Modular transformation of embedded systems from firm-cores to soft-cores

Ehsan Ali and Wanchalerm Pora*

Department of Electrical Engineering,
Chulalongkorn University, Thailand
Email: ehssan.aali@gmail.com
Email: wanchalerm.p@chula.ac.th
*Corresponding author

Abstract: Although there are many 8-bit IP processor cores available, only a few, such as Xilinx PicoBlaze and Lattice Mico8 firm-cores are reliable enough to be used in commercial products. One of the drawbacks is that their codes are confined to vendor-specific primitives. It is inefficient to implement a PicoBlaze processor on non-Xilinx FPGA devices. In this paper we propose a systematic approach that transforms primitive-level designs (firm-cores) to vendor independent designs (soft-cores), while modularising them during the process. This makes modification and implementation of designs on any FPGA devices possible. To demonstrate the idea, our soft-core version of PicoBlaze is implemented on a Lattice iCE40LP1k FPGA device and is shown to be fully compatible with the original PicoBlaze macro. Rigorous verification mechanisms have been employed to ensure the validity of the porting process; therefore, the quality of transformation matches the industry expectation.

Keywords: embedded systems; FPGA; microprocessors; soft-core; firm-core; transformation; HDL; Xilinx PicoBlaze; Lattice; verification.

Reference to this paper should be made as follows: Ali, E. and Pora, W. (2021) 'Modular transformation of embedded systems from firm-cores to soft-cores', *Int. J. Embedded Systems*, Vol. 14, No. 3, pp.259–276.

Biographical notes: Ehsan Ali received his BEng degree in Computer Systems from Assumption University of Thailand in 2015. He is currently a PhD candidate in Electrical Engineering Department of Chulalongkorn University of Thailand. His research interests include data centres, digital circuits, microprocessor design, compiler design, and FPGA-based reconfigurable computing.

Wanchalerm Pora received his BEng and MEng degrees in Electrical Engineering from Chulalongkorn University in 1992 and 1995 respectively. He received his PhD degree from Imperial College, London in 2000. He has been with the Faculty of Engineering, Chulalongkorn University since 1994, and is an Associate Professor at the Department of Electrical Engineering. His research interests are in reconfigurable circuits, intelligent devices and systems for smart grid and healthcare.

1 Introduction

Although there are many 8-bit IP processor cores available, their integrity and reliability can be questioned. Only a few commercial firm-cores such as Xilinx PicoBlaze and Lattice Mico8 are being used by the FPGA community. One of the limitations of these cores is that their HDL source code is locked to vendor-specific primitives. In this paper we propose a systematic approach which transforms primitive-level designs (firm-cores) to vendor independent designs (soft-cores). By modularisation we make design modifications easier and allow the design to be implemented on any FPGA devices. A case study soft-core is implemented on a Lattice iCE40LP1k FPGA device and is shown to be fully compatible with the PicoBlaze macro. Rigorous verification mechanisms have been employed to ensure the validity of the porting process; hence, the quality of transformation matches the industry expectation.

IP cores offered by commercial companies are either closed source or technology dependant (Romero-Troncoso 2006). For example, the source code of PicoBlaze is not in behavioural-level, but in highly optimised Xilinx primitive-level (firm-core). This restricts it to only Xilinx development tools and devices and makes modification of the design impractical.

Moreover, no method suggesting transformation of firm-cores to soft-cores with modularisation concept exists. This situation motivates us to propose a systematic approach that transforms firm-cores to soft-core while retaining the optimisation level and reliability as the main contribution. Our proposed method allows designers to convert protected non-HDL IP cores to a modular HDL

version. The modularity feature enables them to gain deep knowledge of internal structure of the core. Furthermore, major modification, or applying minor changes to the design become feasible. Additionally, the transformed modular soft-core has the advantage of not being locked to a specific FPGA platform or technology, as it uses standard HDL constructs which allows the soft-core product to be synthesised and implemented on all FPGA vendors that support standard HDL constructs. All IP cores that do not implement anti reverse engineering techniques such as physically unclonable functions (PUFs) (Barbareschi and Bagnasco, 2017) can take advantage of our proposed method.

The second minor contribution of this paper is the case study of the proposed method which converts the Xilinx PicoBlaze as a firm-core to Zipi8 which is a soft-core and verifies and implements it on a Lattice device. Required development tools to support the Lattice platform is also provided which can be reused in order to implement Zipi8 on other platforms with no or minimum modifications.

This paper is divided into six sections. First section is the introduction which provides the scope and motivation of the work. Second section mentions background, related work, and 8-bit IP cores review. Some applications of Xilinx PicoBlaze and its architecture are also presented. It then analyses the PicoBlaze source code and provides steps to convert vendor-specific primitives to technology independent VHDL code. Third section explains the modular transformation procedure which is the main contribution of this paper. It provides a modular architectural analysis of PicoBlaze so designers can use it to modify and customise the processor according to project requirements. In Section 4, a verification method is discussed which ensures that the Zipi8 operates exactly the same as the original PicoBlaze. In Section 5, the Zipi8 soft-core is synthesised on a tiny Lattice FPGA device. Meanwhile the necessary memory modification needed to port to Lattice devices is provided. Section 6 concludes the work by comparing the resource utilisation, and advantages of Zipi8 soft-core with related cores.

2 Background

Surprisingly, the 8-bit processors continue to drive the semiconductor industry alongside with their newer 16/32/64/128-bit counterparts since the introduction of Intel 8008 (Morse et al., 1980) until now. For embedded systems, tiny 8-bit processors are the most popular choice. The implementation of an 8-bit processor-based design can be done via two mediums:

1 microcontroller unit (MCU)

2 field-programmable gate array (FPGA).

An MCU is composed of a processor with a limited amount of random access memory (RAM), ROM, timers, I/O Ports, communication ports, etc. All parts are inside a single chip (Mazidi et al., 2016).

In cases that 32-bit accurate computation is not necessary, migration to 8-bit solutions can improve performance and save resources. Nie et al. (2020) show how replacing a 32-bit floating-point multiplication by an 8-bit fixed-point multiplication can save up to 87% of resources, sacrificing only 1% accuracy loss.

An FPGA chip includes input/output, programmable logic (PL) fabric blocks, and routing resources (Chen et al., 2006). FPGAs are being used extensively to cover a broad range of digital applications from simple *glue logic* circuits (Fawcett, 1996), *hardware accelerators* (Possa et al., 2011), to very powerful *system-on-chip* (SoC) platforms (Rodríguez-Andina et al., 2015).

FPGAs have higher level of *flexibility* than MCUs by providing a PL fabric. This for example, allows designers to improve a product after release by upgrading both its hardware and firmware (Makowski, 2013). If *flexibility* in design has highest priority and consequently FPGA approach is chosen, then the next design decision is about the type of processor. FPGA-based embedded processor types are categorised into three groups (Cofer and Harding, 2013):

- *soft-cores* are written in HDL language without extensive optimisation for a target FPGA architecture
- *firm-cores* are also written in HDL language, but have been optimised for a target FPGA architecture
- *hard-cores* are fixed-function gate-level intellectual properties (IPs) within an FPGA fabric.

Hard-cores implemented in SoC chips run faster and consume less power than soft-cores, but their rigid implementation prevents them from being changed for accommodating custom designs. In contrast soft-cores can be adapted easily, and have much higher level of portability (Cofer and Harding, 2013). In many embedded applications, high performance is not of prime concern but required functionality is. A soft-core processor allows designers to add or omit peripherals from the system-on-programmable-chip (SoPC) with ease. A soft-core processor also offers flexibility of configuring the core itself for an application (Nade and Sarwadnya, 2013). At CERN institution, Ammendola et al. (2017) evaluate the performance of a soft processor versus pure VHDL code. They show that the usage of embedded processors could surely lead advantages in the readability of the code, and consequently, contribute to reliability as well as the maintainability of the whole system.

