A Multiported Register File with Register Renaming for Configurable Softcore VLIW Processors

Fakhar Anjam, Stephan Wong, and Faisal Nadeem
Computer Engineering Laboratory
Delft University of Technology, Delft, The Netherlands
E-mail: {F.Anjam, J.S.S.M.Wong, M.F.Nadeem}@tudelft.nl

Abstract—**In this paper, we present the design and implementation of a BRAM-based multiported register file with arbitrary number of read and write ports. In order to avoid the conflicts associated with write ports, we present a register renaming technique that is applied between the compiler and the assembler. This technique enables the utilization of a banked-BRAM register file as a true multiported register file. The advantage is that we do not need to modify the compiler nor the assembler and the technique is scalable. A register file with the register renaming technique has the highest performance, requires fewer resources and consumes less power compared to other approaches. As a case study, we applied our technique to the configurable open-source ρ-VEX VLIW processor. We implemented a 64×32-bit, 4-write and 8-read ports register file utilizing BRAMs for a 4-issue ρ-VEX processor. This register file with register renaming saves 9109 Xilinx Virtex-4 FPGA slices by just utilizing 32 BRAMs compared to a pure slice-based register file with no effect on the overall frequency of the processor as well as the cycle count for any application.**

I. Introduction

When a processor issue-width gets wider [1][2][3], the number of read and write ports on the multiported register file has to increase accordingly. Generally, multiported memories in FPGAs are implemented using configurable resources (slices). The resource requirement for these memories increases exponentially with the number of read and write ports and the depth of the memory. Figures 1(a) and 1(b) depict the resource utilization and the maximum frequency, respectively, for a 32-bit, 4-write and 8-read (*4W8R*) ports memory with varying depths implemented utilizing Xilinx Virtex-4 FPGA slices. Figures 2(a) and 2(b) depict the resource utilization and the maximum frequency, respectively, for a 32-bit, 64-element multiported memory with varying number of ports implemented utilizing the same FPGA slices. From Figures 1 and 2, it can be observed that by increasing the depth and/or the number of ports of the memory increases the resources and reduces the frequency.

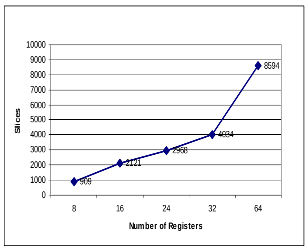
The multiported memories implemented utilizing BRAMs in banking and replication [4] have inherent conflicts associated with the write ports, and hence can not be utilized as true multiported memories unless a technique called *register renaming* [5][6] is applied, or additional hardware logic is utilized for port indirection. In [7], the presented design comprises banks of replicated BRAMs where a mechanism of indirection called *Live Value Table* (LVT) steers each read to the bank holding the most-recent write value. The LVT by itself is a multiported memory made up of configurable resources (slices), and has the same number of ports and depth as the actual multiported memory, but it stores a smaller number of bits ($\log_2$ of the total write ports). Figure 3 depicts the resource utilization for two LVTs with different port numbers (LVT-4W8R and LVT-8W16R) and varying depth. Along with these LVTs, the corresponding memories would need BRAMs for their implementations (32 for 4W8R ports memory and 128 for 8W16R ports memory). From Figure 3, it can be observed that even though the LVT stores a smaller number, its resource requirement increases exponentially with the number of ports and the depth of the required memory.

In this paper, we present a register renaming technique that is applied at compile (assemble) time. We implement our register file utilizing only BRAMs. We do not utilize the LVT approach and implement a register renaming technique to avoid write port conflicts and save configurable resources (slices). The technique is scalable and easily extendable for a register file with any number of ports and depth. As a case study, we implemented a register file and register renaming technique for a 4-issue very long instruction word (VLIW) processor called ρ-VEX [3]. Our experimental results show that a register file with the register renaming technique has the highest performance, requires fewer resources, and consumes less power compared to the other approaches.

