Technische Universität Dresden
Department of Computer Science
Institute for Computer Engineering
Chair for VLSI design, diagnostic and architecture

Complex Training Period Processor Design

PauloBlaze

Paul Richard Genßler
born on May 26, 1990 in Berlin
(Mat.-Nr.: 3569856)

Supervising Professor:
Prof. Dr.-Ing. habil. Rainer G. Spallek

Adivsor:
Dipl.-Inf. Oliver Knodel

Dresden, September 22, 2015

Contents

1 Introduction

Field Programmable Gate Arrays (FPGAs) have always had a great potential for an efficient implementation of parallelizable algorithms. Streaming based problems benefit from the low I/O latency and, thanks to the configurable cells, can be mapped very well. However, step by step sequences have proven to be difficult. Complex tasks may demand a hybrid solution, configurable logic combined with a dedicated CPU on a single chip like a Xilinx Zynq or an Altera Cyclone. State machines are suitable for short sequences and are often used. But those state machines have practical limits, there is a huge bulk of states, one cannot cope with the number of transitions, more input values, more output signals, a more and more complex implementation leads to more resource usage. This problem is not new and the typical solution is a so-called softcore. Such a softcore contains the mapping of a CPU model onto the internal FPGA resources and the user can execute ordinary Assembler or C source code. A popular core is the KCPSM6 also known as PicoBlaze [Xil11]. It is used in many applications like elliptic curve cryptography [HB10], a floating-point controller [KG05] or a multiprocessor system [YS06]. Its implementation is utterly compact and with up to 238 MHz very fast, however, because of its direct description of Look-Up Tables (LUTs) it is limited to current Xilinx FPGAs and not easily customizable.
In this work a processor, called PauloBlaze, is developed which is 100 % compatible to the PicoBlazes ISA and all of the signal timings. It should be very easy to replace a PicoBlaze in a current project, to modify this new implementation, or to deploy it without being restricted to particular platforms. These benefits should outweigh the speed and area losses.
This work is structured into four chapters. An introduction and motivation is given in chapter 1 followed by chapter 2, an overview on the implementation of the single modules and how they work together. The resource usage is summarized in chapter 3 and compared to the PicoBlazes. Advantages and compromises of the PauloBlaze are discussed in the last chapter.

2 Implementation

The PicoBlaze is implemented by the direct description of LUTs. This provides the desired speed and area advantages, but it is at the same time the biggest disadvantage. The core only works with Xilinx FPGAs providing 6-input LUTs, limiting it to Spartan-6, Virtex-6 and all 7-Series devices.
This chapter describes the implementation of the PauloBlaze, which is written in pure VHDL code to provide maximum flexibility and portability. It consists of several components like the decoder (section 2.1), for jumps and calls the program counter (section 2.2), the register file (section 2.3) for interaction with the Scratch Pad Memory, simple calculation and data manipulation operations are handled by the ALU (section 2.4) and the I/O module (section 2.5) takes care of input and output operations. Figure 2.3 presents these components and the data flow between them. Because it is fully compatible to the PicoBlaze, one may refer to the original documentation [Cha12] for specific details, available instructions and timing diagrams.

2.1 Decoder

This critical component in the processor is the interconnection point in the design as it can be seen in figure 2.2. It takes the complete 18-bit instruction word and evaluates it. Depending on the type of instruction, different word structures are possible. The most common instructions are displayed in figure 2.1, after the 6-bit opcode, it uses 4 bits to address the register and the remaining 8 bits as an immediate or to address a second register.

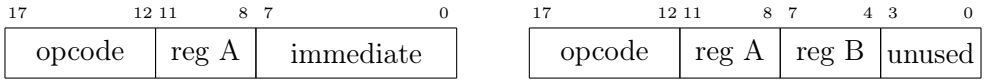

Figure 2.1: Bit Patterns of common Instruction

It is also possible to use those 8 bits as a parameter. This is done in the shift and rotate operations extensively where the lower 4 bits determine which direction to shift and what happens to the shifted bit.
The opcodes 17_{16}, 23_{16}, 27_{16}, $2A_{16}$, 33_{16}, $3B_{16}$, $3F_{16}$ are free and can be used to implement new instructions.
External control signals like *reset*, *interrupt* or *sleep* as well as internal ones (*reset request*, *zero*, *carry*) will be processed by the decoder.

