PBIW Instruction Encoding Technique on the ρ-Vex Processor: Design and First Results

Renan Marks Felipe Araujo
Ricardo Santos
High Performance Computing Systems Laboratory - LSCAD
School of Computing - FACOM
Federal University of Mato Grosso do Sul - UFMS
Campo Grande - MS - Brazil

April 1, 2012

Abstract

This work presents the Pattern Based Instruction Word technique for instruction encoding applied to a soft-core VLIW processor named ρ-VEX. The ρ-VEX processor is a VLIW soft-core based on the ISA VEX architecture. Besides the experience on applying PBIW on the ρ-VEX processor, we also present some results considering area and power of the processor with the PBIW decoder on the datapath. Moreover, we show the code size reduction by adopting PBIW as the instruction encoding technique.

1 Introduction

Instruction encoding is a viable and alternative mechanism to minimize the impacts of the instruction fetch processor module on the area and performance of a processor. In this work, we present a instruction encoding technique and we apply the technique on a soft-core VLIW processor based on the ISA VEX processor architecture [1].

This technical report presents the Pattern Based Instruction Word Encoding (PBIW) Technique [2] for instruction encoding on processors and it shows the implementation of PBIW on the ρ-vex processor [3]. In addition, it presents experiments and some initial results by using PBIW on the processor. The PBIW technique is comprised of an instruction encoding algorithm which extracts out redundant data (operands) from the instructions and creates an encoded instruction and an instruction pattern. The encoded instruction has a pointer to its pattern and a set of non-redundant operand fields. The instruction pattern works as a map for the PBIW hardware decoder. A PBIW instruction pattern has pointers to the encoded instruction in order to indicate the operands to rebuild the original (non-encoded) instruction.

The rest of the report is organized as follows: Section **??** presents the PBIW technique; Section 3 sketches the ρ-vex VLIW processor; Section 4 presents the approaches used to apply PBIW on the ρ-vex processor; Section 5 shows the experiments and it discusses the results of the encoding technique on the processor; Finally Section 6 presents the conclusions of the work.

2 Pattern Based Instruction Word Encoding Scheme

The PBIW [4] encoding scheme explores a surjective function between the instruction set and the pattern set, aiming at reducing the instruction memory size of the programs. This technique is recommended for architectures that fetch long instructions from memory, such as VLIW and EPIC. The PBIW encoding scheme is based on the operand factorization technique [5–7] and an auxiliary table for storing instruction patterns.

After the instruction and register scheduling phases of the compiler backend, the PBIW encoding algorithm extracts out the redundant operands among the operations and creates an encoded instruction without operand redundancy. This instruction is stored in an Instruction Cache. The algorithm also keeps track of the original position of the operands creating a pattern which is stored in a pattern table. The instruction pattern is a data structure which contains the necessary information to rebuild the original instruction for the execution stages.

The PBIW strategy brings a dramatic reduction in instruction size since the redundant data no longer appears in the program instructions. The instruction patterns can be reused (surjective function) by different encoded instructions. An example of the (simplified) PBIW instruction decoding process can be seen in Figure 1.

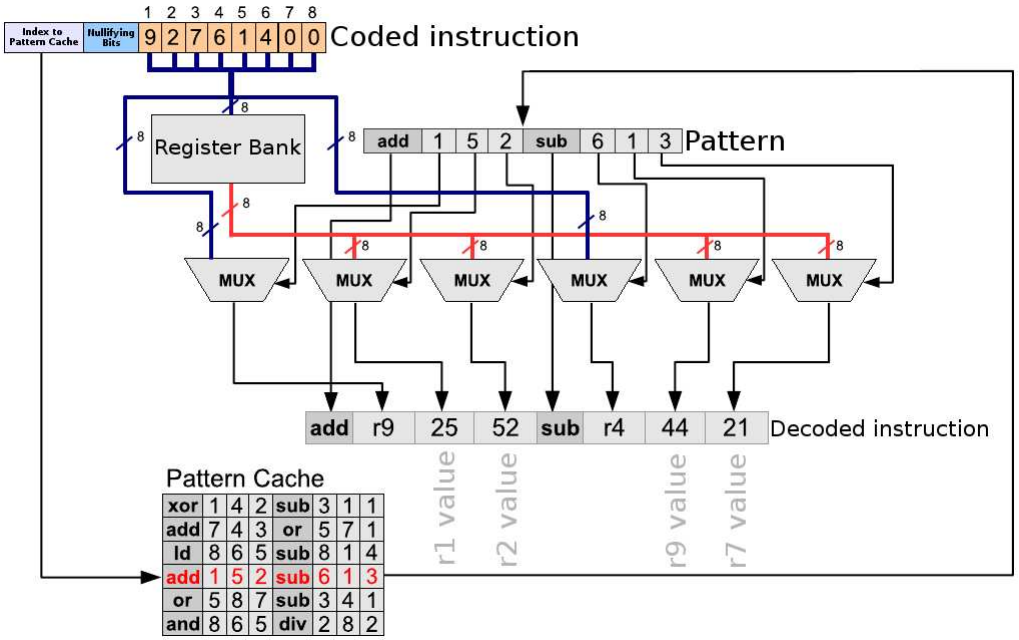

Figure 1: PBIW decoding circuitry diagram

The encoding algorithm that extracts operands and opcodes follow the operations dependencies in the original instruction set by the compiler scheduler. The PBIW encoded instruction is equivalent to a CISC instruction while the RISC operations for that CISC instruction are stored into the pattern. The pattern can be seen as the microcode to execute the CISC instruction.