One of the important applications of soft-core processors is in safety-critical real-time embedded systems where designers can take advantage of deterministic timing of soft macros (Romeo et al., 2018). For instance, each instruction of PicoBlaze takes exactly two clock cycles (Chapman, 2014), which ensures deterministic response time to external events and interrupts. Meanwhile, if a project calls for both a microcontroller and FPGA, a soft-core processor can decrease the overall printed circuit board (PCB) footprint, speed up development time, and

permit more flexible redesigns by implementing both on a single chip (Romeo et al., 2018). We also can mention multi-core custom soft processors which can be used in CPU-intensive DSP applications such as image processing tasks (Amiri et al., 2017), or to simply boost parallel applications by using multi-softcore architecture (Baklouti and Abid, 2014).

2.1 Related work

The 8-bit microcontrollers are used in various applications from implementing simple RGB LEDs (Yang, 2010), control applications (Hsu et al., 2009), battery-powered data acquisition (Mukaro and Carelse, 1999), maximum power point tracking (MPPT) (Khan and Hossain, 2010), up to efficient cryptography (Eberle et al., 2005), and implementing TCP/IP stack (Dunkels, 2003).

An SoC platform, or *platform FPGA* (Anvar et al., 2006) is a single chip which accommodates a PL fabric next to fixed-function components such as sophisticated clocking circuitry, phase-locked loops, analogue-to-digital and digital-to-analogue converters (Zanikopoulos et al., 2005), hard-core processors, high-speed hardened peripherals (Ahmad et al., 2016), memory controllers, etc.

Dynamic reconfiguration is a special feature of FPGA devices. For example, different interpolation algorithms for a computer numerical control (CNC) system can be dynamically programmed into FPGA device to lower cost and achieve more functionality (Ni et al., 2017).

There is growing body of research showing that if the critical kernels within a software application is identified and reimplemented on FPGA hardware next to a soft processor, it can compete and even out-perform a hard-core processor (Lysecky and Vahid, 2005). This is achievable by mapping algorithms to FPGA hardware to leverage the inherent parallelism of FPGA devices in an optimal way (Teubner and Woods, 2013). In the near future, many mobile devices will be implemented/delivered on FPGA-based reconfigurable chips (Perera and Li, 2019), which can take advantage of the soft-cores.

FPGAs can exhibit better performance in parallel computing applications such as matrix operations which demand numerous processing elements (PEs) (Wang and Ziavras, 2015). They also can be used as hardware accelerators to speed up the execution (Sharat et al., 2017).

Works related specifically to the PicoBlaze cloning are as follow: Merchant et al. (2006) provide a platform independent implementation of older version of PicoBlaze (KCPSM3) by replacing lookup-tables (LUTs), multiplexers (MUXs), and RAMs, with behavioural HDL models, and then implement it on an Altera device. Their transformed core uses 236 LUTs while the original design uses just 99, which is a 138% increase. There is also no verification mechanism that ensures the reliability of the new core.

The PauloBlaze soft-core written in VHDL exists on github.com that is 100% compatible with instructions set architecture (ISA) of latest version of PicoBlaze (KCPSM6) (Genßler, 2019). This design uses 276 LUTs, and 91 flip-flops (FFs) on a Xilinx Vortex-6 device while the original PicoBlaze uses 121 LUTs, and FFs. That is 128% increase in LUTs and –20.9% decrease in FFs. Their verification method is based on simulating a test program, unfortunately this is not a sufficient verification mechanism.

The authors of this paper observed discrepancies between the core and the PicoBlaze by conducting a more thorough verification. A testbench which puts PicoBlaze and PauloBlaze alongside of two block RAMs holding exact copy of a test program was implemented. A test program with several calls to routines of an IEEE 754 floating point library (Ali and Pora, 2020) was executed. The clock accuracy comparison was skipped, and only the final calculation results were recorded and then compared. The experiment yielded numerous discrepancies that denounce the integrity of PauloBlaze.

The PacoBlaze (Kocik, 2007) is another behavioural Verilog clone of KCPSM3 firm-core. There is no official resource utilisation of PacoBlaze reported by either the original author (Kocik, 2007) or third parties. Therefore, the authors of this paper had to synthesise and implement the design on a Spartan6 device using Xilinx ISE 14.7. The report obtained from our synthesis yields a utilisation of 158 LUTs, 8 MUXs, 30 FFs. In conclusion, there is no reliable soft-core version of latest PicoBlaze (KCPSM6) available.

2.2 Embedded system 8-bit IP cores review

In embedded systems the resources are scarce and that prompts designers to use tiny 8-bit processors in their designs. A thorough search was conducted to identify all available 8-bit IP cores. The result is categorised into three groups:

1 commercial product (Tong et al., 2006)

2 academic work

3 individual project.

Table 1 shows all notable 8-bit IP cores available as of writing this article. We have omitted those academic works that their HDL source code could not be found in public domain. Additionally, individual projects which have no proper documentation or were simply duplication of other designs were also excluded.

The highest priority in deciding which core to use is the reliability factor. Cores written by individuals or developed in academia are less reliable than commercial products which enjoy larger community, alongside a support team that continuously fix reported bugs, and release updates. Moreover, commercial cores are supported by more mature development tools (simulator, compiler, debugger, etc.), and provide more extensive documentation. For example, we tested PauloBlaze (Genßler, 2019), which is a plain VHDL implementation of PicoBlaze, and is hosted on GitHub website. We observed that under specific circumstances the processor produces wrong result. This prompts us to exclude unreliable individual/academic projects.

Table 1 8-bit IP processor cores, sorted alphabetically

No.	*Name*	*Author/company, year*	*Instr. set*	*Source code*	*Instr. width*	*CPI*
1	Core8051*	Microsemi (2019)	Intel MCS-51	Verilog VHDL	1-3 B	1-11
2	DP80390*	Digital Core Design (2019)	Intel MCS-51	Verilog VHDL	1-3 B	2-3
3	DRPIC16*	Digital Core Design (2019)	PIC 16XXX	Verilog VHDL	14-bit	1-2
4	G.P. 8-bit RISC†	Zavala et al. (2015)	G.P. 8-bit RISC	Verilog VHDL	16-bit	2-3
5	HCS08*	Silvaco (2019)	Freescale MC9S08xx	Verilog	1-4 B	2-6
6	L8051XC1*	CAST Inc. (2019)	Intel MCS-51	Verilog VHDL	1-3 B	4/6/12
7	M8051EW M8051W*	Silvaco (2019)	Freescale MC9S08xx	Verilog	1-4 B	2-6
8	MCL51*	MicroCore Labs (2019)	Intel MCS-51	Encrypted Verilog	1-3 B	1-4
9	MCL65*	MicroCore Labs (2019)	NMOS 6502	Encrypted Verilog	1-3 B	2-7
10	Mico8*	Lattice Semi (2017)	Mico8	Verilog RTL	18-bit	2
11	MiniMIPS†	Cesar (2011)	MIPS	VHDL	16-bit	1
12	Natalius‡	Guzman (2012)	Natalius	Verilog	16-bit	3
13	Navré‡	Bourdeauducq (2013)	Atmel AVR	Verilog	16-bit	1.7
14	Open8 uRISC‡	Hays (2016)	V8-uRISC	VHDL	1-3 B	1-7
15	pAVR‡	Cuturela (2009)	Atmel	VHDL	16-bit	1.7
16	PicoBlaze*	PicoBlaze	AVR PicoBlaze	Primitive level	18-bit	2
17	risc8‡	Coonan (2016)	PIC16C5X	Verilog	12-bit	2-4
18	ZA-SUA†	Santa et al. (2018)	ZA-SUA	Verilog	17-bit	4

Notes: *Commercial product: the RTL (or behavioural level) source code is not freely available.
†Academic work: might provide more reliability, and design integrity.
‡Individual project: lacks rigorous testing with high probability of having hidden bugs.

The Xilinx company, the inventor of FPGA technology, has the highest FPGA market share (Ahmed, 2018). This naturally makes their community larger than others and consequently their 8-bit IP core which is named PicoBlaze to be more reliable. Other commercial products such as Mico8, DP80390, HCS08, etc. are also viable options, but this should be considered that sometimes when a smaller company is acquired by a larger one their products might get discontinued, and all support tools and documentation become outdated or inaccessible. For example, the RISC V8-uRISC core (VAutomation, 1998) got disappeared after ARC International acquisition of *VAutomation* (ARC, 2002).

Fortunately there is an open-source implementation of it named Open8 uRISC on public domain (Hays, 2016).