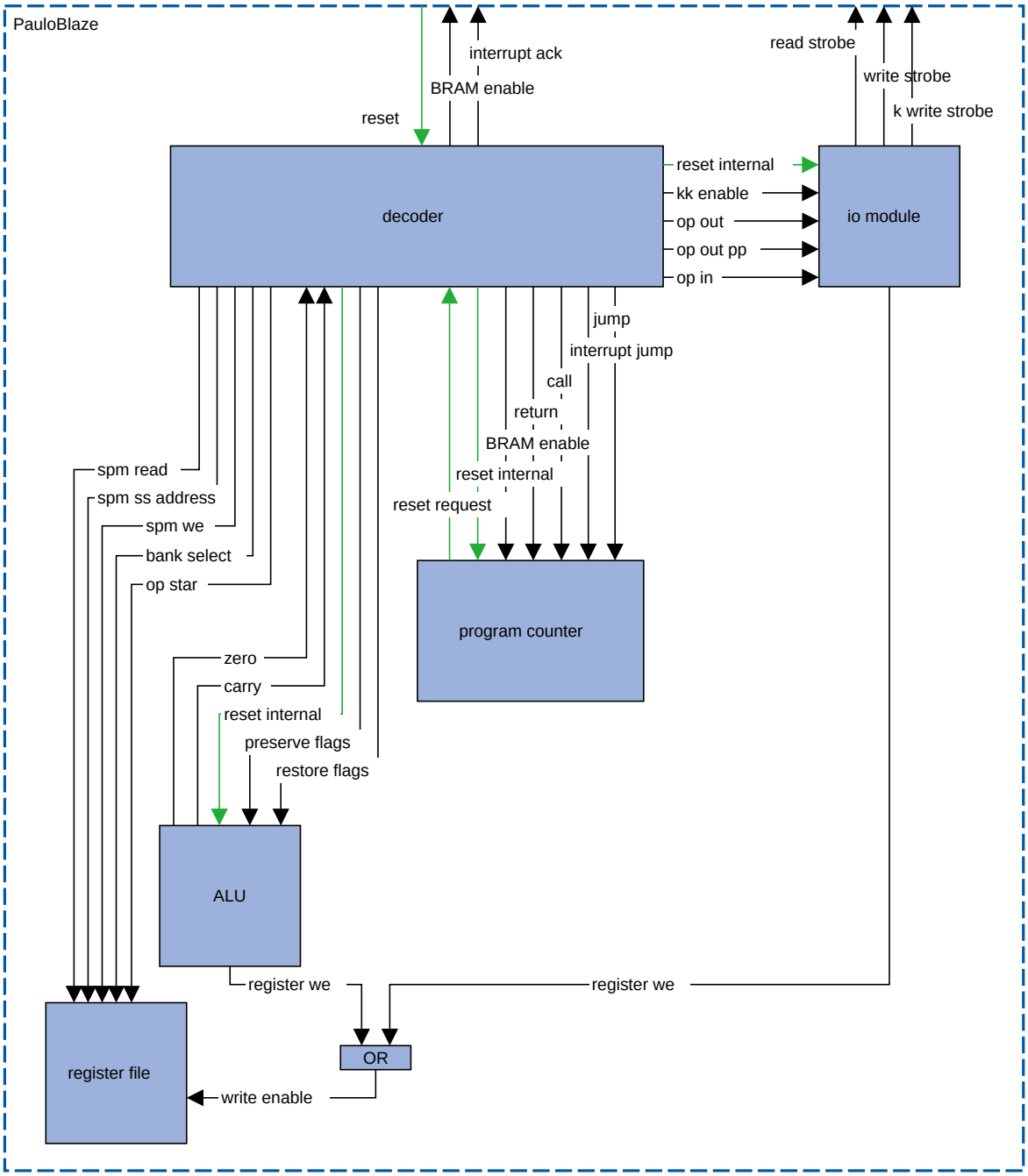

Figure 2.2: Control Flow Graph

2.2 Program Counter

The next address is calculated by the program counter, which increments it by one every second cycle during normal execution. The decoder drives the *jmp_addr* input used to determine the next address in case of a jump or a call. If the latter occurs, the program counter stores the subsequent address onto a stack and jumps to the target. After the interrupt or call has finished

(using one of the return instructions), the topmost stack value is used as a jump target. In case of an underflow, more returns than calls, or an overflow, more calls than entries on the stack, the program counter requests a reset of the whole processor. It needs to be pointed out that an interrupt also needs one entry. The size of the stack is a design parameter with the default value 30, but it can be as low as one, thus, only one interrupt or call may happen at the same time.

