the Xilinx Timing Analyzer tool to measure the latency of the register file and the post-place-and-route synthesis reports to measure its utilization of FPGA resources. Although somewhat crude, the number of BRAMs and FPGA slices used to implement a register file is still an accurate measure of its area.

### 3.1 Register File Latency

The latency of the register file is determined by its critical path delay as reported by the timing analyzer tool. The critical path delay can be generally expressed by the following equation:

$$\textit{Critical Path Delay} = T_{\text{bram}} + T_{\text{routing}} + T_{\text{mux}} \ . \tag{1}$$

Here, $T_{\text{bram}}$ is the latency of a BRAM block, $T_{\text{routing}}$ is the routing delay along the critical path between the corresponding register bank and output multiplexer, and $T_{\text{mux}}$ is the delay of the output multiplexer. Typically, $T_{\text{bram}}$ is a constant that depends on the fabrication process technology and speed grade of the FPGA. For the Xilinx XC4VLX160-12FF1148, $T_{\text{bram}} = 1.647$ ns. On the other hand, $T_{\text{routing}}$ and $T_{\text{mux}}$ depend on the organization of the register file and vary with word length and the number of read and write ports. Both delays also depend on the architecture of the FPGA routing network and the efficiency of the placement and routing tool. In Sect. 4, we examine how various design parameters affect the latency and FPGA resource utilization of our register file architecture.

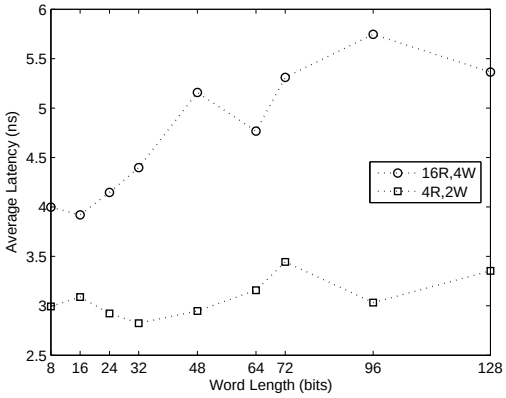


**Fig. 2.** Latency vs. word length

## 4 Results and Analysis

In this section we present the results of three experiments we conducted to assess the impact of word length, register file size, and the number of read and write ports on both the latency and FPGA resource utilization of our register file architecture. To minimize the effect of the place-and-route tool on latency measurements, we report the *average latency* for each register file organization, which we compute based on *nine* latency measurements corresponding to different starting placer cost tables [11]. We also report FPGA resource utilization in terms of the number of BRAMs and slices used to implement each register file.

### 4.1 Effect of Word Length

Our first experiment examined the impact of register word length on the latency and area of two register files configured with 16 read and 4 write ports and 4 read and 2 write ports, respectively. Figure 2 shows the latencies of both register files as the word length is increased from 8 to 128 bits. Our results show that latency tends to increase with word length since the amount of routing used to connect register banks to output multiplexers also increases with word length. The additional routing delays cause $T_{\text{routing}}$ to increase, and this causes overall latency to increase. However, our results also suggest that the latency is sensitive to variations in placement and routing. In Fig. 2, the latencies associated with some word lengths are actually *lower* than those associated with smaller word lengths. Finally, our results show that the impact of word length becomes more pronounced as the number of register ports increases. This is due to the corresponding increase in routing resources needed to connect register banks to output multiplexers, which is also proportional to the product of the number of read and write ports.