It is worth noting that that PBIW is not an instruction encoding technique for a specific ISA (instruction set architecture). It seems to be straightly applied to ISAs which handle long fixed instructions comprised of single operations. However, the technique is not limited for those ISAs only. In order to apply PBIW on a specific ISA, the user should set, firstly, the number of operands fields to the encoded instruction. The number of operand fields will

determine the size of the pointers' fields of an instruction pattern. Another important step to determine the format of the encoded instruction is the size of the field to index a pattern from the pattern table. Eventually, the user can separate some bits to optimizations on the patterns since PBIW supports optimizations at encoding time. An optimization on the patterns is called patterns union. The optimization merges two patterns into one since the sum of operations in both patterns is less than the number of operations in a non-encoded (original) instruction and that union follows the hardware constraints of available functional units and registers.

3 The ρ-Vex Processor

ρ-Vex [3] is a configurable, extensible VLIW softcore processor described in VHDL based on the ISA VEX [1]. The ISA VEX is inspired by the ISA of HP/ST Lx processor family [1]. The VEX ISA supports multi-cluster cores, which one can provide a distinct VEX implementation. Each cluster supports instructions with a distinct number of operations. The extensibility of the instruction set allows, in a organized form, the definition of new instructions that have specific purposes.

The ρ-Vex processor is organized as following: 128 bits instructions comprised of 4 operations each, 4 arithmetic-logic units (ALU's), 2 multiplication/division units, 1 control unit (for branch instruction) and 1 memory access unit. The organization of the ρ-VEX units can be seen in Figure 2. The ρ-Vex toolchain for software development is provided with a robust framework which contains: a C compiler, ANSI C libraries, a software simulator — provided by HP [8] — and ρ-ASM, an assembler for the ρ-Vex processor — provided by [9]. The software development infrastructure for the ρ-Vex processor can be seen in Figure 3.

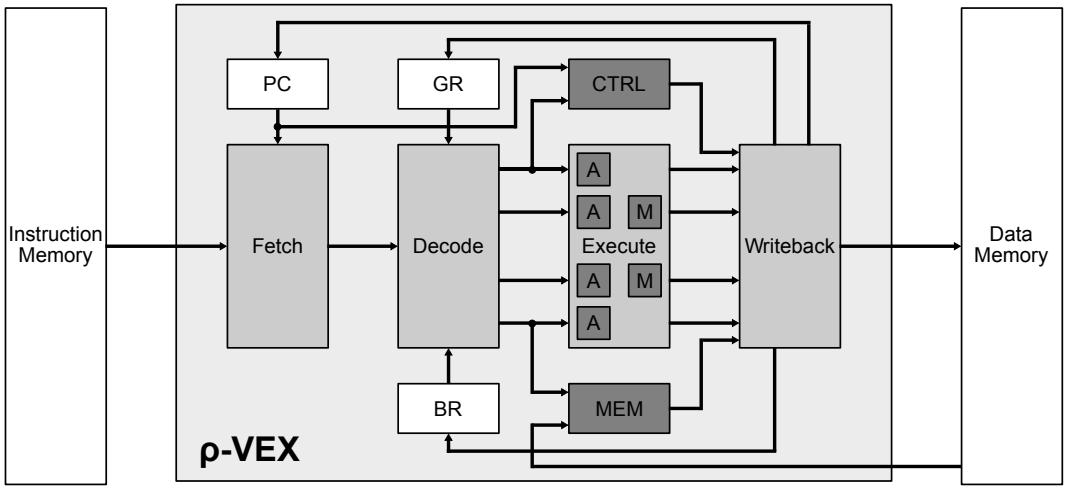

Figure 2: The ρ-VEX architecture.

Since ρ-ASM is a very limited assembler, mostly for extensibility, we have designed and implemented from scratch an entirely new ρ-VEX assembler in C++. The new assembler adopts a robust parser provided by HP [8] which can recognize flawlessly the assembly generated by the VEX C Compiler. This new assembler is able to generate the binary code described in VHDL for the ρ-Vex Instruction Memory with and without the PBIW encoded scheme.