2.3 Register File

A PauloBlaze can access up to 32 8-bit registers. They are split into two banks to be addressable by a 4-bit halfword. The current bank can be set with the *REGBANK* operation. Only the ALU and the I/O module are capable of altering the registers. The decoder controls the multiplexer selecting whose write enable signal and data is active. The star operation is a special case in which the register file itself changes the content.
To store larger amounts of data, this module contains the Scratchpad Memory (SPM). With the generic *scratch_pad_memory_size* its size can be changed to 64, 128 or 256 bytes. A bigger SPM demands more resources on the device. *STORE* and *FETCH* operations are needed to transfer data between the SPM and the current register bank.

2.4 ALU

The module's primary purpose is to handle arithmetic, logical, shift and rotate operations. They can alter the *zero* and *carry* flags which are handled in the ALU. In case of an interrupt they will be backed up and restored after the interrupt has been handled.
The first clock cycle is used to calculate the results, the second one to write them into the register file and to update the flags. The PauloBlaze is a two address processor. Thus, the ALU writes the result back into the first operands register, overriding it.

2.5 I/O Module

A connection to outside world is established by the I/O module. It can read and write 8-bit words and address up to 256 channels with an 8-bit *port_id*. Even though the user has two clock cycles to present or read the data, using a register before and after the ports is good practice and helps to meet the performance requirements. Various high active strobe signals are asserted during the second cycle of a read or a write operation.