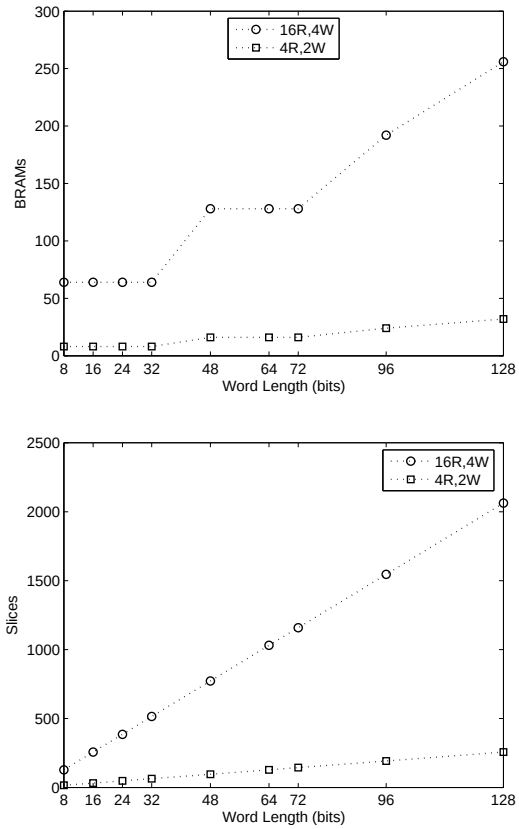


**Fig. 3.** Resource utilization vs. word length

Figure 3 shows the utilization of FPGA resource as a function of word length for both register files. In Sect. 3 we saw that the number of BRAMs used to implement a register file depends on the word length, the data width of the BRAM, and the number of read and write ports. Figure 3 shows that, for each register file, the number of BRAMs increases in steps that track the ratio of the register word length to BRAM data width. Figure 3 also shows that the number of FPGA slices used to implement the register file is a linear function of its word length. This is due to the fact that slices are only used to implement output multiplexers and that the number of slices needed to implement a register file is proportional to the product of word length by the number of read ports.

### 4.2 Effect of Register File Size

Our second experiment examined the impact of register file size. For this experiment we used three register files that were each configured with 16 read ports and

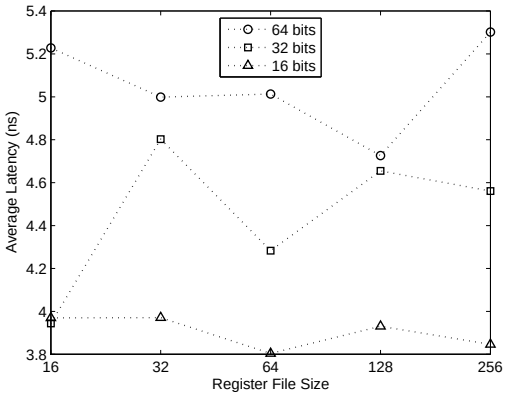


**Fig. 4.** Latency vs. register file size

4 write ports, and that had word lengths of 16-, 32-, and 64-bits, respectively. Figure 4 shows the average latencies of the three register files as we increased the size of the register file from 16 to 256.

Our results show that there is little correlation between the size of a register file and its latency. As the size of the register file increases, additional addressing lines are needed for each of its read and write ports, and the number of registers allocated to each bank increases. Routing the additional addressing lines can introduce some variation in the latency of different sized register files, but these variations are mainly due to the effects of placement and routing. Moreover, the increase in the number of registers per bank is typically absorbed by the BRAMs, which can each store up to 512 registers. As a result, even if the size of the register file increases, its *organization* – the number of its BRAMs, register banks, output multiplexers, and the data paths used to connect them – does not change. This leads us to conclude that the latency of a register file is independent of its size, and that the variations in the latencies of different sized register files that otherwise have the *same* organization are mainly due to the effects of placement and routing. Of course, if more than 512 registers per bank are needed, additional BRAMs will be required, and this will effect both the organization and the latency of the register file. However, since we have assumed that this limit will not be exceeded, we do not consider this case any further. Finally, since the organization of a register file does not change with its size, it follows that the area occupied by the register file also does not change. Table 1 shows the utilization of FPGA resources for each of the register files used in this experiment. As we increase the size of each register file, its utilization of FPGA resources remains unchanged.