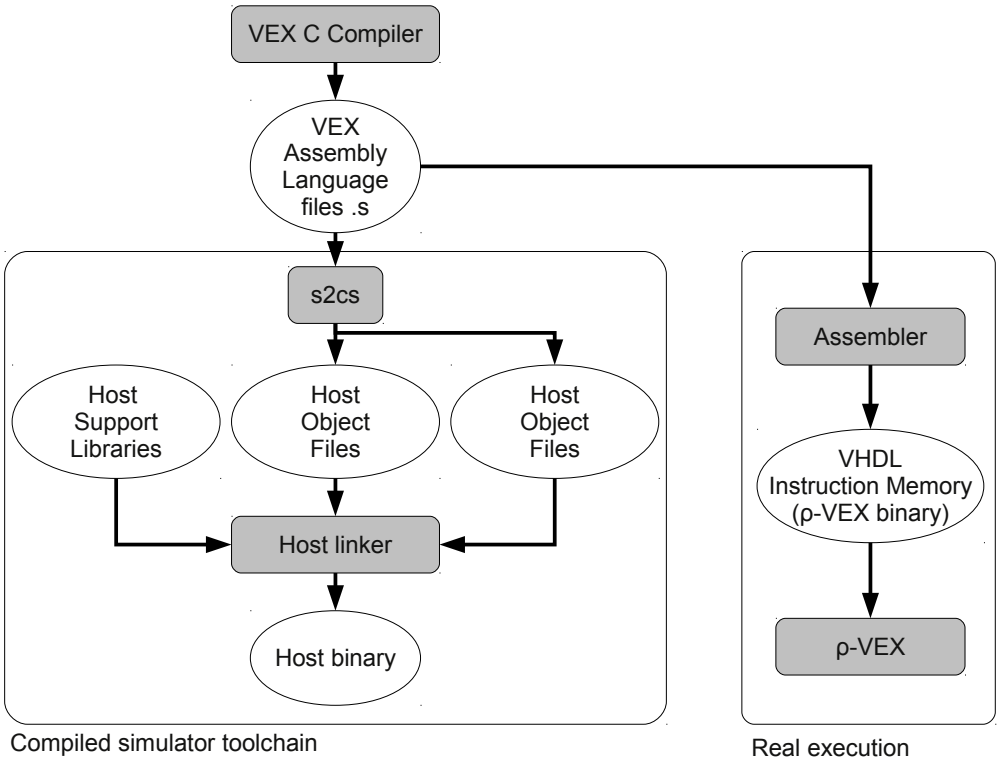

Figure 3: Toolchain for the ρ-VEX processor.

4 PBIW on the ρ-Vex Processor

After designing the encoded instruction format (Figure 4) and instruction pattern format (Figure 5) for the ρ-VEX processor, the design of the PBIW hardware decoder on the ρ-VEX processor, named decoder with constraints, has been carried out.

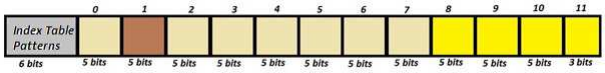

Figure 4: The ρ-VEX PBIW encoded instruction.

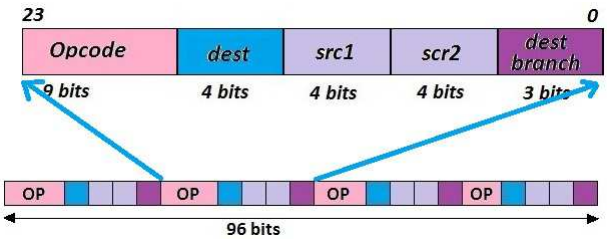

Figure 5: The ρ-VEX PBIW instruction pattern.

The instruction pattern has 9 bits dedicated for the opcode field (OP), 4 bits for the GR destination operand (dest), 4 bits for operand fields 1 and 2 (src1 and SCR2), 3 bits for destination operand BR (branch reg dest) resulting in 24 bits per operation. A complete pattern has 96 bits (24×4 operations).

The PBIW encoded instruction has a size of 64 bits and it is organized as shown in Figure 4. Field Index Patterns Table has 6 bits, 40 bits are dedicated for read registers fields (8 fields $\times$ 5 bits each). For write registers and immediate there are 18 bits divided into three fields of 5 bits and one 3-bit field.

The VEX architecture supports three types of immediate operands formats: 9-bits (short immediate), 12-bits (immediate branch offset), and 32-bits (long immediate). However, the ρ-VEX implementation does not support long immediate (32 bits).

In order to maximize the encoded instruction fields' usage, immediates are stored in the fields nine, ten and eleven of the PBIW encoded instruction. If the immediate has 9 bits (short immediate), it uses only the last two fields of 5 bits. If the immediate has 12 bits (branch offset immediate), it uses all three fields. If there is not immediate in the original (non-encoded) instruction, the fields are used for read registers.

For instructions reading data from the BR register bank we have designed that the address (BR src) would be stored in the first field of the Fields dedicated to read registers. The ρ-VEX operations which need that feature are operations of the types III, IV, and VIII (ADDCG, DIVS, SLCT, SLCTF, and BRBRF) as indicated in [3].

4.1 The PBIW Decoder Circuit with Constraints

Figure 6 sketches the PBIW decoder circuit designed and prototyped at the data path of ρ-VEX processor. After the fetch of an encoded instruction, the read registers addresses are sent to the read ports of BR and register banks. Concurrently with the reading of the registers, a pattern is fetched from the pattern table using the index stored in the PBIW encoded instruction. In that example, the Index Table Patterns field size has 6 bits so that the pattern table (P-cache) has a size of 1.6 Kbytes (64 lines x 96 bits = 6.2 kbits = 1.6Kbytes) and it is implemented as a direct access table.