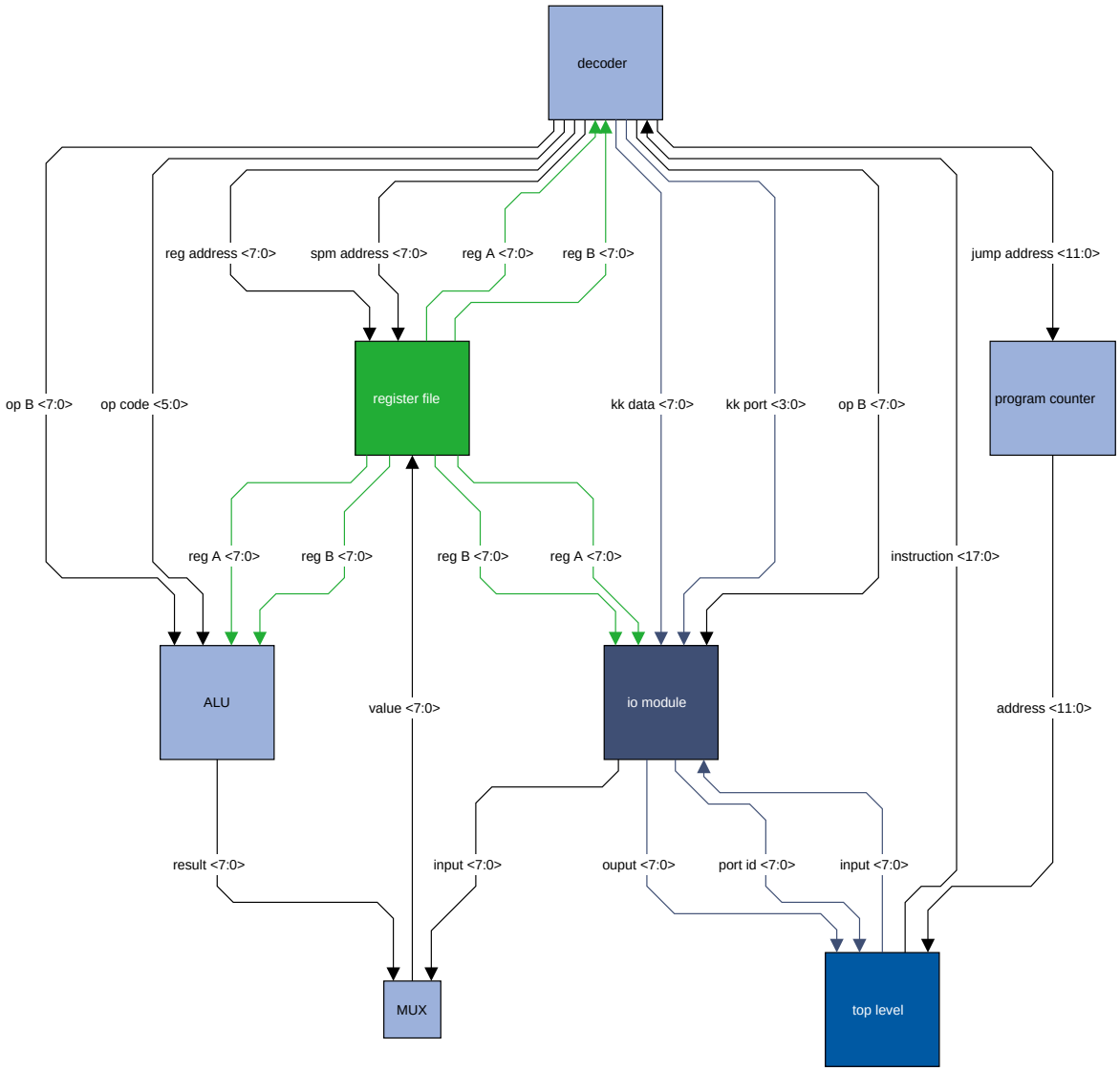

Figure 2.3: Data Flow Graph

3 Evaluation

Implementing a design is only one step in the development, other important steps are verification and evaluation of the result. Even before the evaluation it was clear that a certain performance loss has to be accepted when writing pure VHDL instead of directly describing vendor specific elements, however the PauloBlaze still delivers good performance combined with an acceptable area increase. Those measured values are based on a solution without any significant optimizations to keep the code readable, portable and easily changeable. After all these are the project's main goals and that required a trade-off between them or area and speed.

3.1 Verification

The whole PauloBlaze was simulated and later used in bigger designs to detect possible bugs. To automate the simulation, a self testing program (cf. Appendix A) issues every data chancing instruction and checks the result afterwards. Those checks helped to test jumps and other control flow instructions. After the core passed the simulations, it was deployed into the PicoBlaze-Library [Leh15] and performed well on Xilinx and Altera chips. On the basis of the simulation and the usage in a bigger design, it can be said that there are no known bugs.

3.2 Timing

Several tests were conducted to show the good performance regarding speed, one of the PicoBlazes key aspects. The first test scenario is a simple I/O design based the PicoBlaze manual [Cha12, p. 72] where one can also find the values used in this comparison. The frequency of the PauloBlaze was increased step by step until the Xilinx tool *trace* (? link) resulted in a timing violation. Table 3.1 presents the measurements. It is not surprising to get the same speed from two different 7-Series devices because they share the same architecture. Also, this architecture and the good speed grade fit the implementation best. On the other hand, a Spartan-6 with a low grade is less affected than one with a better grade.

Another test was conducted using the PicoBlaze-Library mentioned in section 3.1. It was augmented and deployed on the Atlys Board [Xil15], featuring a Spartan-6 (speed grad -3). Both, PicoBlaze and PauloBlaze, were able to meet the timing requirements of 100 MHz in this real world example.

Table 3.1: Achievable Speeds with a simple I/O Design

Family	Speed Grade	PicoBlaze	PauloBlaze	Slowdown
Kintex-7	−1	185 MHz	156 MHz	15.7 %
	−3	238 MHz	222 MHz	6.7 %
Virtex-7	−3	232 MHz	222 MHz	4.3 %
Virtex-6	−3	238 MHz	200 MHz	16.0 %
Spartan-6	−1L	82 MHz	74 MHz	9.8 %
	−3	136 MHz	114 MHz	16.2 %

3.3 Resource Usage

Similar to the timing results the PauloBlaze utilizes more resources than its optimized, but restricted, counterpart as shown below with a few examples. The first design was taken from the PicoBlaze Library and implemented on the same Atlys Board. It contains a module called SoFPGA featuring extensions like a divider unit for the single processor. The *Pico constrained* represents the normal PicoBlaze instantiation whereas all the placement instructions were deleted in the *unconstrained* version. Without the directives the tools could place parts of the processor into other modules. Such modules may also occupy a part of a slice, but the usage report assigns the whole slice to the processor. Thus, the resulting utilization values are only estimates. All values were taken after the Xilinx tool *map* had finished.