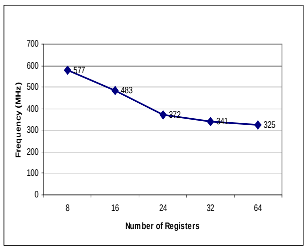
The remainder of the paper is organized as follows. Section II presents some previous work related to multiported register files. The ρ-VEX VLIW processor is briefly discussed in Section III. Section IV presents the design and implementation of our multiported register file utilizing BRAMs. Our register renaming technique for the ρ-VEX VLIW processor register file is presented in Section V. Results are discussed in Section VI. Finally, conclusions are presented in Section VII.

II. Related Work

The multiported register file of a VLIW processor implemented in FPGA is one of the most resource consuming modules. Generally, the multiported register file is implemented utilizing the FPGA configurable resources (slices) [3]. A quad-port memory design utilizing BRAMs is presented in [8]. Basically, it is a multi-pumped design with multiplexing. The external ports run at a slower frequency than the internal ports. In [4], the architecture of a configurable multiported register file for soft processor cores is presented. The register



(a) Resource utilization

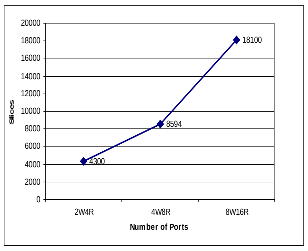


(b) Maximum frequency

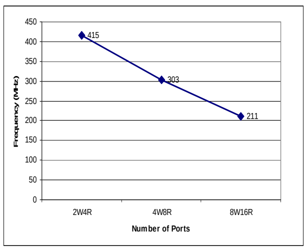
Figure 1: Resource utilization for a 4W8R ports memory implemented using FPGA slices

file is designed utilizing BRAMs. The design introduces write ports conflicts, and hence can not provide true multiported memories unless some additional hardware logic is utilized along with the BRAMs for port indirection or a register renaming technique is applied at either the hardware or software level. A quad-port memory implementation utilizing BRAMs is presented in [9]. The memory utilizes a mechanism of port indirection. In [7], the authors extended the technique of [9], and called it *Live Value Table* (LVT). The LVT is a multiported memory that keeps track of the write port number for each location of the multiported memory that is implemented utilizing BRAMs. The LVT is itself implemented utilizing configurable resources (slices) and has the same number of ports and depth as the actual multiported memory. Although the LVT-based design is very efficient, its resource requirement is likely to increase with the number of ports and depth of the memory up to an unacceptable value.

In this paper, we implemented a multiported register file utilizing only BRAMs. We do not utilize the LVT, and remove the write port conflicts with register renaming which is applied between the compiler and the assembler at static time. As a test case, we present the results for a 4-issue ρ-VEX VLIW processor. The advantage is that we do not need the extra resources as required by the LVT-based method, and we do not need to modify the compiler nor the assembler.



(a) Resource utilization



(b) Maximum frequency

Figure 2: Resource utilization for 64-element multiported memories implemented using FPGA slices

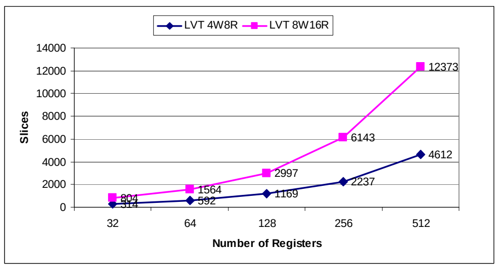


Figure 3: Resource utilization for LVT-4W8R and LVT-8W16R ports

III. The ρ-VEX VLIW Processor

The ρ-VEX is a configurable open-source softcore VLIW processor [2][3]. The instruction set architecture (ISA) is based on the VEX (VLIW Example) ISA [10], which is loosely modeled on the ISA of the HP/ST Lx [11] family of VLIW embedded cores. A VEX software toolchain including the VEX C compiler and the VEX simulator is made freely available by the Hewlett-Packard Laboratories [12], and can be utilized for architecture exploration. Different parameters of the ρ-VEX processor, such as the number and type of functional units (FUs), number of multiported registers (size of register file), number and type of accessible FUs per syllable, width of memory buses and different latencies can be changed. Additionally, it supports reconfigurable operations, as the VEX compiler supports the use of custom instructions via pragmas within the application code.

IV. The Multiported Register File utilizing BRAMs

In this section, we present the design and implementation of a multiported register file with W write ports and R read ports. We additionally present the limitations and conflicts imposed by the design.