**Table 1.** FPGA resource utilization for a register file with 16 read and 4 write ports

| Word Length | BRAMs | Slices |
|---|---|---|
| 16-bits | 64 | 257 |
| 32-bits | 64 | 515 |
| 64-bits | 128 | 1,031 |

### 4.3 Effect of Read/Write Ports

Our third experiment examined the impact of the number of register read and write ports. For this experiment we used a 256 × 32-bit register file and varied the number of its read ports from 2 to 128 and the number of its write ports from 1 to 64. Since our target FPGA, the Xilinx XC4VLX160, contains 288 BRAMs, we only considered register file configurations whose BRAM requirements could be accommodated by the FPGA.

Figure 5 shows the average latency of the register file as a function of its read and write ports[1]. As expected, our results show that latency increases with the number of read and write ports. This reflects the architecture of our register file where the *width* of the output multiplexers – the number of inputs to select from – depends on the number of register banks, which, in turn, depends on the number of write ports. Since output multiplexers are implemented as a hierarchy of lookup tables and two-input multiplexers, the depth of the hierarchy and the corresponding delay, $T_{\text{mux}}$, increase as the width of the output multiplexers increases. Moreover, as the number of register banks increases, more routing resources are used to connect the register banks to the output multiplexers, causing $T_{\text{routing}}$ to also increase. Similarly, increasing the number of read ports increases the *number* of output multiplexers and the amount of routing needed to connect the register banks to the output multiplexers, and this also increases $T_{\text{routing}}$. The latency of a register file therefore increases in proportion to the product of the number of read and write ports.

Figure 6 shows the utilization of FPGA resources as a function of read and write ports. In Sect. 3 we saw that BRAM utilization is proportional to the product of the number of read and write ports. Similarly, the utilization of FPGA slices depends on the width of the output multiplexer, which determines the number of slices needed to implement the multiplexer. The number of slices used to implement the register file is proportional to the product of the number of read and write ports.

Having gained some insight into the effects of various register file design parameters on latency and area, we next examine how our register file can be used to customize the datapath of a soft VLIW processor, and we assess its impact on the overall performance and area of the processor.

[1] Write ports are labeled WP on the graphs.

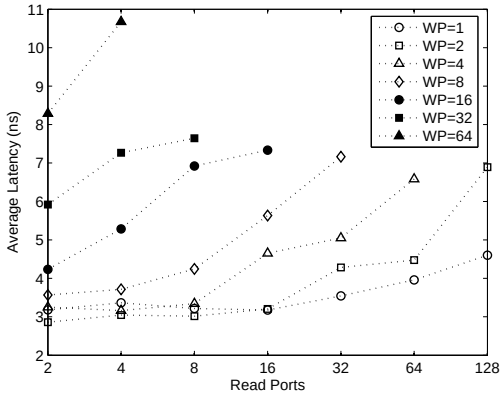


**Fig. 5.** Latency vs. number of read and write ports

## 5 Soft Processor Datapath Customization Using a Multi-Ported Register File

To demonstrate the flexibility of our multi-ported register file and assess its impact on the overall performance and area of a high-performance soft processor core, we used two instances of our register file to customize the datapath of a configurable soft VLIW processor based on [12]. The datapath uses separate register files, DRF and ARF, for storing data and memory addresses, respectively. As the number of functional units in the datapath is increased, more register ports are generally needed to provide access to operands and results. For this experiment, we used our register file architecture to implement the DRF for two configurations of the processor datapath, DP1 and DP2. We configured the DRF of the first processor datapath (DRF1) as a 32 $\times$ 32-bit register file with 8 read ports and 2 write ports. We then configured the DRF of the second processor datapath (DRF2) as a 64 $\times$ 32-bit register file with 14 read and 4 write ports.