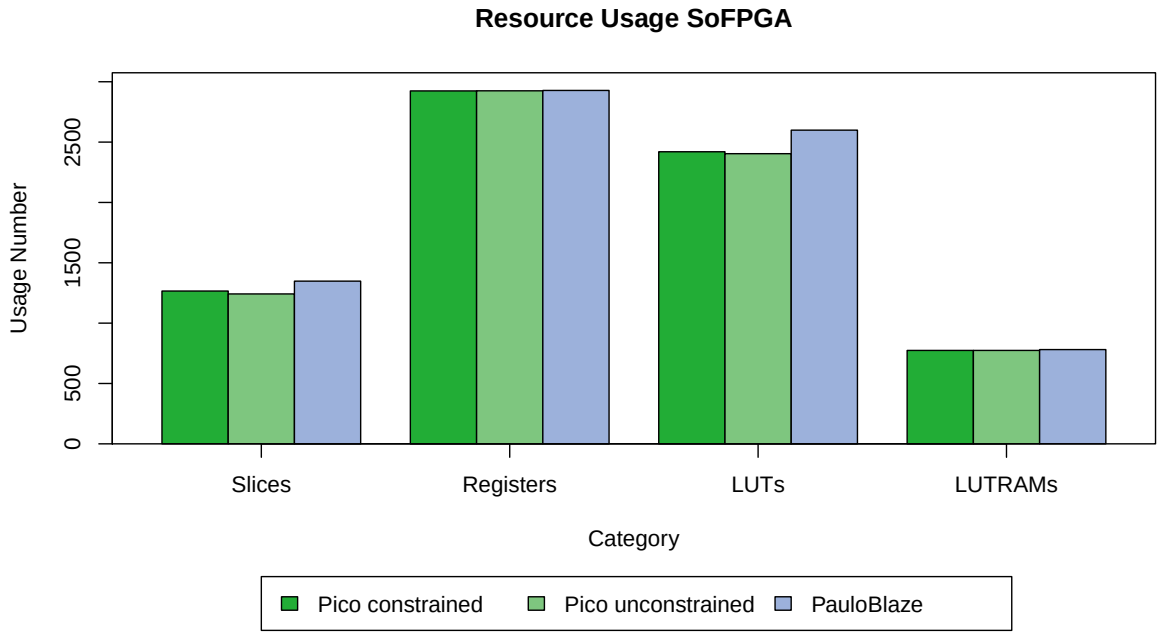

Figure 3.1: Resource Usage of the surrounding SoFPGA Module

The differences between the two PicoBlaze versions seen in figure 3.1 and table 3.2 are most likely caused by the placers behaviour to spread out and use more of the chip when there is

Table 3.2: Resource Usage of the surrounding SoFPGA Module

Resource	Pico unconstrained	Pico constrained	PauloBlaze	increase	
Slices	1266	1242	1348	6.5 %	8.5 %
Slice Register	2924	2925	2928	0.1 %	0.1 %
LUTs	2420	2402	2599	7.4 %	8.2 %
LUTRAM	774	774	781	0.9 %	0.9 %

enough space left. It is not surprising to see an additional demand of resources because of the PauloBlaze. However, the increase, especially to the constrained version, is higher than expected compared to the following measurements.

On basis of the same utilization report a direct look at the single processor is possible. It needs to be pointed out that the report is less precise because of the small number of resources used and the shifting of those between different components.