Many cores listed in Table 1 are based on Intel MCS-51 (Wharton, 1980) which is a complex instruction set computer (CISC). Others are based on reduced instruction set computer (RISC) architectures like PIC16 and MIPS. As Jamil (1995) points out, several studies show that 25% of the instructions belonging to an ISA make up 95% of the total execution time. This observation justifies the adaptation of RISC in 8-bit IP cores.

2.3 PicoBlaze applications

Antonio-Torres et al. (2009) used the PicoBlaze in embedded systems for ‘monitoring applications’, Ivanov (2015) has employed the processor to provide a controller for traffic light, Zaykov (2007) has constructed a multiprocessor parallel architecture based on message passing paradigm using multiple PicoBlaze cores, Mandala (2011) has studied the usage of the PicoBlaze in ‘multiprocessor systems’, and Mattson (2004) has implemented a network interface using the PicoBlaze.

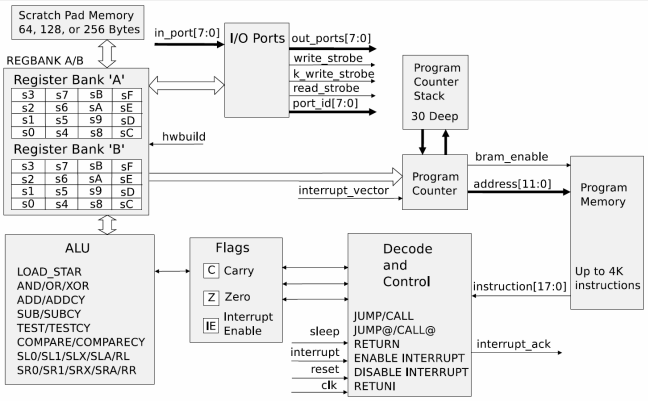
Claudiu et al. (2012) have implemented ‘smart sensor using multiple cores’ of PicoBlaze. Borawake and Chilveri (2014) have used PicoBlaze to implement a ‘wireless sensor

network'. PicoBlaze has been used as a 'configuration engine' in a fault-tolerance technique by Pham et al. (2013). Hassan and Benaissa (2009) have implemented a scalable elliptic curve cryptography (ECC) on PicoBlaze. Good and Benaissa (2006) have used PicoBlaze for 'advanced encryption standard' (AES). This body of literature justifies the usage of 8-bit soft-core processors such as PicoBlaze in a broad range of applications.

2.4 PicoBlaze architecture

In this section the details of PicoBlaze architecture will be provide. As shown in Figure 1, the 18-bit instruction fetched from data bus of program memory (up to 4KB supported) has two bit-fields as shown in Table 2. The 6-bit opcode provides up to 64 instructions, which PicoBlaze utilises 55 of them. This makes room for 9 instructions to be added in the future. The operands field can have just one or a mixture of the following fields: '*aaa*, *kk*, *pp*, *p*, *ss*, *x*, *y*' as shown in Table 2.

Figure 1 KCPSM6 architecture and features



Source: Chapman (2014)

For example, the 'JUMP aaa' instruction is encoded to '22aaa' hex value which 22 is the opcode and aaa is the 12-bit jump address, or 'LOAD sX, sY' is encoded to '00xy0' which 00 is the opcode and 4-bit x is destination register, and 4-bit y is source register. There is a scratch pad memory (SPM) with maximum size of 256 bytes which can be used as data memory.

Table 2 PicoBlaze instruction bit-fields

Opcode (6-bit)	*Operands (12-bit)*	
6-bit always	*aaa*	12-bit address 000 to FFF
	kk	8-bit constant 00 to FF
	pp	8-bit port ID 00 to FF
	p	4-bit port ID 0 to F
	ss	8-bit scratch pad location 00 to FF
	x	4-bit register within bank s0 to sF
	y	4-bit register within bank s0 to sF

Source: Chapman (2014)

The PicoBlaze has three flags: carry (C), zero (Z), and interrupt enable (IE). There are 256 input and 256 output ports, and a stack with the depth of 30. There is an interrupt pin that forces the processor to execute code which resides in the interrupt service routine (ISR) with a predefined memory address location, and a sleep pin which freezes all operations (Chapman, 2014).

3 PicoBlaze to Zipi8 transformation

3.1 PicoBlaze source-code analysis

The 'very high speed integrated circuit (VHSIC), Hardware Description Language (VHDL), and Verilog Hardware Description Language (Verilog-HDL)' (Smith, 1996) are two industry standard Hardware Description Languages (HDL) (IEEE 1076-2008, 2009; IEEE 1364-2005, 2006). The PicoBlaze core is provided in both VHDL and Verilog languages. We choose VHDL version to take advantage of having a very strongly typed language model (Smith, 1996).

FPGA primitives are the basic building blocks of a design. They perform dedicated functions, implement standards for I/O pins in devices, and their names are standard (AN 307, 2018). We propose three steps in order to analyse a design completely:

1 Primitive analysis: To scan the code for all primitives used in the design. The list of all primitives used in PicoBlaze is as follow: 'LUT6, LUT6_2, FD, FDR, FDRE, XORCY, MUXCY, RAM32M, RAM256X1S'.

2 Primitive definitions: To study the FPGA manufacturer library guide to retrieve the detailed functionality of each primitive, and then write a VHDL implementation of it accordingly. In our case, the 'Xilinx 7 Series FPGA Libraries Guide' (Xilinx UG799, 2011) provides the detailed behaviour of each primitive.

3 Modularisation: To draw the schematic of LUTs, MUXs, and FFs, and combine combinational logics (CLs) that is implemented using LUTs into independent modules. All primitives between FFs which contribute to FF excitation equation should be packed into a module. The module name can be chosen based on internal signal names. For example, a module that produces carry and zero flag can be named as 'flags'.

In next section, we will provide an equivalent vendor-independent VHDL code for all primitives used in the design.

3.2 Primitive conversion to technology independent VHDL

In this section, all primitives used in PicoBlaze firm-core are scanned and identified. Then an equivalent technology independent VHDL version of them is proposed to replace the primitives. By doing this, we have essentially transformed the firm-core nature of the processor to soft-core and converted the design into a more

self-explanatory state. This opens up the possibility for designers to be able to modify the design and retarget it to other platforms.

Table 3 provides the summary of the VHDL approaches adapted in transformation process.

Table 3 Summary of the primitive conversion to technology independent VHDL

#	*Primitive*	*Conversion method*
1	LUT6	Espresso minimiser yields a continuous assignment equation
2	LUt6_2	Espresso minimiser yields two continuous assignment equations
3	FD	VHDL process sensitive to rising edge of clock
4	FDR	VHDL process sensitive to rising edge of clock and reset signal
5	FDRE	VHDL process sensitive to rising edge of clock, reset, and enable (CE) signals
6	XORCY	Continuous assignment with equation: Out <= A x or B
7	MUXCY	VHDL process sensitive to 3-inputs (2-inputs and 1 selector)
8	RAM32M	One port VHDL array with synchronised write, asynchronous read
9	RAM256X1S	One port VHDL array with synchronised write, asynchronous read

3.2.1 LUT6, and LUT6_2: 6-input lookup table

Both design elements are 6-input look-up table (LUT). LUT6 has 1-output, and LUT6_2 has 2-outputs. They can either act as asynchronous 64-bit ROM (with 6-bit addressing) or implement any 6-input logic function. LUTs are the basic logic building blocks and are used to implement most logic functions of the design (Xilinx UG799, 2011). The LUT6 primitive in PicoBlaze is used only to implement combination logic (CL). Listing 1 shows an example of PicoBlaze LUT6 instance. The 'pc_mode2_lut' is instance name, and X 'FFFFFFFF00040000' is a 64-bit hexadecimal constant used as initial value of LUT6 primitive. I0, I1, I2, I3, I4, and I5 are inputs, and O is output.