Tables 2 and 3 show the performance and resource utilization of the two datapaths and register files, respectively. For each processor, we show the maximum number of instructions executable per clock cycle (IPC), the processor clock frequency and corresponding period, and the number of FPGA slices, BRAMs, and hardware multipliers used to implement the datapath. For each register file we show the corresponding latency and the number of FPGA slices and BRAMs used to implement the register file. Our results show that DRF1 uses 5.5% of the slices and 69.6% of the BRAMs used to implement DP1, and that its latency corresponds to 38.9% of the processor clock period. Our results also show that DRF2 uses 9.0% of the slices and 86.2% of the BRAMs used to implement DP2, and that its latency corresponds to 43.1% of the processor clock period. While the high BRAM utilization is expected – BRAMs are, after all, only used

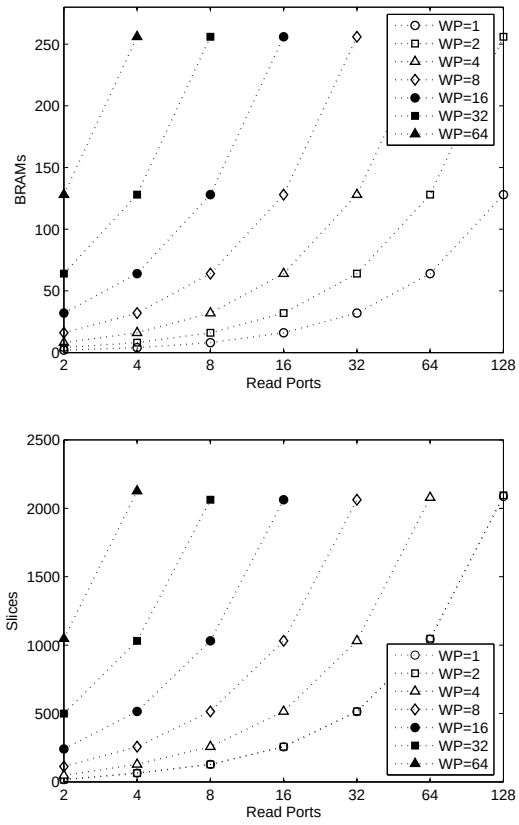


**Fig. 6.** Resource utilization vs. number of read and write ports

to implement register files and data memories – these results demonstrate the flexibility, performance, and area efficiency of our multi-ported register file.

## 6 Related Work

The use of embedded BRAMs to implement register files is a well known microarchitectural optimization for the efficient implementation of soft processor cores [4]. As soft processor architectures evolve to include wide instruction issue [5, 12], multithreading [6], and customized datapaths [7], multi-ported register files become necessary. However, while most of the register files used in these processors provide multiple read ports, none, to the best of our knowledge, provides efficient support for multiple write ports.

Due to their central role in exploiting high levels of instruction parallelism, the design of efficient multi-ported register files for high-performance, general-

**Table 2.** Performance and FPGA resource utilization of DP1 and DP2

| Datapath | IPC | Clock | | Slices | BRAMs | MULT18×18 |
|---|---|---|---|---|---|---|
| | | Frequency | Period | | | |
| DP1 | 5 | 132.4 MHz | 7.55 ns | 2,347 | 23 | 6 |
| DP2 | 8 | 119.3 MHz | 8.38 ns | 5,152 | 65 | 12 |

**Table 3.** Performance and FPGA resource utilization of DRF1 and DRF2

| Register File | Word Length | Register Size | Read Ports | Write Ports | Latency | Slices | BRAMs |
|---|---|---|---|---|---|---|---|
| DRF1 | 32 bits | 32 | 8 | 2 | 2.94 ns | 128 | 16 |
| DRF2 | 32 bits | 64 | 14 | 4 | 3.61 ns | 465 | 56 |

purpose processors has been the subject of much research [13]. Most techniques focus on partitioning the register file into multiple banks with few ports each to improve access time and reduce area and power consumption. [14–16]. We also partition our register file into multiple banks, and benefit from BRAMs with latencies that are much smaller than the typical clock period of a soft processor. In [17], the register file is partitioned to reduce the number of ports in each partition, and data is duplicated to ensure data consistency. However, an additional clock cycle is needed to update the other partition when one of the partitions is updated. In our implementation we avoid this problem by connecting all BRAMs that contain duplicate information to the same write port, thus ensuring all BRAMs are updated simultaneously.