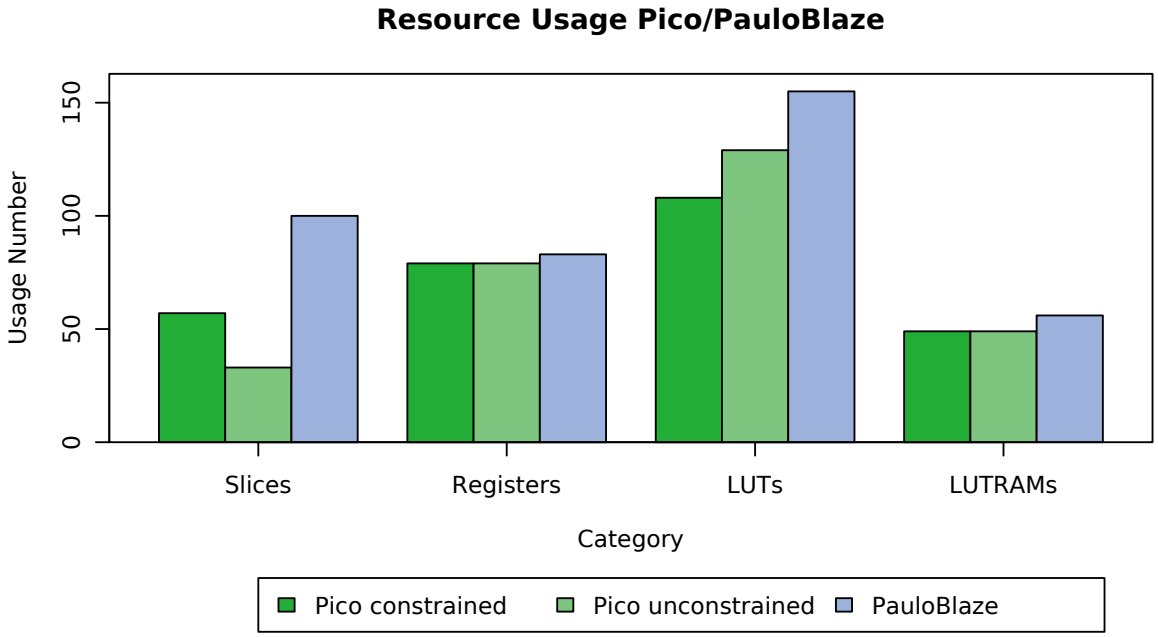

Figure 3.2: Resource Usage of an embedded Pico/PauloBlaze

Table 3.3: Resource Usage of an embedded Pico/PauloBlaze

Resource	Pico unconstrained	Pico constrained	PauloBlaze	increase	
Slices	57	33	100	75.4 %	203.0 %
Slice Register	79	79	83	5.0 %	5.1 %
LUTs	108	129	155	43.5 %	20.2 %
LUTRAM	49	49	56	14.3 %	14.3 %

The shifting of resources from one module to another becomes clear in figure 3.2 when comparing the slice and LUT numbers of the two PicoBlaze versions. On one hand the unconstrained version shares "its" slices with other components almost doubling the usage, on the other hand 17 % of the LUTs are not counted. The PauloBlaze shows an additional slice usage by 43 or 67. This is less than the 100 extra slices suggested by the report of the SoFPGA module.

To get a precise and reasonable result, the timing sections I/O design was tested with different chip families. It is a simple design with only a few extra constructs around the processor. The normal constrained PicoPlaze was used and compared to the PauloBlaze.

Table 3.4: Resource Usage of a Simple I/O Design

Family	PicoBlaze		PauloBlaze		increase	
	S. LUTs	S. Reg	S. LUTs	S. Reg	S. LUTs	S. Reg
Spartan-6	114	84	275	90	141.2 %	7.1 %
Virtex-6	121	115	276	91	128.1 %	−20.9 %
Virtex-7	113	83	283	82	150.4 %	−1.2 %

Although the numbers in table 3.4 are pointing out a significant addition, it is a negligible increase regarding the overall utilization from 1.7 % to 3.1 % on a medium sized Spartan-6 (XC6SLX75).

4 Summary

The primary goal of this work was to create a fully compatible processor to the PicoBlaze. This has been achieved by developing the PauloBlaze, a processor that can replace the other and neither hardware nor software code have to be changed. Even though there are some compromises, the PauloBlaze offers useful features enabling it to compete. The PicoBlaze, a hand optimized design, is smaller than the new alternative. The user has to compensate an increase of up to 150 %, but because of the initially low requirements, the overall resource usage increases only marginally. It is also faster than a PauloBlaze, it varies from a 16.0 % to an only 4.3 % slowdown. If the design does not require maximum speed or if fast 7-Series FPGAs are used, the difference is negligible.

On the other hand the new processor provides the possibility for specific optimization or an easy implementation of new instructions, which may save many clock cycles. Furthermore the designer is not limited to the Spartan-6, Virtex-6 or 7-Series devices, the vendor independent description can be deployed everywhere, as long as the target supports VHDL.

The PauloBlaze trades a small performance penalty for a far more adaptive model, providing flexible designers a very useful tool.