Listing 1 An example of PicoBlaze LUT6 primitive instantiation

```
pc_mode2_lut: LUT6
  generic map (INIT => X"FFFFFFFF00040000")
  port map (I0 => instruction(12)
    I1 => instruction(14)
    I2 => instruction(15)
    I3 => instruction(16)
    I4 => instruction(17)
    I5 => active_interrupt
    O => pc_mode(2))
```

We first perform Boolean minimisation on the 6-input logic function using the given 64-bit LUT value. The minimisation method can be either manual or automated using algorithms such as Espresso logic minimiser (McGeer et al., 1993). In above example, the minimised function is shown in (1).

$$O = I5 + I4.I3.I2.I1.10 \quad (1)$$

After replacing the I0, I1, I2, I3, I4, I5, and O variables in (1) with the name of signals connected to them, we get the exact equivalent vendor independent VHDL implementation of LUT6 which is shown in Listing 2.

Listing 2 An example of VHDL Implementation of LUT6 primitive

```
pc_mode(2) <= (active_interrupt or
  instruction(17) and
  (not instruction(16)) and
  (not instruction(15)) and
  instruction(14) and
  (not instruction(12))
```

The case for LUT6_2 is similar except that the lower 32-bit LUT value is used for first, and the full 64-bit of the same shared value is used for the second output. For example, if X"7777027700000200" is the LUT6_2 value, then for O5 pin output, the value X"00000200" is used, and for O6 pin output, the value X"7777027700000200" is used.

3.2.2 FD: D FF, and its variants: FDR, FDRE

This design element is a D-type FF. The data on input is loaded into the FF during the Low-to-High clock transition (Xilinx UG799, 2011). Listing 3 shows an example of PicoBlaze FD instance. The 'alu_mux_sel0_flop' is the instance name, D is input, Q is output, and C is clock.

Listing 3 An example of PicoBlaze FD primitive instantiation

```
alu_mux_sel0_flop: FD
  port map (D => alu_mux_sel_value(0)
    Q => alu_mux_sel(0)
    C => clk)
```

The vendor independent VHDL code for FD primitive is shown in Listing 4.

Listing 4 General VHDL implementation of FD primitive

```
flipflops_process: process (C) begin
  if rising_edge(C) then
    Q <= D;
  end if;
end process flipflops_process;
```

Replacing C, Q, and D with the name of connected signals will yield the final equivalent vendor independent VHDL code for FD primitive as shown in Listing 5.

Listing 5 An example of VHDL implementation of FD primitive

```
flipflops_process: process (clk) begin
if rising_edge(clk) then
alu_mux_sel(0) <= alu_mux_sel_value(0);
end if;
end process flipflops_process;
```

The design elements FDR and FDRE are D-type FF with synchronous reset, and clock enable, and synchronous reset respectively. FDR has an extra R pin used for resetting the FF, and FDRE in addition to a synchronous reset has a CE pin used as clock enable signal. Listing 6 shows the vendor independent VHDL implementation of these primitives.

Listing 6 General VHDL implementation of FDR and FDRE primitives

```
-- FDR
flipflops_R_process: process (C) begin
  if rising_edge(C) then
   if (R = '1') then
     Q <= '0';
   else
     Q <= D;
   end if;
  end if;
end process flipflops_R_process;

-- FDRE
flipflops_R_CE_process: process (C) begin
  if rising_edge(C) then
   if (R = '1') then
     Q <= '0';
   elsif CE = '1' then
     Q <= D;
   end if;
  end if;
end process flipflops_R_CE_process;
```

3.2.3 XORCY: XOR gate, and MUXCY: 2-to-1 multiplexer

The XORCY is a special XOR with general output that generates faster and smaller arithmetic functions. It is a dedicated XOR function within the carry-chain logic of FPGA slice. It allows for fast and efficient creation of arithmetic (add/subtract) or wide logic functions (large AND/OR gate) (Xilinx UG799, 2011); the MUXCY is a simple 2-to-1 Multiplexer (Xilinx UG799, 2011).

Listing 7 shows an example of PicoBlaze XORCY, and MUCY instances. For XORCY, the 'arith_carry_xorcy' is the instance name, LI, and CI are inputs, O is output. For MUXCY, the 'parity_muxcy' is the instance name, DI, and CI are inputs, S is selector, and O is multiplexer output. If S is low then DI drives the O, and if S is high then CI drives the O output.

Listing 7 An example of PicoBlaze XORCY and MUXCY primitives instantiation

```
arith_carry_xorcy: XORCY
  port map (LI => '0',
            CI => carry_arith_logical(7),
            O => arith_carry_value);
parity_muxcy: MUXCY
  port map (DI => lower_parity,
            CI => '0',
            S => lower_parity_sel,
            O => carry_lower_parity);
```

Listing 8 General VHDL implementation of XORCY primitive

```
-- XORCY
O <= LI xor CI;

-- MUXCY
muxcy_process: process (S, DI) begin
  case S is
    when '0' => O <= DI;
    when '1' => O <= CI;
    when others => O <= 'X';
  end case;
end process muxcy_process;
```

3.2.4 RAM32M, RAM256X1S: multi port random access memories (select RAM)

These design elements are multi-port, RAM with synchronous write and asynchronous independent read capability. RAM32M is a 32-bit deep by 8-bit wide, and RAM256X1S is a 256-bit deep by 1-bit wide (Xilinx UG799, 2011).

Listing 9 shows an example of PicoBlaze RAM32M instance. The 'stack_ram_low' is the instance name, INIT_A, INIT_B, INIT_C, INIT_D define initial RAM values, DIA, DIB, DIC, DID, are data input, DOA, DOB, DOC, DOD, are data output, ADDRA, ADDRB, ADDRC, ADDRD, are read address bus, WE is write enable, and WCLK is write clock. All writes are synchronous, while all reads are asynchronous. The RAM32M can have several configurations. PicoBlaze uses this primitive as a 32x8 single port RAM by connecting ADDRX pins to the same signal (stack_pointer).

Listing 9 An example of PicoBlaze RAM32M primitive instantiation

```
stack_ram_low : RAM32M
  generic map (
   INIT_A => X"0000000000000000",
   INIT_B => X"0000000000000000",
   INIT_C => X"0000000000000000",
   INIT_D => X"0000000000000000")
  port map (
   DOA(0) => stack_carry_flag,
   DOA(1) => stack_zero_flag,
   DOB(0) => stack_bank,
   DOB(1) => stack_bit,
     DOC => stack_memory(1 downto 0),
     DOD => stack_memory(3 downto 2),
    ADDRA => stack_pointer(4 downto 0),
    ADDRB => stack_pointer(4 downto 0),
    ADDRC => stack_pointer(4 downto 0),
    ADDRD => stack_pointer(4 downto 0),
   DIA(0) => carry_flag,
   DIA(1) => zero_flag,
   DIB(0) => bank,
   DIB(1) => run,
     DIC => pc(1 downto 0),
     DID => pc(3 downto 2),
      WE => t_state(1),
    WCLK => clk);
```

The vendor independent VHDL code for RAM32M primitive is shown in Listing 10. The general 'ram' VHDL module is defined in 'ram.vhd' file. In order to have a 32x8 RAM the depth and width of memory is set through generic parameters: 'DATA_WIDTH' is set to 8, and 'ADDRESS_WIDTH' is set to 5. Note that DIA, DIB, DIC, DID, are all 2-bit signals which are combined into 8-bit DI signal.

Similarly, DOA, DOB, DOC, DOD, are all 2-bit signals which are combined into 8-bit DO. In PicoBlaze design, ADDRA, ADDRB, ADDRC, ADDRD are all connected to a shared bus (e.g., stack_pointer), therefore we combine all of them into ADDR signal.

Similar approach can be taken in order to convert RAM256X1S primitive except that 'DATA_WIDTH' is set to 1, and 'ADDRESS_WIDTH' is set to 8.