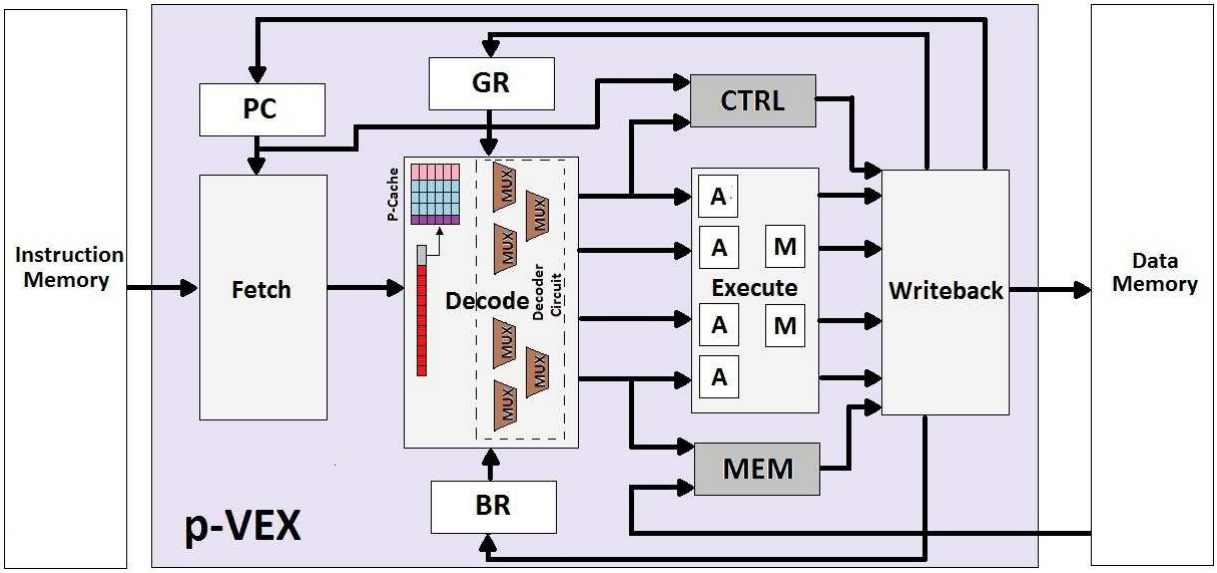

Figure 6: The PBIW decoder circuit on the ρ-VEX datapath.

After reading the registers, the values are available on the input of the source operand multiplexers. The destination register and immediate are provided on the input of the

destination operand multiplexers [2]. Select signals that select the values available on the input of the multiplexers are also provided by the instruction pattern.

According to our design of the PBIW scheme on the ρ-VEX isa, field dest of the instruction pattern can only point to items 8-11 of the PBIW encoded instruction. Fields src1 and src2 point to items 0-7 of the PBIW encoded instruction. An example of the designed PBIW decoder circuit decoding an operation can be seen in Figure 7.

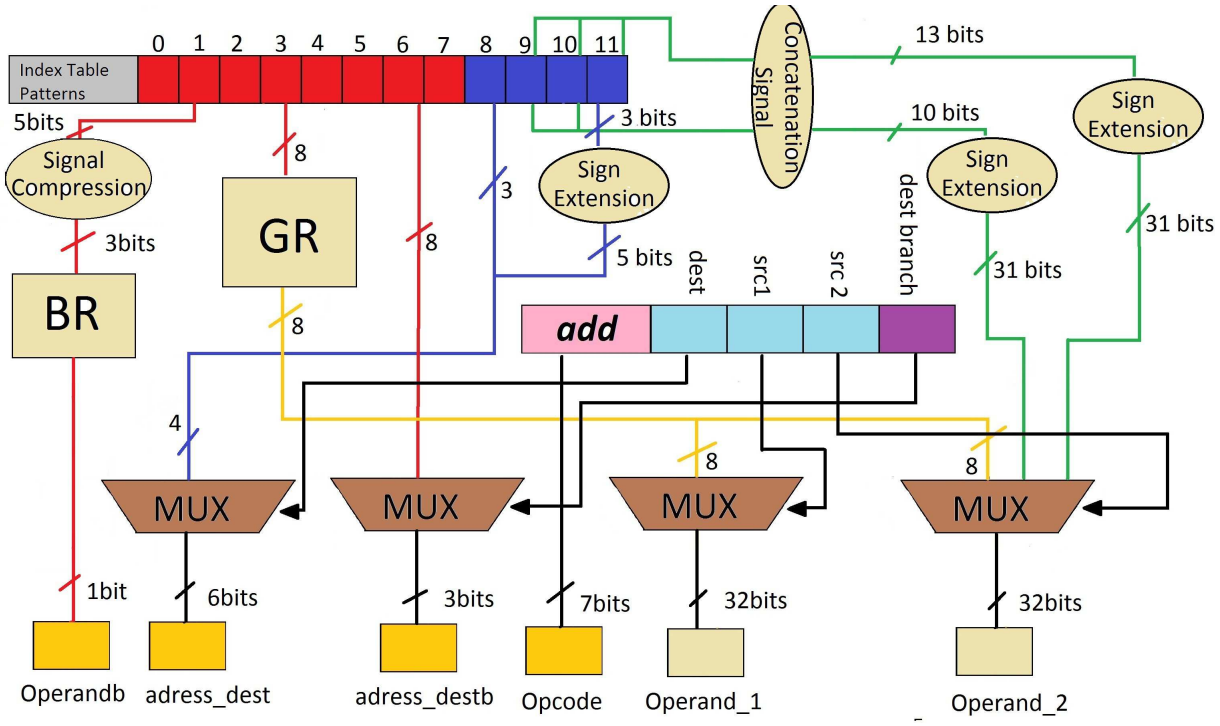

Figure 7: An instruction decoding example by the ρ-VEX PBIW decoder circuit with constraints.