Appendices

A Test Program Listing

```
   columns
 1 CONSTANT A_port, 00
 2 CONSTANT B_port, 01
 3 CONSTANT C_port, 02
 4 CONSTANT D_port, 03
 5 CONSTANT W_port, 01
 6 CONSTANT X_port, 02
 7 CONSTANT Y_port, 04
 8 CONSTANT Z_port, 08
 9
10 start:
11     ENABLE INTERRUPT
12     HWBUILD sF
13     JUMP test_star      ; change to test_pc to test under/overflow
14
15 test_star:
16     LOAD s0, 01
17     STAR s1, s0
18     REGBANK B
19     COMPARE s1, 01
20     JUMP NZ, error
21     REGBANK A
22     JUMP test_add
23
24 test_add:
25     LOAD s0, 01
26     ADD s0, 04
27     COMPARE s0, 05      ; check simple add, 1 + 4 = 5
28     JUMP NZ, error      ; 0 means it's equal
29     LOAD s0, 10
30     LOAD s1, 0D
31     ADD s0, s1
32     COMPARE s0, 1D      ; check 2 register add, 0x10 + 0x0D = 0x1D
33     JUMP NZ, error
34     LOAD s0, 05
35     ADD s0, FB
36     JUMP NC, error      ; check overflow = carry, 5 + 251 = 0 + carry
37     ADD s0, 01
38     COMPARE s0, 01
39     JUMP NZ, error
40     LOAD s0, 00
41     ADD s0, 00
```

```
42      JUMP NZ, error      ; check for zero flag
43      JUMP test_add_carry
44
45  test_add_carry:
46      LOAD s0, FF
47      ADD s0, 01          ; s0 = 0, carry set
48      ADDCY s0, 01            ; s0 = s0 + 1 + carry(1)
49      COMPARE s0, 02
50      JUMP NZ, error      ; s0 is not 2
51      LOAD s0, FF
52      LOAD s1, 01
53      ADD s0, s1          ; s0 = 0, carry set
54      ADDCY s0, s1        ; s0 = s0 + s1(1) + carry(1)
55      COMPARE s0, 02
56      JUMP NZ, error
57      JUMP test_sub
58
59  test_sub:
60      LOAD s0, 0A
61      SUB s0, 0A
62      JUMP NZ, error      ; s0 is supposed to be 0
63      LOAD s0, AB
64      LOAD s1, 0B
65      SUB s0, s1          ; s0 = s0(AB) - s1(0B) = A0
66      COMPARE s0, A0
67      JUMP NZ, error
68      JUMP test_sub_carry
69
70  test_sub_carry:
71      LOAD s0, 00
72      SUB s0, 01          ; s0 = 255, carry set
73      JUMP NC, error
74      SUBCY s0, FE        ; s0 = s0(255) - FE - carry(1) = 0
75      JUMP NZ, error
76      LOAD s0, 00
77      SUB s0, 01
78      LOAD s1, 0A
79      SUBCY s0, s1        ; s0 = s0(255) - s1(10) - carry(1) = 244
80      COMPARE s0, F4
81      JUMP NZ, error
82      JUMP test_logic
83
84  test_logic:
85      LOAD s0, CA
86      AND s0, 53
87      COMPARE s0, 42      ; CA and 53 = 42!! (but it's just a hex 42)
88      JUMP NZ, error
89      LOAD s0, CA
90      LOAD s1, 14
91      AND s0, s1          ; CA and 14 = 0
92      JUMP NZ, error
```

```
 93      LOAD s0, FF
 94      ADD s1, 01          ; carry set
 95      AND s0, 01
 96      JUMP C, error       ; carry was not cleared
 97      LOAD s0, CA         ; -- testing or --
 98      OR s0, 53
 99      COMPARE s0, DB      ; CA or 53 = DB
100      JUMP NZ, error
101      LOAD s0, F0
102      LoAD s1, 0F
103      OR s0, s1           ; F0 or 0F = 0
104      JUMP Z, error
105      LOAD s0, FF
106      ADD s1, 01          ; carry set
107      OR s0, 01
108      JUMP C, error       ; carry was not cleared
109      LOAD s0, CA         ; -- testing xor --
110      XOR s0, 53
111      COMPARE s0, 99      ; CA xor 53 = 99
112      JUMP NZ, error
113      LOAD s0, F0
114      LOAD s1, F0
115      XOR s0, s1          ; F0 or F0 = 0
116      JUMP NZ, error
117      LOAD s0, FF
118      ADD s1, 01          ; carry set
119      XOR s0, 01
120      JUMP C, error       ; carry was not cleared
121      JUMP test_shift
122
123  test_shift:
124      LOAD s0, 7F
125      SL1 s0
126      JUMP C, error
127      COMPARE s0, FF
128      JUMP NZ, error
129      LOAD s0, 80
130      SL0 s0
131      JUMP NZ, error
132      SLA s0
133      COMPARE s0, 01
134      JUMP NZ, error
135      LOAD s0, 11
136      RL s0
137      COMPARE s0, 22
138      JUMP NZ, error
139      LOAD s0, 81
140      SLX s0
141      COMPARE s0, 03
142      JUMP NZ, error
143      LOAD s0, FE
```

```
144     SR1 s0
145     JUMP C, error
146     COMPARE s0, FF
147     JUMP NZ, error
148     LOAD s0, 01
149     SR0 s0
150     JUMP NZ, error
151     SRA s0
152     COMPARE s0, 80
153     JUMP NZ, error
154     LOAD s0, 22
155     RR s0
156     COMPARE s0, 11
157     JUMP NZ, error
158     LOAD s0, 81
159     SRX s0
160     COMPARE s0, C0
161     JUMP NZ, error
162     JUMP test_io
163
164 test_io:
165     LOAD s0, 01
166     LOAD s1, 02
167     LOAD s2, s1
168     LOAD s4, 1E
169     OUTPUT s0, (s2)      ; output value 01 on port 02
170     OUTPUT s1, 03        ; output value 02 on port 03
171     OUTPUTK 03, 4        ; output value 03 on port 04
172     INPUT s3, 05         ; read value on port id 05 into s3
173     INPUT s1, (s4)       ; read value on port id 1E into s1
174     OUTPUT s1, 10        ; output read value on port id 10
175     JUMP test_spm
176
177 test_spm:
178     LOAD s0, 12
179     LOAD s1, 0A
180     LOAD s2, FF
181     STORE s0, (s1)       ; write 12 into addr 0A
182     STORE s1, C3         ; should be addr 03 in a 64 byte spm
183     FETCH s3, (s1)       ; read data 12 back from addr 0A
184     COMPARE s0, s3
185     JUMP NZ, error
186     FETCH s4, 03         ; read data from previously masked addr C3 = 03
187     COMPARE s4, s1
188     JUMP NZ, error
189     JUMP test_call
190
191 inc_s00:
192     ADD s0, 01
193     LOAD s2, 05
194     LOAD&RETURN s2, 07
```

```
195
196 test_call:
197     LOAD s0, 01
198     LOAD s5, inc_s00'upper
199     LOAD s4, inc_s00'lower
200     CALL@ (s5, s4)
201     COMPARE s0, 02
202     JUMP NZ, error
203     CALL Z, inc_s00     ; zero flag still set
204     COMPARE s0, 03
205     JUMP NZ, error
206     HWBUILD s6          ; generate a carry
207     CALL NC, inc_s00
208     COMPARE s0, 03      ; carry still set, s0 should be 03
209     JUMP NZ, error      ; if inc was called (s0 = 04) ... it was wrong
210     CALL C, inc_s00     ; carry set to 0 by compare
211     COMPARE s0, 03      ; call was done ? s0 = 4 -> error
212     JUMP NZ, error
213     LOAD s1, passed'upper
214     LOAD s0, passed'lower
215     JUMP@ (s1, s0)
216
217 test_pc:
218     COMPARE sD, C9      ; random value to switch between over and underflow test
219     JUMP NZ, test_underflow
220     CALL test_overflow
221
222 test_overflow:
223     CALL test_overflow
224
225 test_underflow:
226     LOAD sD, C9
227     RETURN
228
229 error:
230     JUMP error
231
232 passed:
233     JUMP passed
234
235     ADDRESS 300
236 ISR:
237     REGBANK B
238     LOAD s0, FF
239     CALL inc_s00
240     RETURNI ENABLE
```