Listing 10 An example of PicoBlaze RAM32M primitive instantiation

```
-- General ram module defined in ram.vhd file
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ram is
  generic (DATA_WIDTH : positive;
            ADDRESS_WIDTH : positive);
  port (WCLK : in std_logic;
         WE : in std_logic;
         DI : in std_logic_vector (DATA_WIDTH-1 downto 0);
        ADDR : in std_logic_vector (ADDRESS_WIDTH-1
        downto 0);
        DO : out std_logic_vector (DATA_WIDTH-1 downto 0)
        );
end ram;

architecture Behavioural of ram is
  type ram_type is array ((2**ADDR'length) - 1 downto 0) of
     std_logic_vector(DI'range);
  signal ram_s : ram_type := others=> (others=>'0'));
  begin
-- Synchronous write, asynchronous read
  RamProc: process(WCLK) begin
     if rising_edge(WCLK) then
       if WE = '1' then
        ram_s(to_integer(unsigned(ADDR))) <= DI;
       end if;
     end if;
end process RamProc;

-- Asynchronous read
   DO <= ram_s(to_integer(unsigned(ADDR)));
end Behavioural;

-- RAM32M instantiation
stack_ram_low: ram
   generic map (DATA_WIDTH => 8, -- 32 x 8-bit RAM
               ADDRESS_WIDTH => 5)

   port map (WCLK => clk,
                WE => t_state(1),
                DI => data_in_ram_low,
              ADDR => stack_pointer,
                DO => data_out_ram_low);
```

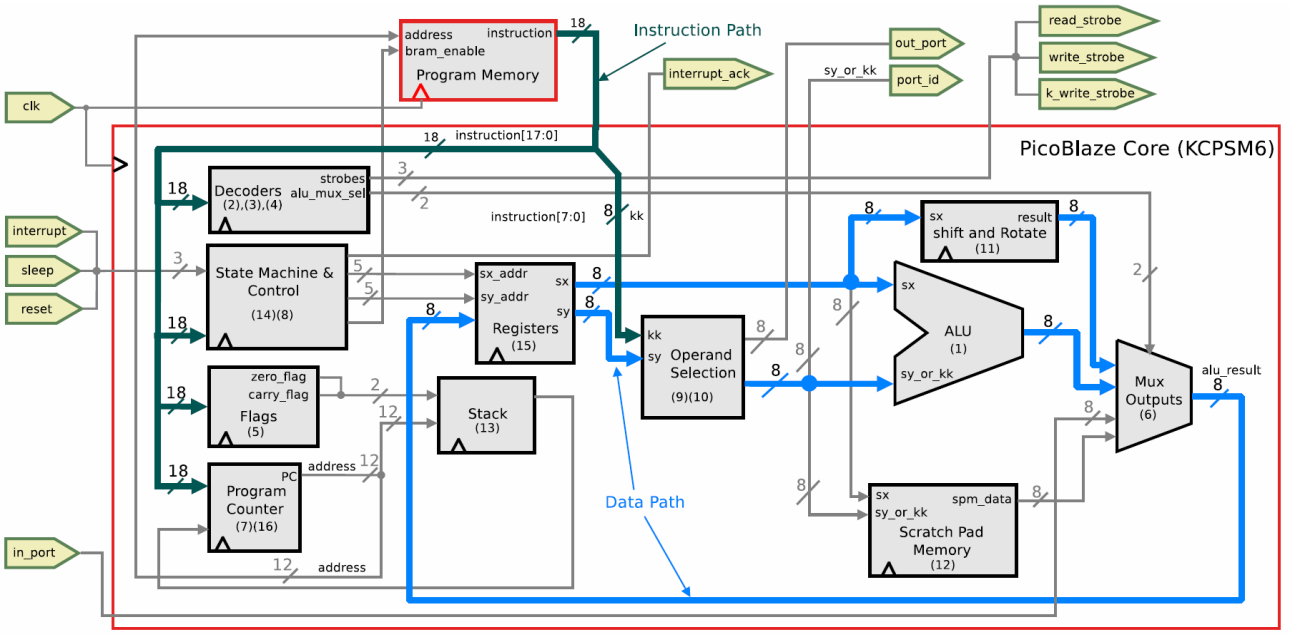
3.3 PicoBlaze conversion using modular approach

The PicoBlaze VHDL source code has no modular structure. It is a single module in a single VHDL file with long list of primitive instances, and signals that connect them. To port the design from firm-core to soft-core it is enough to directly replace all the instances with vendor independent VHDL equivalent codes mentioned in previous section as done by Merchant et al. (2006).

Table 4 List of PicoBlaze modules

No.	*Name*	*No.*	*Name*
1	arith_and_logic_operations	9	sel_of_2nd_op_to_alu_and_port_id
2	decode4alu	10	sel_of_out_port_value
3	decode4_pc_statck	11	shift_and_rotate_operations
4	decode4_strobes_enables	12	spm_with_output_reg
5	flags	13	stack
6	mux_outputs_from_alu_spm_input_ports	14	state_machine
7	program_counter	15	two_banks_of_16_gp_reg
8	register_bank_control	16	x12_bit_program_address_generator

Figure 2 Modular PicoBlaze architecture (Zipi8) (see online version for colours)



By grouping the related primitives into isolated modules, and then perform the transformation we can achieve two goals:

1 to handle the complexity and minimise the human errors

2 to reveal the internal architecture of design which makes modification easier.

The available comments in source code, and primitive instance, and signal names are used to divide the PicoBlaze core into 16 modules. Each module resides in a separate VHDL file with .vhd file extension, the filenames are exactly the same as module names. All modules involved in constructing the PicoBlaze core are listed in Table 4.

The modules, and important signals and buses which connect them are shown in Figure 2. The schematic is the simplified version of a complete and detailed one and is provided in Appendix A. To simplify the diagram occasionally two or three related modules combined as submodules. This is indicated by mentioning module numbers in parentheses inside rectangles. Both program memory and the processor share the same global clock signal. Those modules which are synchronous to the clock are marked with triangular symbol. The absence of clock symbol indicates a pure CL clock (e.g., 'operand selection').

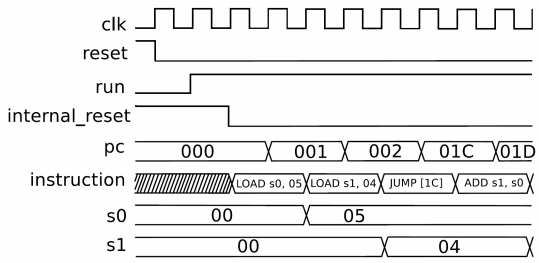
3.4 Zipi8 architecture

The important paths such as 'data path' and 'instruction path' are shown in Figure 2. The allocation of two separate buses connected to two different memory blocks indicates a Harvard Architecture (Furber, 1989). In order to explain the execution cycle of PicoBlaze we go through the sample program (Figure 3).

Listing 11 PicoBlaze sample program

```
Start at 000:
LOAD s0, 05 ; Load 05 into register s0
LOAD s1, 04 ; Load 04 into register s1
JUMP subprogram_at_01c
...
subprogram_at_01c:
ADD s1, s0; s1 <= s1 + s0
```

Figure 3 PicoBlaze instruction cycle



The de-assertion of reset signal puts the processor into *run* state. In this state the processor waits for the first clock transition from low to high to occur, which triggers a fetch instruction from location 0x000 of program memory. The fetch makes the 'instruction path' bus to hold valid data (In our example, it is the first instruction: LOAD s0, 05).

The *instruction* bus is connected to FFs in 'decoders', 'state machine and control', 'flags', and 'program counter' modules. When the second clock cycle occurs the instruction is decoded (sx_addr is set to 0 to select register s0, and 05 constant value is held on *instruction[7:0]* marked as *kk* field); next state of machine is calculated; flags are set, and finally program counter (PC) is incremented by 1.

In clock cycle #3 the instruction at location 0x001 is fetched, and at the same time the result of ALU is written back into register, which results s0 to hold value 05. Next clock fetches instruction at location 1 (LOAD s1, s0). Similarly decode and execute happens in next clock cycle which sets sx_addr to 1 and prompts second ALU operand (*kk*) to hold constant value 04. Next clock cycle writes back the result into register bank, which results s1 to hold value 04, and at the same time fetches the next instruction (JUMP subprogram_at_01c).

Next clock cycle decodes the JUMP instruction and instead of 'PC + 1', the PC is set to value 0x01C which is the jump target location. Next clock cycle fetches the instruction at location 0x01C of program memory (ADD s1, s0), and then one cycle later, it decodes it and finally at next clock cycle the ALU result of addition of 5 + 4 which is 9 is written back into the register s1, and so on.