A. Design and Implementation

The register file is designed utilizing the embedded BRAMs available in modern platform FPGAs. To provide portability among different vendors' FPGAs, we do not use specific vendor tools (such as the Xilinx CORE Generator) to instantiate the BRAMs. Instead, we use VHDL to define the BRAMs with the required aspect ratio and the number of ports of the register file. For our implementation, we utilized the Xilinx Virtex-4 *XC4VFX60-11FF1152* chip, which has 232 BRAMs of 18 kbytes each. The BRAMs on 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. Both ports on each BRAM are independent and each can be configured as either a read or a write port. Figure 4 depicts the organization of a 64×32-bit register file with 4 write ports and 8 read ports. In this design, each BRAM is configured as a 64×32-bit memory block with one read port and one write port.

In order to support multiple ports, the BRAMs are organized into banks and data is duplicated across various BRAMs

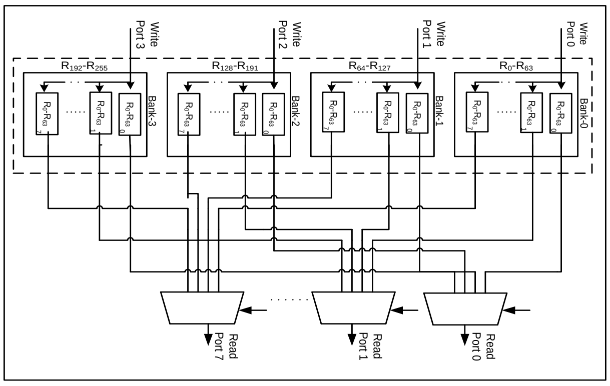


Figure 4: A 4W8R ports register file

within each bank. For a register file with 4-write and 8-read ports, we need 32 BRAMs distributed across 4 banks with 8 BRAMs per bank. Basically, in this design, the number of write ports defines the number of banks and the number of read ports defines the number of BRAMs per bank. For a register width of 32-bit, each BRAM can provide up to 512 such registers. Because register banks hold mutually exclusive sets of registers, they can be updated independently. Each register bank holds a separate write port, which can write to the registers dedicated to that bank.

In order to provide multiple read ports, multiple BRAMs are used within each register bank to store duplicate copies of the corresponding register subset. When register values are duplicated across BRAMs, the values of different registers can be read from different BRAMs simultaneously. Multiplexers are needed to provide access to the registers stored within each register bank.

B. *Design Limitations and Conflicts*

One of the main limitations of this banked-BRAM design of the multiported register file is associated with the write ports. This design influences the way registers and instructions are allocated and scheduled [4]. In this design, 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. Therefore, instructions cannot be scheduled to execute in parallel if they produce results in registers that belong to the same register bank. We remove this limitation by applying *register renaming* technique at software level, which is the main focus of this paper.

V. The Register Renaming Technique: Case Study for the ρ-VEX VLIW Processor

A register renaming technique is used to serve different purposes. It is used to allow multiple execution paths without conflicts between different execution units trying to use the same registers. It allows out-of-order processing by avoiding unnecessary serialization of program operations imposed by the reuse of registers, thereby, improving the performance. It is generally applied at hardware level such as the *Tag-Indexed* scheme and the *Reservation Station* scheme. Because these techniques are generally applied to allow out-of-order processing and these techniques consume considerable hardware resources, we do not use them as the ρ-VEX is an in-order processor system. The register renaming technique can also be applied at the compiler level. We do not modify the VEX compiler rather we apply the register renaming after the compiler at the assembler level. Applying the technique after compiler and at static time has the following advantages:

- It reduces the hardware cost and power consumption.
- It enhances the performance for an in-order machine because it is applied at static time, not at run time.
- All data dependencies are handled by the compiler and only register scattering need to be done to avoid the write ports conflicts making the design of the register renaming algorithm/tool very simple.

Before we discuss our register renaming tool, we first present our multiported register file and the connectivity of this register file ports with the functional units (FUs) of the ρ-VEX processor.