Bibliography

[Cha12] Ken Chapman. *PicoBlaze for Spartan-6, Virtex-6 and 7-Series (KCPSM6)*. 2012.

[HB10] M.N. Hassan and M. Benaissa. "A scalable hardware/software co-design for elliptic curve cryptography on PicoBlaze microcontroller". In: (May 2010), pp. 2111–2114. DOI: 10.1109/ISCAS.2010.5537064.

[KG05] J. Kadlec and Roger Gook. "Floating-point controller as a picoblaze network on a single spartan 3 FPGA". In: (2005), pp. 1–11.

[Leh15] Patrick Lehmann. *PicoBlaze-Library*. 2015. URL: https://github.com/Paebbels/PicoBlaze-Library.

[Xil11] Xilinx. *PicoBlaze 8-bit Microcontroller*. 2011. URL: http://www.xilinx.com/products/intellectual-property/picoblaze.html.

[Xil15] Xilinx. *XUP Atlys Board*. 2015. URL: http://www.xilinx.com/support/university/boards-portfolio/xup-boards/AtlysBoard.html.

[YS06] Pengyuan Yu and P. Schaumont. "Executing Hardware as Parallel Software for Picoblaze Networks". In: (Aug. 2006), pp. 1–6. DOI: 10.1109/FPL.2006.311237.