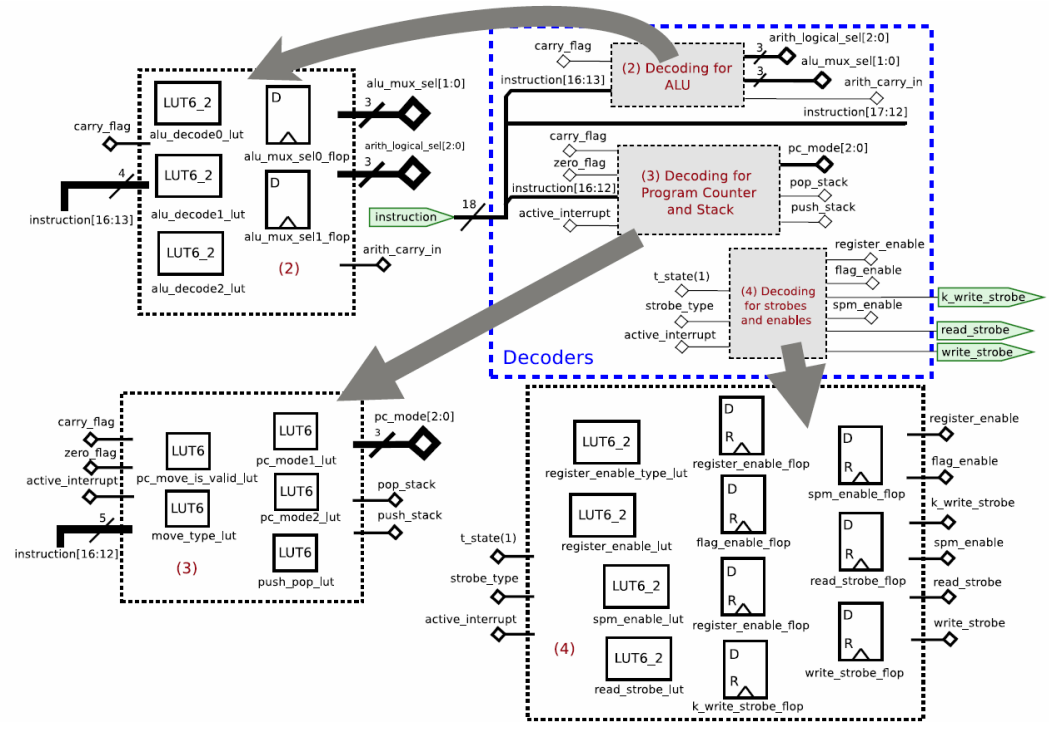
Each PicoBlaze instruction takes exactly two clock cycles to execute which makes its performance deterministic. This turns PicoBlaze into a suitable candidate for safety-critical real-time embedded systems (Romeo et al., 2018).

3.5 *Zipi8 modules' schematic*

In this section, we discuss the correlation between the simplified module in Figure 2 and its full version provided in Appendix A. This helps readers to identify modules, their input/output ports, and in-sheet connections easier.

Figure 4 demonstrates how primitives are grouped into modules. The 'decoders' module is provided as an example. The blue dashed line rectangle marks the 'decoders' module which is a virtual one as it does not have a module number, and therefore there is no corresponding VHDL file. It merely groups three modules which their functionality is related to one another (decoding) under one umbrella. Inside 'decoders' we can see three sub-modules: '(2) decoding for ALU, (3) decoding for program counter and stack, (4) decoding for Strobes and enables'.

Figure 4 PicoBlaze decoder modules with input/output ports, grouped primitives, and in-sheet connections (see online version for colours)



These modules have a corresponding VHDL source file with the exact same name. For example, under the Zipi8 project folder there is a VHDL file named 'decode4alu.vhd' which corresponds to '(2) Decoding for ALU' module depicted in Figure 4. The *instruction* signal bus is an input port to PicoBlaze, and *k_write_strobe* is a PicoBlaze output port (both PicoBlaze input/output ports marked with green colour). The *instruction[16:13]*, and *carry_flag* are inputs, and *alu_mux_sel[1:0]*, *arith_logical_sel[2:0]*, and arith_carry_in are outputs of the module '(2) Decoding for ALU'. The squares rotated by 45-degrees indicate in-sheet local connection.

3.5.1 Zipi8 performance and resource utilisation

Table 5 shows the resource utilisation of Zipi8 soft-core versus others using Vivado v2018.3 (64-bit) synthesis tool for an UltraScale+ architecture. It can be seen that after original PicoBlaze firm-core (123 LUTs), the Zipi8 has the lowest LUT count, it also uses 10 registers less, and consumes no carry and MUX primitives.

Table 5 Core Utilisation Comparison on Xilinx ZYNQ UltraScale+ Device (ZCU104 Board)

Module	*LUTs*	*Registers*	*Carry4/8*	*F7 Muxes*	*F8 Muxes*
PicoBlaze (KCPSM3)	163	74	10	0	0
PacoBlaze (KCPSM3)*	157	31	0	0	8
PicoBlaze (KCPSM6)	123	76	7	16	8
PauloBlaze (KCPSM6)	315	80	12	0	0
Zipi8 (KCPSM6)	143	66	0	0	0

Notes: *Xilinx ISE WebPACK 14.7 was used, synthesised for Spartan6 XC6SLX4 device

Particularly, the main usage of 8-bit soft-cores is in implementing state machines or control applications and not high-performance scientific calculations. Therefore, the core performance (maximum clock frequency) has less importance than resource utilisation (core compactness). Therefore, stating the maximum achievable clock frequencies in Table 5 is omitted. The maximum achievable clock frequency for each core is device dependant. Every FPGA has a specific *speed grade* that determines the maximum clock frequency of designs. For example, the original PicoBlaze achieves up to 105MHz in a Spartan-6 (–2 speed grade) and up to 238MHz can be achieved in a Kintex-7 (–3 speed grade) device (Chapman, 2014).

In the case of Zipi8, the authors of this paper could achieve a clock frequency of 333MHz on XCZU7EV chip with speed grade –2. The 333MHz is the speed limit in the FPGA 'lower power domain clock' which feeds the 'PL fabric clock'. Designers can use the mixed-mode clock manager (MMCM) to generate clock frequency more than 333MHz and push the Zipi8 performance even further.

4 Zipi8 verification

4.1 Verification concepts

Verification is the process of determining that a model implementation accurately represents the developer's conceptual description of the model and the solution to the model (Thacker et al., 2004; AIAA, 2014). Verification can be classified into:

a *Code verification:* To identify and eliminate programming and implementation errors within the software

b *Calculation verification:* To quantify the error of a numerical simulation or in other words 'numerical error estimation' (AIAA, 2014).

A widely used approach in code verification is the comparison method in which one code is compared to another established code (Knupp and Salari, 2002).