A. *4W8R Ports Register File for the ρ-VEX VLIW Processor*

The ρ-VEX is a parameterized VLIW processor. For this paper, we used a 4-issue processor. The register file is a 64×32-bit register file with 4-write and 8-read ports. We logically rename the registers as depicted in the Figure 4. Although each BRAM has 6-bit address bus, the total number of registers is 256, which is 64 times the total number of banks. This representation simplifies the register renaming design and guarantees that we always have free registers to rename.

B. *Register Port Connectivity*

The 4-issue ρ-VEX VLIW processor can execute an instruction composed of 4 operations or syllables at a time. The processor has 4 parallel functional units (FUs) for execution. All FUs have an arithmetic logic unit (ALU), FU1 and FU2 have multipliers units (MUL) as well. FU0 has a control/branch unit (CTRL) and FU3 has a memory/load-store unit (MEM). Syllable0, syllable1, syllable2, and syllable3 are issued to FU0, FU1, FU2, and FU3, respectively. Write ports WP0, WP1, WP2, and WP3 of the register file are associated with FU0, FU1, FU2, and FU3, respectively. This means that syllable0, syllable1, syllable2, and syllable3 can only have a destination register from R1 to R63, R65 to R127, R129 to R191, and R193 to R255, respectively. Registers R0, R64, R128, and R192 are read-only registers with value 0 as specified by the VEX ISA [10].

C. *Multiple Pass Register Renaming Technique*

The VEX compiler is an optimizing compiler. It handles all data dependencies and extracts a high level of instruction level parallelism (ILP) from a source code. Based on the application, the compiler can generate an assembly code having 4, 3, 2, or 1 operation(s) per instruction. It is based on data and control dependencies, and the available ILP. The compiler makes use of the available registers in a optimizing way according to the run-time architecture (RTA) [10]. The compiler can generate ALU, MUL, CTRL, MEM, or any other special operation in any order. It is the job of the assembler to direct an operation to its corresponding FU, and put NOPs (No Operations) for the unused syllables/operations. The main purpose of the register renaming is that within any instruction, no two operations should write to registers from a single register bank. The compiler uses 64 registers to generate the code, but our banked-BRAM register file provides 256 registers with 64 registers per bank.

The register renaming technique that we developed is based on a multiple pass algorithm. The tool is written in the C language. It takes the VEX assembly code as input and generates a register-renamed VEX assembly as output. Multiple passes are made in order to cover all possible conflicting conditions. The tool reads an instruction and parses its operations. It searches for the source and destination registers for all the operations of an instruction. It renames the destination registers for different operations in an instruction such that each operation could write to a separate register bank. A source register is renamed in a following instruction only, if that same register was renamed in an earlier instruction. The tool takes care of the following conditions while renaming the registers:

- An ALU operation can have a destination register from R1 to R255 excluding the read-only registers.
- A MUL operation can have a destination register from R65 to R191 excluding R128.
- A MEM operation can have a destination register from R193 to R255.
- A CTRL operation can have a destination register from R1 to R63.
- If there is a CTRL operation, no other operation in the same instruction can have a destination register from R1 to R63.
- If there is a MEM operation, no other operation in the same instruction can have a destination register from R193 to R255.
- If there is a MUL operation, no other operation in the same instruction can have a destination register from R65 to R127 or R129 to R191.

The tool utilizes different data structures for keeping track of the destination and source registers. A two-dimensional array of 2-by-64 characters is utilized to keep track of the old and new names of the destination registers for the entire input assembly code. Another array is utilized to keep track of all the registers in an instruction. Table I is utilized in renaming the destination registers. For example, if a destination register R2 needs to be renamed, the first choice would be R66, then R130, and then R194. This means that a multiple of 64 needs to be added when a destination register is being renamed. If any one of these registers is not available, or produce some other conflicts, then another first available conflict-free register number is selected. Source registers are renamed only by reading the array that keeps track of the destination registers.

Table I: Simplified register mapping

Bank0	0	1	2	-	-	61	62	63
Bank1	64	65	66	-	-	125	126	127
Bank2	128	129	130	-	-	189	190	191
Bank3	192	193	194	-	-	253	254	255

Having 64 registers in each bank guarantees that we always have a free register to rename and use it.

D. Conflict Conditions

In this section, we discuss some of the conflict conditions and show how these are removed by renaming registers.