4.2 The PBIW Decoder Circuit without Constraints

By designing a PBIW decoder circuit without constraints for the ρ-VEX processor, we have redesigned the formats for the encoded instruction and instruction pattern. The new instruction has a 9 bits opcode field (OP) being 7 bits for the opcode and 2 bits for the branch source (SRC BR), there are a 4 bits field for the destination operand GR (dest), 4 bits fields for the operand fields 1 and 2 (src1 and src2), 3 bits for the destination operand BR (branch reg dest). A complete pattern has 96 bits (24 $\times$ 4-bits operations) as shown in Figure 8.

The encoded instruction is comprised of 12 fields of 5 bits each. The fields are used for the operand (source and destination) register address and a 6-bits field for the pattern table index. An encoded instruction has 64 bits according to the format presented in Figure 9.

For the new decoder without constraints, an encoded instruction has no fixed fields for read and write registers. This means that the address of a general write (GR) can be stored in fields 0-11 of the instruction. The address of a branch write register (BR) can be stored in fields 0-7 of the instruction.

According to the ρ-VEX instruction set, only instructions of type III and IV to VIII (ADDCG, DIVS, SLCT, SLCTF, BR, and BRF) need to read data from the BR register

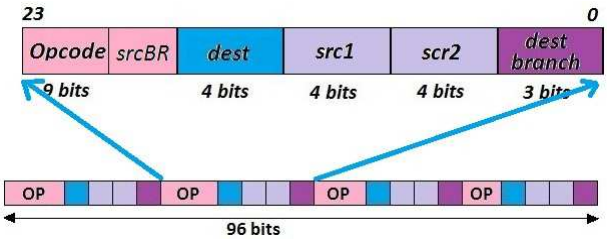

Figure 8: The ρ-VEX PBIW instruction pattern without constraints.

Figure 9: The encoded ρ-VEX instruction without constraints.

bank [3]. In the new design of the decoder circuit, we have modified the way the read branch registers are stored in the instruction. Instructions ADDCG, DIVS, SLCT, and SLCTF have the address of the read branch register in the least significant three bits of the opcode. This new field is responsible for selecting the encoded instruction field that contains the address of the read branch registers for operations ADDCG, DIVS, SLCT, SLCTF, BR, and BRF. Since the size of the field to the address of the branch register has only 3 bits, the addresses of the branch registers should be stored in items 0-7 of the encoded instruction. For operations BR and BRF we have designed that the addresses of their read branch registers are stored in fields 5-6 of the encoded instruction.

The ρ-VEX processor supports three types of immediate operands: 9-bits (short immediate), 12-bits (immediate branch offset), and 32-bits (long immediate). However, the current ρ-VEX implementation [3] does not support the use of long immediate (32 bits). An immediate is stored the same way as exemplified in [2]. The last three fields of the encoded instruction are used for immediates. If the immediate is 9 bits, it uses fields 9 and 10 (the decoder uses only 9 bits). If the immediate is 12 bits, it uses fields 9, 10, and 11. If the instruction does not have immediates, fields 9, 10, and 11 can be used for any other kind of operand.

Figure 6 sketches the PBIW decoder circuit without constraints designed in the datapath of ρ-VEX. The location of the decoder circuit PBIW without constraints is the same to the decoder circuit PBIW with constraints. Although the number of multiplexers of that second approach (without constraints) has increased when compared to the PBIW decoder circuit with constrains, there is no significant increase in area. The new encoding scheme for the PBIW decoder circuit without constraints provides simplifications in the decode unit of the ρ-VEX processor. These simplifications brought not only a reduction in occupied area but also in energy consumption, which will be discussed in more detail in Section 5.

After the encoded instruction fetch, the addresses of the read registers are sent to the read ports of BR and GR register banks. At the same time, the encoded pattern is fetched from the pattern table using the index stored in the encoded instruction. The pattern table (P-cache) has 1.6 Kbytes (64 lines $\times$ 96 bits = 6.2 kbits = 1.6Kbytes) and it is implemented as a direct access table.

After the registers read, the values are available on the input multiplexers dedicated to the source operands, the rest of the registers and immediate are provided on the input

multiplexers dedicated to the destination operand. The signals that select the values of the multiplexers come from the pattern. An example of the decoder circuit without constraints for an operation can be seen in Figure 10.