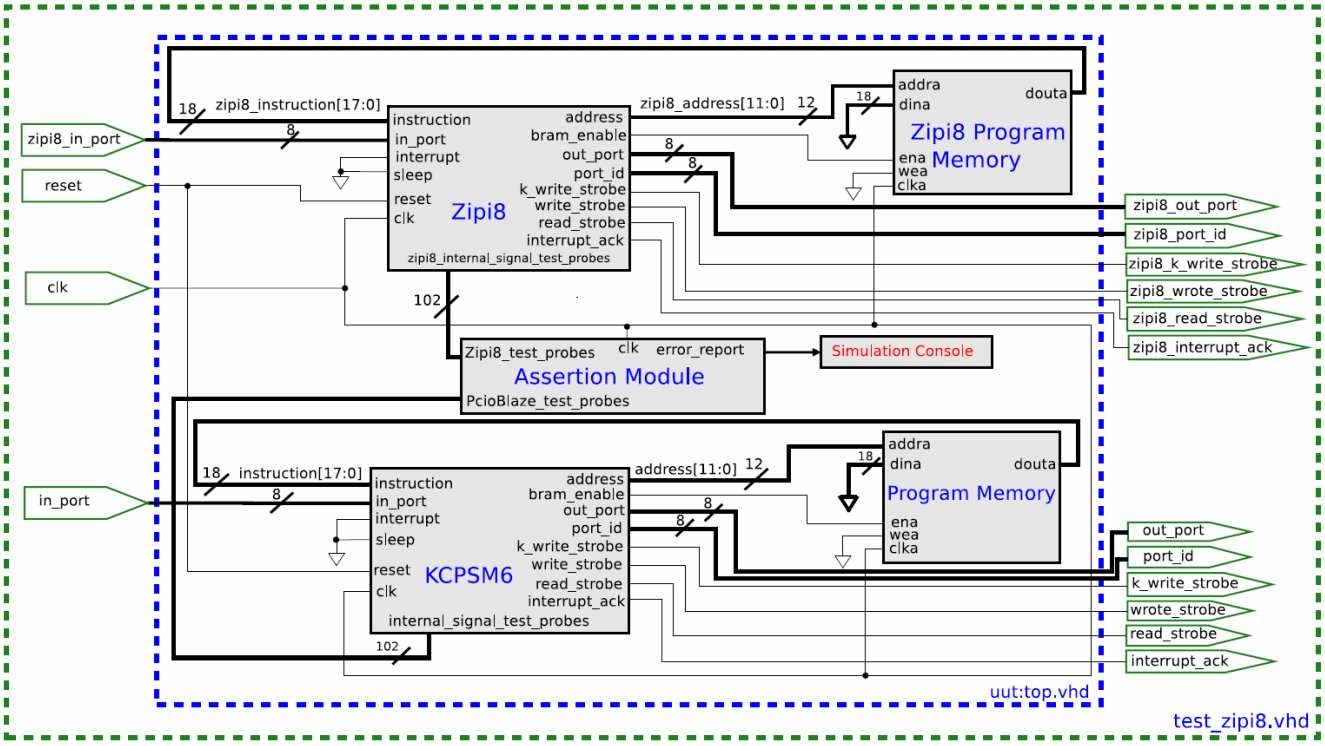
After firm-core to soft-core transformation, we can use comparison method to verify the integrity of Zipi8 by comparing the state of all Zip8 signal buses to PicoBlaze on every clock cycle. Here we use the *concept* of *comparison method* by taking advantage of this fact that the Xilinx PicoBlaze is an establish design, and we can compare our proposed design (Zipi8) against it. The first step in comparison is to take the fully designed and implemented Zipi8 soft-core and probe *all* its internal signals; in parallel, as there is a one to one relationship between internal signals of both cores, the associated signals in PicoBlaze are also probed.

Next, we compare these two sets of signals (coming out of both processor cores) against each other in every clock cycle. As both cores execute the same test program synchronously, their internal states and bus values change accordingly, which gives us the opportunity to look for any discrepancies. Next section gives details of this verification mechanism.

4.2 Comparison method verification mechanism

Figure 5 shows the details of testbench that is used for verification process. The VHDL simulation module 'test_zipi8.vhd' instantiates the 'top' module as unit under test (UUT). The top module consists of two block RAM modules, both holding an exact copy of a PicoBlaze program.

Figure 5 Zipi8 integrity verification: VHDL simulation Testbench (see online version for colours)



The PicoBlaze program resides in those BRAMs is automatically generated by a tool developed by the authors of this paper. We have developed the tool using C++ language to generate random instructions based on a pool (instruction pool class in Figure 6).

All classes used in our random program generator tool are shown in Figure 6. The *InstructionPool* class instantiates 51 instructions and returns a random instruction whenever its *getRandomInst*() method is called. The *Instruction* class represents a PicoBlaze instruction and has opcode, and operands fields as its data members. The *generate_ops*() is a function member of *Instruction* class that calls the toss() method of *Operand* class to generate random values for each operand.

This allows generation of instructions randomly and then assigns arbitrary values to their operand(s). The instruction pool does not contain the jump and subroutine instructions (CALL and RETURN variations) as they are associated with labels and modify the PC value. The random placement of these instructions will disrupt the normal flow of the program.

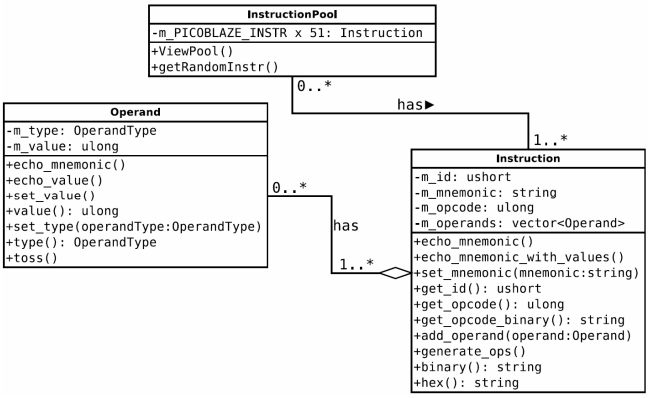
For example, a randomly generated RETURN instruction in the first location of program memory simply causes a stack overflow; a random CALL instruction with a randomly generated target address might set the PC register to data section of the program and forces the processor to execute data, instead of code which puts the processor into unknown and unpredictable state. Therefore, instead of automatic generation, a test program is written manually to test those instructions.

As we mentioned in Section 3.3, the Zipi8 has 16 modules. We probe the output of all these 16 modules (102 signals in total) and compare it against the corresponding signals in KCPSM6 using VHDL assert simulation command. The assert statements are synchronised with clocks, and they check the validity of all 102 signals in every clock cycle. We use VHDL alias command for assigning short names to internal signals which run down into hierarchy of modules. Listing 12 shows a sample of VHDL code for probing one of those 102 signals. The Vivado project that contains the complete VHDL simulation source code is provided as supplementary material to this paper in Appendix B.

Listing 12 VHDL verification: signal assertion

```
test_internal_signals: process (uut_clk)
  alias zipi8_run is
   << signal uut.processor_zipi8.state_machine_i.run : std_logic
   >>;
   alias kcpsm6_run is
   << signal uut.processor_kcpsm6.run : std_logic >>;
begin
   if rising_edge(uut_clk) then
    assert (zipi8_run = kcpsm6_run)
    report “zipi8_run internal signal mismatch @ “ &
      integer’image (now / 1ns) & “ ns” severity failure;
   end if;
end process;
```

Figure 6 PicoBlaze random program generator classes



In conjunction with above method a second verification mechanism is employed to verify the Zipi8 integrity. In this method, a VHDL process is defined that prompts both Zipi8, and KCPSM6 cores to dump the 18-bit hex value of the instruction under execution into two separate files on every clock cycle. We then use *byte comparison* to find out the existence of any discrepancy in simulation dumped files. The absence of any discrepancies, and assertion failure affirms this conclusion: 'Zipi8 is a PicoBlaze compatible soft-core and it is as reliable as the original version'.

Listing 13 VHDL verification: instruction dump

```
instruction_seq_dump : process(uut_clk)
  -- open file: "zipi8_instructions.txt" in write_mode;

  file file_handler : text;
  variable outline : line;
  variable file_is_open: boolean := false;
begin
  if not file_is_open then
    file_open (file_handler, "zipi8_instructions.txt",
    write_mode);
    file_is_open := true;
  end if;

  if rising_edge(uut_clk) then
   if(zipi8_reset = '0') then
    hwrite(outline, "00" & zipi8_instruction);
    writeline(file_handler, outline);
   end if;
  end if;
end process instruction_seq_dump;
```

Listing 13 shows the VHDL process in the second part of the simulation code that dumps the instructions executed by Zipi8 into 'zipi8_instructions.txt'. The instructions executed by KCPSM6 are obtained when we convert the test program source code to .hex file in the assembling process (PicoBlaze assembler automatically *dumps* a .hex file) For example if the test program is saved in ABC.psm source file then issuing the assembler with ABC.psm as input, will output the ABC.hex file which contains all the KCPSM6 instructions. We can then change the extension ABC.hex to ABC.txt, and then perform *byte comparison* against zipi8_instructions.txt file to find potential discrepancies.