1) Simple Instruction: When there is no loop generated within an instruction, register renaming becomes simple. It does not effect the previous instructions, and registers are renamed in a manner that no two operations have destination registers from a single register bank. For example, consider the following VEX assembly instruction composed of 4 operations:

```
add $r0.2 = $r0.1, $r0.0
add $r0.3 = $r0.1, $r0.0
mpyll $r0.4 = $r0.5, $r0.6
stw 0x0[$r0.6] = $r0.1
```

In the VEX assembly, \$r0.* represents the general-purpose register number * in cluster number 0. \$b0.* represents the branch register number * in cluster number 0. In order to execute correctly, the tool renames the registers as presented below:

```
add $r0.2 = $r0.1, $r0.0
add $r0.67 = $r0.1, $r0.0
mpyll $r0.132 = $r0.5, $r0.6
stw 0x0[$r0.6] = $r0.1
```

2) Loop/CTRL operation with a source register which is also the destination register in another operation of the same instruction: Loops are generated with CTRL operations. Renaming registers in instruction forming a loop may affect registers in the previous instructions and therefore another pass of the algorithm is required to correct this effect. Consider the following example code having few VEX instructions:

```
mov $r0.2 = -40
mov $r0.8 = $r0.0
;;
LABEL: cmple $b0.0 = $r0.2, 3
add $r0.2 = $r0.2, 4
;;
;;
goto LABEL
```

In the first pass, the tool will rename register *\$r0.8* in the first instruction. But when it parses the instruction with *LABEL*, it finds a conflict for *\$r0.2* which is used as a source as well as a destination register. When *\$r0.2* is renamed here, a second pass of the tool is required in order to rename *\$r0.2* in the previous instruction. The correct code is presented below:

```
mov $r0.66 = -40
mov $r0.136 = $r0.0
;;
LABEL: cmple $b0.0 = $r0.66, 3
add $r0.66 = $r0.66, 4
;;
;;
goto LABEL
```

3) CTRL operation with another operation in the same instruction which has a destination register from bank0: Consider the following example:

```
mov $r0.3 = $r0.0
goto LABEL
```

As the CTRL operation goes to FU0, therefore, no other operation in the same instruction can have a destination register from bank0. This register is renamed to the next available register from a different bank as presented below:

```
mov $r0.67 = $r0.0
goto LABEL
```

4) Destination register for a MEM operation should always be renamed: Consider the following example:

```
ldw $r0.2 = 0x0[$r0.1]
add $r0.3 = $r0.0, 1
```

As the MEM operation goes to FU3, and the compiler would always generate registers from R0 to R63 which belong to bank1 in our case, therefore, the destination register for a MEM operation should always be renamed to a register in bank3 as presented below:

```
ldw $r0.194 = 0x0[$r0.1]
add $r0.3 = $r0.0, 1
```

5) Destination register for a MUL operation should always be renamed : Consider the following example:

```
add $r0.2 = $r0.1, $r0.0
mpyll $r0.3 = $r0.7, $r0.8
mpyhh $r0.4 = $r0.7, $r0.8
add $r0.5 = $r0.1, $r0.0
```

As the MUL operation goes to FU1 or FU2, and the compiler would always generate registers from R0 to R63 which belong to bank1 in our case, therefore, the destination register for a MUL operation should always be renamed to a register in bank1 or bank2 as presented below:

```
add $r0.2 = $r0.1, $r0.0
mpyll $r0.67 = $r0.7, $r0.8
mpyhh $r0.132 = $r0.7, $r0.8
add $r0.197 = $r0.1, $r0.0
```

E. Scalability of the Approach

The design of a multiported register file is only limited by the number of BRAMs available in an FPGA. Implementing a register file with W write ports and R read ports would require $W \times R$ BRAMs distributed across W banks with R BRAMs per bank. Our register renaming technique is scalable and can be easily extended for any number of ports and depth of a register file.