## 7 Conclusions and Future Work

In this paper we described and evaluated the architecture of a configurable, multi-ported register file for soft processor cores. Our register file is designed around the low-latency embedded memory blocks found in contemporary high-denisty FPGAs such as the Xilinx Virtex-4 FPGA. Our results show that both the latency of the register file and its utilization of FPGA resources are proportional to the product of its word length and the number of its read and write ports. The latency of the register file also depends on the latency of individual BRAM blocks. When integrated into the datapath of a configurable VLIW soft processor, our register file architecture provided the flexibility to support processor datapath customization while exhibiting low latencies and occupying a small proportion of the processor datapath.

To continue this work, we are developing register file latency and area models that we will use in a soft VLIW processor architecture exploration tool. We are also assessing the power consumption characteristics of our register file with the aim of also developing a power model for our register file architecture. Finally,

we are developing new register allocation and instruction scheduling passes for our port of the GCC compiler to handle the proposed register file architecture.

## References


1. *MicroBlaze Processor Reference Guide*. `http://www.xilinx.com`.
2. *NIOS-II Processor Reference Handbook*. `http://www.altera.com`.
3. P. Yiannacouras, J. G. Steffan, and J. Rose, "Application-Specific Customization of Soft Processor Microarchitecture", *Proceedings of the 14th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2006)*, pp. 201–210, ACM, Feb. 22-24, 2006.
4. Xilinx Corportaion, "Using Block RAM in Spartan-3 Generation FPGAs", *Xilinx Application Note XAPP463 (v2.0)*, March 1, 2005.
5. A. Jones et al., "An FPGA-based VLIW Processor with Custom Hardware Execution", *Proceedings of the 13th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2005)*, pp. 107-117, ACM, Feb. 20-22, 2005.
6. R. Dimond, O. Mencer, and W. Luk, "CUSTARD - A Customizable Threaded FPGA Soft Processor and Tools", *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL 2005)*, pp. 1–6, IEEE, Aug. 24-26, 2005.
7. D. Jain et al. "Automatically Customizing VLIW Architectures with Coarse-Grained Application-Specific Functional Units", *Proceedings of the Eighth International Workshop on Software and Compilers for Embedded Systems (SCOPES 2004)*, pp. 17-32, Springer Berlin/Heidelberg, Sept. 2-3, 2004.
8. *Virtex-4 User Guide*, UG070 (v1.6), `http://www.xilinx.com`, Oct. 6, 2006.
9. Steven Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers/Elsevier, 1997.
10. John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Third Edition, Morgan Kaufmann Publishers/Elsevier, 2003.
11. *Development System Reference Guide (8.2i)*, `http://www.xilinx.com`.
12. M. A. R. Saghir, M. El-Majzoub, and P. Akl, "Customizing the Datapath and ISA of Soft VLIW Processors", *Proceedings of the 2007 International Conference on High Performance Embedded Architectures and Compilers (HiPEAC 2007)*, pp. 276–290, Springer LNCS 4367, Jan. 28-30, 2007.
13. K. Farkas, N. Jouppi, and P. Chow, "Register File Design Considerations in Dynamically Scheduled Processors", *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, pp. 40–51, IEEE, 1996.
14. J. L. Cruz et al., "Multiple-Banked Register File Architectures", *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA 2000)*, pp. 316–325, ACM, 2000.
15. J. Zalema et al., "Two-Level Hierarchical Register File Organization for VLIW Processors", *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 2000)*, pp. 137–146, ACM/IEEE, 2000.
16. J. H. Tseng and K. Asanovic, "Banked Multiported Register Files for High-Frequency Superscalar Microprocessors", *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA 2003)*, pp. 62–71, ACM, 2003.
17. R. E. Kessler, E. J. McLellan, and D. A. Webb, "The Alpha 21264 Microprocessor Architecture", *Proceedings of the International Conference on Computer Design (ICCD'98)*, pp. 90–95, IEEE, 1998.