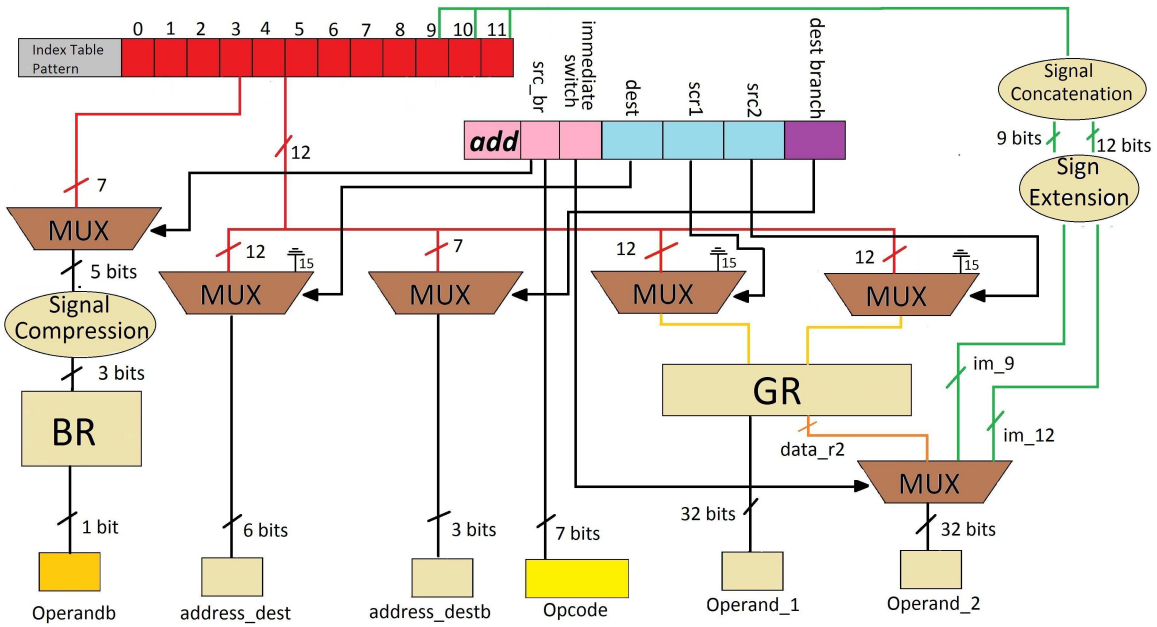

Figure 10: An instruction decoding example by the ρ-VEX decoder circuit without constraints.

The dst GR field of the instruction is checked to decide which is the destination register as the result of the operation. When the dst GR field has the address of the register \$r0.0, the dst BR address field is used as the result of the operation. The responsible for this evaluation is the ρ-VEX control unit.

In the decoder circuit without restrictions we have set field zero of the encoded instruction to always store zero (= 00000). So that if an operation needs to reference the register \$b0.0 as the destination, the field dest of pattern must contain the value zero, thus selecting field zero of the encoded instruction.

Currently, we have optimized the procedure to obtain a zero value from the instruction. We have changed the approach presented before aiming at saving bits of the encoded instruction. Whenever an instruction has a zero value (\$r0.0), its instruction pattern should stores value 15. At instruction decoding, value 15 will select input values on multiplexers grounded to zero. If an instruction uses register \$b0.0 (read or write), the respective pattern points to one of the fields 0-7 of the encoded instruction.

Fields src1 and src2 of the instruction pattern can also point to items 0-11 of encoded instruction. Signals src1 and src2 works as selectors for two multiplexers and they allow select the right position (in the encoded instruction) which has the address of its source operands 1 and 2 respectively. The selected operands are sent to the BR register bank.

When an operation handles 9-bits or 12 bits immediate, the second source operand is changed. The responsible for the selection is the immediate switch field of the pattern. Case the immediate is 9 bits, the immediate switch field should be 01 selecting signal im_9. Case the immediate is of 12 bits, field immediate switch should be 10 selecting signal im_12. Otherwise, field immediate switch must be 00 selecting signal data_r2.

Logic elements	ρ-VEX	ρ-VEX with constraints	ρ-VEX without constraints
Total combinational functions	10,506	8,730	9,581
Dedicated logic registers	1,805	1,739	1,768
Total pins	7	7	7
Total logic elements	10,510	8,766	9,602
Total memory bits	8,192	16,384	8,192
Embedded Multiplier 9-bit elements	28	28	20

Table 1: Area comparison between the three projects.

5 Experiments and Results

The experiments presented below aim at analyzing the impact on the occupied area and power consumption of the decoder circuit on an FPGA platform. We have compared the original ρ-VEX processor and the two designs of the PBIW decoder circuit. The inputs for the processor are two classic programs (Fibonacci and Factorial). The two program codes are loaded directly into memory instruction (i_men.vhdl) before synthesis. A new signal, called select_program has been implemented in order to select the program to be executed from the instruction memory. The FPGA technology used in the experiments was based on the Altera Cyclone II family (device EP2C35F672C6) which contains a total of 33,216 combinational functions, logic elements, dedicated logical registers, and a total of 475 pins.

5.1 Area

In this experiment we have used Altera Quartus II web Edition software. The project top-level entity is system.vhdl. All three projects (original ρ-VEX, ρ-VEX with constraints, and ρ-VEX without constraints) have been standardized for comparison.

Table 1 summarizes and compares the number of logic elements used by the three implementations of the ρ-VEX processor. It can be seen that the ρ-VEX+decoder provides a small area compared to the original ρ-VEX implementation. Decoder with constraints achieved better results compared to the decoder without constraints. Despite reducing the program footprint (Tables 2, 3, 4, and 5), the approach without constraints increase of inputs for the multiplexers in the decoder circuit. In the experiment, the ρ-VEX with constraints obtains a reduction of 16.5% in the number of logic elements used when compared to the original ρ-VEX. The ρ-VEX without constraints obtains a reduction of approximately 8.6%.

The ρ-VEX without constraints provides better results with respect to the amount of embedded multipliers and the total bits of memory. That design achieves a reduction of 28.6% considering the total embedded multipliers compared to the original ρ-VEX.