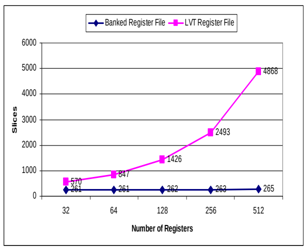
F. Application Development Framework

To optimally utilize the ρ-VEX VLIW processor, we present an application development framework. An application program written in C is compiled with the VEX compiler to generate the VEX assembly. The compiler is directed to generate code for 64 registers. The compiler generated assembly code is passed through our register renaming tool, which resolves the conflicts for the write ports and renames registers in the conflicting instructions. The tool utilizes 256 registers during renaming, as each of the 4 banks provides 64 registers. The output of the tool is again a VEX assembly code. The register-renamed assembly code is then assembled with the ρ-VEX assembler [3] to generate an executable for the ρ-VEX processor.

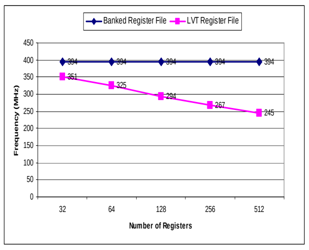
VI. Results and Discussion

A. Implementation Results

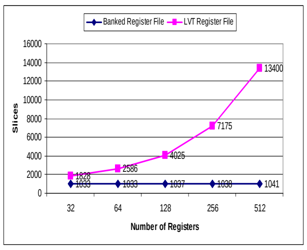
We utilized the Xilinx Virtex-4 *XC4VFX60-11FF1152* FPGA chip available on the *ML410* development board. Figures 5(a) and 5(b) depict the configurable resource (slice) requirement and the maximum frequency, respectively, for a *4W8R* ports LVT-based and our banked-BRAM register files (register renaming) with varying depth. Figures 5(c) and 5(d) depict the slice requirement and the maximum frequency, respectively, for *8W16R* ports LVT-based and banked-BRAM register files with varying depth. From these figures, it can be observed that the resource requirement for the LVT-based register file increases exponentially with the number of ports as well as with the number of registers for the same number of ports. For the banked-BRAM register file, the resource requirement does not vary with the number of registers for the same number of ports. In addition to the slices, the *4W8R* and *8W16R* ports register files need 32 and 128 BRAMs, respectively. It can be further observed that the frequency for the LVT-based designs reduces with the number of ports as well as with the number of registers for the same ports, while frequency for the banked-BRAM register files remains



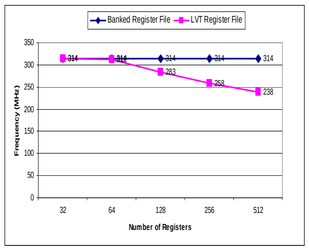
(a) Resource utilization (4W8R)



(b) Maximum frequency (4W8R)



(c) Resource utilization (8W16R)



(d) Maximum frequency (8W16R)

Figure 5: Resource utilization for 4W8R and 8W16R ports register

Table II: Resource utilization for 64 $\times$ 32-bit 4W8R ports register files

Register File Design	Slices	BRAMs	Max. Frequency
Slice-based	8594	0	303.012 MHz
LVT-based	847	32	324.992 MHz
Banked-BRAM	261	32	394.003 MHz

Table III: Resource utilization for 4-issue ρ-VEX VLIW processor

ρ-VEX with Register File	Slices	BRAMs	Max. Frequency
Slice-based	15494	0	68.685 MHz
LVT-based	6940	32	71.510 MHz
Banked-BRAM	6385	32	71.643 MHz

constant for the increasing number of registers for the same number of ports.

For the 4-issue ρ-VEX VLIW processor, we implemented a 64 $\times$ 32-bit register file with 4-write and 8-read ports. We implemented this register file using three types of designs: slice-based, LVT-based and our banked-BRAM-based. Table II presents the resource utilization for the three cases. We implemented a 4-issue ρ-VEX processor utilizing these three versions of the register file designs. Table III presents the resource utilization for the three designs of the ρ-VEX processor. From Tables II and III, it can be observed that our BRAM-based designs can considerably reduce the configurable resource utilization for the register file as well as the ρ-VEX processor. As our BRAM-based register file runs at a higher clock frequency compared to the slice-based register file, therefore, the BRAM-based design does not affect the critical path of the ρ-VEX processor. Hence, the overall frequency of the processor remains the same, even a little bit improved. Additionally, the cycle count for any application remains the same as is for the original ρ-VEX processor with the slice-based register file.

B. Dynamic Power Consumption Analysis

We calculated the dynamic power consumption for the three types of register files utilizing the Xilinx *XPower Analyzer* tool. We run these designs at 50 MHz frequency and put 100% load (writing/reading) on them. Compared to a slice-based and LVT-based, a 32-bit, 64-element 4W8R ports banked-BRAM register file with register renaming consumes 73.38% and 10% less dynamic power, respectively. Figure 6 depicts the dynamic power for a 32-bit 4W8R ports with varying number of registers for the LVT-based and the banked-BRAM (register renaming) register files. As it was not feasible to implement a slice-based register file with more than 64 registers, because the resources requirement is unacceptably high, therefore, we did not calculate its power consumption. From Figure 6, it can be observed that the dynamic power consumption for the LVT-based register file increases with the number of registers, while that for our banked-BRAM register file remains constant.

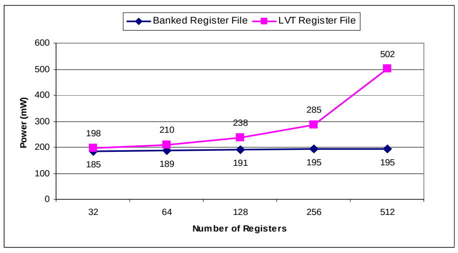


Figure 6: Dynamic power consumption for 32-bit 4W8R ports register files

VII. Conclusions

In this paper, we presented the design and implementation of a multiported register file utilizing BRAMs. We developed a *register renaming* technique that is applied between the compiler and the assembler and statically renames registers to avoid conflicts at write ports of the register file. We utilized the Delft configurable VLIW processor ρ-VEX as a case study for our multiported register file and our register renaming technique. We implemented a 64 $\times$ 32-bit, 4-write and 8-read ports register file utilizing BRAMs for a 4-issue ρ-VEX VLIW processor. This register file which can only be utilized with the *register renaming* tool saves 9109 slices by just utilizing 32 BRAMs compared to a slice-based register file with no effect on the overall clock frequency of the processor as well as the cycle count for any application. Our register file design has the highest performance, fewer resources and consumes less power compared to the other approaches.

References

[1] A.K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster, "An FPGA-based VLIW Processor with Custom Hardware Execution", in *13th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '05)*, pp. 107 - 117, 2005.

[2] S. Wong and F. Anjam, "The Delft Reconfigurable VLIW Processor", in *17th International Conference on Advanced Computing and Communications (ADCOM '09)*, pp. 244 - 251, 2009.

[3] S. Wong, T.V. As, and G. Brown, "ρ-VEX: A Reconfigurable and Extensible Softcore VLIW Processor", in *IEEE International Conference on Field-Programmable Technologies (ICFPT '08)*, pp. 369 - 372, 2008.

[4] M.A.R. Saghir, and R. Naous, "A Configurable Multi-ported Register File Architecture for Soft Processor Cores", in *International Symposium on Applied Reconfigurable Computing (ARC '07)*, LNCS 4419, pp. 14 - 25, 2007.

[5] Steven Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[6] J.L. Hennessy and D.A. Petterson, *Computer Architecture: A Quantitative Approach*, Third Edition, Morgan Kaufmann, 2003.

[7] C.E. LaForest, J.G. Steffan, "Efficient Multi-ported Memories for FPGAs", in *18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '10)*, pp. 41 - 50, 2010.

[8] Xilinx, Inc. 2002. Application Note XAPP228: Quad-Port Memories in Virtex Devices, http://www.xilinx.com.

[9] Altera, Inc. 2009. Advanced Synthesis Cookbook: A Design Guide for Stratix II, Stratix III, and Stratix IV Devices, http://www.altera.com.

[10] J.A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2004.

[11] P. Faraboschi, G. Brown, J.A. Fisher, G. Desoli, and F. Homewood, "Lx: A Technology Platform for Customizable VLIW Embedded Processing", in *27th Annual International Symposium of Computer Architecture (ISCA '00)*, pp. 203 - 213, 2000.

[12] Hewlett-Packard Laboratories. VEX Toolchain. [Online]. Available: http://www.hpl.hp.com/downloads/vex/.