5.2 Static Evaluation

The static experiments indicate the the code size in bytes, the reuse rate, and the compression ratio comparing the implementation of the original ρ-VEX to the implementation of the ρ-VEX without constraints. The reuse rate indicates the amount of encoded instructions which use the same pattern on average. The compression ratio reports the percentage of

Program	Code Size original ρ-VEX	Code Size ρ-VEX with constraints	Compression Ratio (%)	Reuse Rate
Fibonacci	64	168	-162%	1,13
Factorial	240	332	-38,3%	2,3
Fibonacci+Factorial	304	452	-48,7%	2,3

Table 2: Static evaluation comparison between the original ρ-VEX design and the ρ-VEX with constraints.

Program	Code Size original ρ-VEX	Code Size ρ-VEX without constraints	Compression Ratio (%)	Reuse Rate
Fibonacci	64	120	-87%	1.0
Factorial	240	264	-10%	1.8
Fibonacci+Factorial	304	328	-7.89%	2.0

Table 3: Static evaluation comparison between the original ρ-VEX design and the ρ-VEX without constraints.

reduction in the program code size obtained with the use of the PBIW technique. It should be noted that a higher reuse and the reduction factor mean a more efficient encoding technique.

Tables 2 and 3 present the results of the static evaluation of the programs implemented in instruction memory. Columns Reuse Rate and Compression Ratio (%) are calculated according to Equations 1 and 2, respectively.

$$\text{Reuse Ratio} = \frac{\text{Number of encoded instructions}}{\text{Number of patterns}} \tag{1}$$

$$\text{Compression Ratio} = 1 - \frac{\alpha}{\beta} \tag{2}$$

Where: α is the size of compressed code (encoded). Value of column ρ-VEX without constraints. β is the size of uncompressed code. Value of the column original ρ-VEX.

Tables 2 and 3 shows negative compression rate for the Fibonacci, Factorial and Fibonacci + Factorial which means that their codes were increased, in the better case, by 87%, 10%, and 7.89% respectively (Table 3).

One can observe that the use of the PBIW encoding technique have not brought improvements (reduction) on the size of the program. The results of the static evaluation considering the ρ-VEX with constraints are even worst than the design without constraints. The compression ratio considering the encoding with the design with constraints provided an increase of 162%, 38.3% and 48.7% for programs Fibonacci, Factorial, and Factorial+Fibonacci, respectively.

5.3 Dynamic Evaluation

The dynamic evaluation measures the impact of the encoding technique in the performance of the processor during execution. In this experiment, it has been considered that the instruction fetch time per byte is the same for both implementations of the ρ-VEX processor (with and without the decoder circuit). Therefore, if the design adopts a small instruction size, the instruction fetch latency will be lower. The decrease in the latency of the instruction

Program	original ρ-VEX	ρ-VEX with constraints	Compression Ratio (%)	VEX Instruction	Encoded Instruction
Fibonacci	1520	1400	8%	94	175
Factorial	448	360	20%	28	45
Fibonacci+Factorial	1952	1760	10%	122	220

Table 4: Dynamic evaluation comparing the original ρ-VEX and the ρ-VEX with constraints.

Program	original ρ-VEX	ρ-VEX without constraints	Compression Ratio (%)	VEX Instruction	Encoded Instruction
Fibonacci	1,520	1,112	27%	94	139
Factorial	448	264	42%	28	33
Fibonacci+Factorial	1,952	1,376	30%	122	172

Table 5: Dynamic evaluation comparing the original ρ-VEX and the ρ-VEX without constraints.

fetch implies an increase in performance, since all the operations have the same latency for both designs of the processor and, especially, the number of operations is the same for both programs (non-encoded and encoded).

Tables 4 and 5 summarize the results of the dynamic evaluation of the programs comparing the original ρ-VEX and the ρ-VEX with and without constraints. Columns original ρ-VEX and ρ-VEX with (without) constraints indicate the number of bytes fetched from the instruction memory. Column VEX instruction and Instruction PBIW contain the size of non-encoded and encoded instructions fetched in the instruction memory. It is worth noting that the an original ρ-VEX instruction has 128-bits and an encoded PBIW instruction has 64-bits. Column Compression Ratio (%) shows the reduction in the number of bytes fetched from the memory.

Table 5 shows a promising compression rate for programs Fibonacci, Factorial and Fibonacci +Factorial. The programs have been reduced by 27%, 42%, and 30%, respectively. The compression ratio achieved for Fibonacci, Factorial, and Factorial + Fibonacci implicated a reduction of fetch latency resulting in a real performance gain as expected theoretically. For the dynamic experiments, the results obtained with the design of the ρ-VEX with constraints has reached a maximum reduction of 20% in the size of program Factorial.

5.4 Power Analysis

Analysis of power consumption in digital circuits has been known by two methods: software estimation and measurement. The measurement involves the implementation and prototyping of the circuit together with additional circuitry, allowing more precise conclusions.

In our experiments we have chosen power analysis via software estimation. We have adopted PowerPlay Power Analysis integrated into Altera Quartus II as the basic tool for power estimation. PowerPlay allows estimating static and dynamic power consumption showing the partial results for the core and pads of the FPGA, the total current that traverses the circuit, and the power consumption associated.

Metric	original ρ-VEX	ρ-VEX with constraints	ρ-VEX without constraints
Total Thermal Power Dissipation	138.96 mW	136.47 mW	131.57 mW
Core Dynamic Thermal Power Dissipation	27.86 mW	25.41 mW	20.56 mW
Core Static Thermal Power Dissipation	80.02 mW	80.01 mW	79.99 mW
I/O Thermal Power Dissipation	31.08 mW	31.05 mW	31.01 mW

Table 6: Power analysis of the three designs for program Factorial.

Metric	original ρ-VEX	ρ-VEX with constraints	ρ-VEX without constraints
Total Thermal Power Dissipation	147,46 mW	148,67 mW	135,16 mW
Core Dynamic Thermal Power Dissipation	36,35 mW	37,60 mW	24,12 mW
Core Static Thermal Powe Dissipation	80,05 mW	80,05 mW	80,01 mW
I/O Thermal Power Dissipation	31,06 mW	31,06 mW	31,03 mW

Table 7: Power analysis of the three designs for program Fibonacci.

PowerPlay performs the estimation of the power consumption of circuits implemented in FPGA based on knowledge of the specific characteristics of the device such as: current leakage, capabilities of each of the internal nodes, and a switching rate associated with each of these nodes.

The switching rate or activity rate of each node depends on the circuit and all events of the entries. The rates of activity can be provided by the tool in two different ways. For manual insertion or through a simulation file (*.Vcd). Building the testbench to generate this file should be made faithfully reproducing the actual operation of the circuit.

Based on the features of the tool, we have conducted two experiments for each design of the ρ-VEX processor (original ρ-VEX, ρ-VEX with constraints, and ρ-VEX without constraints). The first experiment investigates power consumption of the design taking the code of program Factorial as the input. The second experiment adopts the same methodology but for Fibonacci program. The two programs have been loaded to instruction memory (i_men.vhdl) before synthesis. Tables 6 and 7 summarize and compare the results obtained for the three designs.

One can observe (Tables 6 and 7) that the encoding technique designs provide better (lower) power consumption. PowerPlay provides four metric to analyze the desigs in the experiment. Metric Core Dynamic Thermal Power Dissipation provides an estimate of the resulting dynamic power dissipated by the circuit. Metric Core Static Thermal Power Dissipation provides an estimate of the resulting static power dissipated by the circuit. Metric I/O Thermal Power Dissipation Power provides an estimate of the I/O (Input and Output) power consumption for each design. The Thermal Power Dissipation Total metric provides an estimate of the total power consumed by the circuit.

The total power consumption by a device is comprised of the following components: dynamic power, static power, and I/O (Input and Output) power. The static power is the consumed energy by a device when it is turned off. This type of energy is known as “standby”. The dynamic power is the consumed energy for the device when it is running. The main variables affecting dynamic power are capacitive loads, supply voltage and clock frequency. The I/O power (Input Output) is the consumed energy due to charge and discharge of external capacitors connected to pins of external devices.

It can be observed that the addition of the PBIW decoder circuit on the ρ-VEX design provides a reduction in the dynamic power consumption component. By using t program Fibonacci as the input, we have obtained a maximum reduction of 33.7% while program Factorial have obtained a maximum reduction of 26.2%.

6 Conclusions

This work presented the Pattern Based Instruction Word Encoding (PBIW) Technique for instruction encoding on processors and it shows the design and implementation of PBIW on the ρ-vex processor. Besides the discussion concerning the design of PBIW and the implementation on the ρ-vex soft-core VLIW processor, we have also shown some initial results of the technique considering the area usage and power consumption of the processor.

The experiments show gains up to 42% considering compression rate in the dynamic evaluation. The power analysis experiments have achieved improvements up to 33.7%. All the comparisons have been carried out with the original ρ-VEX to the ρ-VEX with a PBIW decoder circuit.

We are still working on the PBIW technique and its usage on processors like ρ-VEX. By the results, we can notice that the instruction encoding technique brings relevant gains in performance, area, and power consumption for the adopted hardware (processor).

References

[1] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Elsevier, 2005.

[2] R. Batistella, R. Santos, and R. Azevedo, "A New Technique for Instruction Encoding in High Performance Architecture," Tech. Rep. IC-07-027, Unicamp, Brazil, September 2007.

[3] T. van As, "ρ-VEX: A Reconfigurable and Extensible VLIW Processor," Master's thesis, Faculty of Electrical Enginnering, Mathematics and Computer Science. Delft University of Technology, 2008.

[4] R. Batistella, "Pbiw: Um esquema de codificacao baseado em padrões de instrução," Master's thesis, Instituto de Computação - Universidade Estadual de Campinas, Campinas-SP, Fevereiro 2008.

[5] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain, "Code Compression Based on Operand Factorization," in *Proceedings of the* 31^{st} *Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pp. 194–201, IEEE Computer Society, 1998.

[6] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting, "Code compression," in *Proceedings of Programming Language Design and Implementation*, pp. 358–365, ACM, 1997.

[7] M. Franz and K. Thomas, "Slim binaries," *Communications of the ACM*, vol. 40, no. 2, pp. 87–94, 1997.

[8] Hewlett-Packard Laboratories, "VEX Toolchain." Disponível em: http://www.hpl.hp.com/downloads/vex/, Março 2011.

[9] T. van As, S. Wong, and G. Brown, "ρ-VEX: A Reconfigurable and Extensible VLIW Processor," in *IEEE International Conference on Field-Programmable Technology*, IEEE, 2008.