5 PicoBlaze on lattice

5.1 Synthesis utilisation result

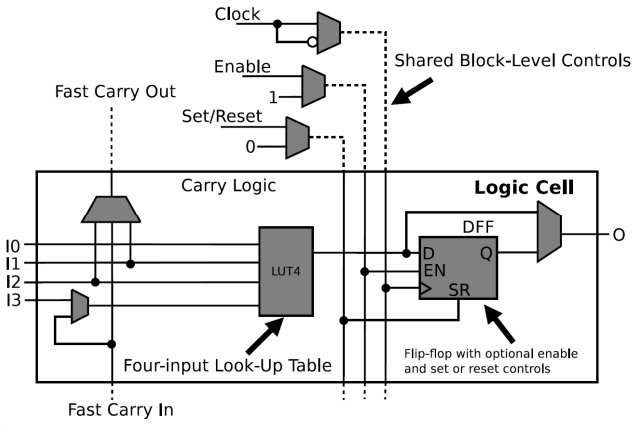
This section provides proof of concept by synthesising Zipi8 and implementing it on a Lattice FPGA device (Lattice Semi, 2019). The Lattice iCEcube2 version 2017.08.27940 is used as project manager, and 'Synplify Pro L-2016.09L+ice40, Build 077R, Dec 2 2016' is used as synthesis tool. The complete source code and project files are provided in Appendix C.

Table 6 shows the resource utilisation reported by Synplify Pro for Lattice iCE40LP1K after synthesising and mapping the Zipi8. The most important count is LUT4 consumption. Table 6 shows that for Zipi8, 'distribution of all consumed LUTs' is 642 (SB_LUT4). Synthesis of PicoBlaze using Vivado v2018.3 (64-bit) for a ZYNQ UltraScale+ device utilises 143 LUTs.

Table 6 Zipi8 resource utilisation on Lattice iCE40LP1K

Cell usage	*Count*
DFF variation	322
Logic cell	642 of 1280 uses (50%) (190 inferred register)
SB_RAM2048x2	9 uses
SB_RAM256x16	2 uses
Block Rams:	11 of 16 (68%)

Figure 7 Lattice logic cell



Source: Lattice Semi (2017)

The reason for an increase in LUT count is that UltraScale+ devices provide LUTs with 5- and 6-inputs, while Lattice iCE40 series devices equipped with only 4-input LUTs. Additionally, the synthesis tool fails to map a memory block to Lattice technology specific RAM primitive and maps it to 256 individual registers instead. A programmable logic block (PLB) in Lattice device consist of an LUT4 and a D flip-flop (DFF) as shown in Figure 7 (Lattice Semi, 2017). Therefore, 256 DFF automatically increases the LUT4 count which must be considered. This consequently makes the final LUT count for Zipi8 on the Lattice to be 642 − 256 = 382.

5.2 Lattice RAM blocks

The PicoBlaze macro uses RAM elements in order to implement SPM, stack, and internal registers. These modules (plus the program memory) with their depth and width are listed in Table 7. It is up to synthesis tool, and its user settings to infer memory clock elements, therefore, we refrain from converting general parametrised RAM blocks to Lattice RAM blocks.

Table 7 Zipi8 modules with parametrised memory block

Zipi8 module	*Depth*	*Width*
two_banks_of_16_gp_reg	32	8
spm_with_output_reg	256	8
stack	32	16
program_memory	4096	18

5.2.1 Program memory

A 4KB block RAM with width of 18-bit must be connected to PicoBlaze as ‘program memory’. Xilinx devices provide 9-bit RAM blocks which makes it very efficient to construct program memory by simply grouping 2 block RAMs next to each other ($2 \times 9bit = 18bit$). Lattice devices do not provide 9-bit wide block RAMs, therefore forcing designers to construct an 18-bit wide block RAM using other combinations. Lattice iCE40LP1K has 16 Memory Block of type RAM4k. Each 4k memory block can be used in a variety of depths and widths such as: ‘256x16 (4K)’, ‘512x8 (4K)’, ‘1024x4 (4K)’, ‘2048x2 (4K)’ (Lattice Semi, 2017).

Instead of standard 4KB program memory, we construct a 2KB program memory by grouping 9 instances of SB_RAM2048x2 primitive ($9 \times 2bit = 18bit$), and leave the rest of memory blocks used in Zipi8 (such as memory blocks used as register banks, stack, and SPM) to synthesis tool to infer (they will be inferred into either FF primitives or block RAMS).

Due to this change in program memory structure the original Xilinx assembler fails to generate the correct VHDL template for program memory. Therefore, a new tool is developed in C++ language which receives PicoBlaze program in .hex format, and outputs a .vhd file as PicoBlaze program memory template which can be directly imported into the project without any modification.

The tool takes advantage of INIT_0 (to INIT_F) directives to set initial values of Lattice RAM RAM4K primitives to initialise the memory blocks. These initial values are read from .hex file and inserted into 9 separate instances of SB_RAM2048x2 in a .vhd file. The complete C++ source code of this tool is provided in Appendix D. Researchers, and designers can be inspired by looking into the approach used in our tool to facilitate development of their own tools if they need to implement Zipi8 on other FPGA platforms.

Finally, as shown in Table 6, Zipi8 uses 11 out of 16 block RAMs available on the Lattice device. 9 uses of SB_RAM2048x2 is directly instantiated in program memory module, 1 use of SB_RAM256x16 is inferred to map ‘stack’, and 1 use of SB_RAM256x16 is to map ‘spm_with_output_reg’ (SPM). Synplify Pro is unable to map block memory in ‘two_banks_of_16_gp_reg’. The reason is that the RAM block defined there mimics the behaviour of Xilinx primitives which allows ‘Synchronous Write, Asynchronous Read, with separate read/write address bus’, while Lattice RAM primitives do not provide this feature (Sync-Async R/W). Listing 14 shows the difference in VHDL implementation of RAM block for Lattice devices which has a subtle difference with Listing 10 which is the VHDL implementation of RAM block for Xilinx devices.

Listing 14 General VHDL implementation of RAM32M primitive with separate R/W

```
-- General ram module defined in ram.vhd file
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ram_rw is
  generic (DATA_WIDTH : positive;
       ADDRESS_WIDTH : positive);
  port (WCLK : in std_logic;
       WE : in std_logic;
       DI : in std_logic_vector (DATA_WIDTH−1 downto 0);
       ADDR_RD : in std_logic_vector
       (ADDRESS_WIDTH−1 downto 0);
       ADDR_WR : in std_logic_vector
       (ADDRESS_WIDTH−1 downto 0);
       DO : out std_logic_vector (DATA_WIDTH−1 downto
       0) );
end ram_rw;

architecture Behavioural of ram32m_rw is
  type ram_type is array ((2**ADDR_RD'length) −1 downto
  0)
    of std_logic_vector(DI'range);
  signal ram_s_RD_WR : ram_type := (others=>
  (others=>'0'));
begin
```

```
-- Synchronous write, asynchronous read with
-- separate R/W
RamProc: process(WCLK) begin
if rising_edge(WCLK) then
  if WE = '1' then
    ram_s_RD_WR(to_integer(unsigned(ADDR_WR))) <=
    DI;
  end if;
end if;
end process RamProc;

DO <= ram_s_RD_WR(to_integer(unsigned(ADDR_RD)));

end Behavioural;
```

6 Conclusions

In this paper a systematic approach is presented to transform firm-core designs to soft-core ones. The proof of concept is demonstrated by porting Xilinx PicoBlaze firm-core to a soft-core, named 'Zipi8'. It is then implemented on a tiny Lattice FPGA device. This new macro is vigorously tested, in order to be sure that it is fully compatible with the original firm-core. The method proposed in this paper improves *flexibility* with a slight change in resource consumption on a Xilinx FPGA. PicoBlaze core consumes 123 LUTs, 76 registers, and 25 MUXes; whereas Zipi8 consumes only 139 LUTs, 66 registers, and no MUXes. The LUT count on Lattice device is 382, a three-fold increase due to lack of 5- and 6-input LUT primitives. As future work, the proposed method can be scripted with the primitive's definition knowledge from FPGA vendor library guides.

Acknowledgements

This research is supported financially by 'the Chulalongkorn Academic Advancement into Its 2nd Century Project'. We would like to thank Prof. Ekachai Leelarasmee and Asst. Prof. Kittiphan Techakittiroj for their academic advices and continual encouragement. The student is awarded a joint scholarship, composed of The 100th Anniversary Chulalongkorn University Fund for Doctoral Scholarship and The 90th Anniversary of Chulalongkorn University, Rachadapisek Sompote Fund.

Appendices/supplementary materials are available on request by emailing the corresponding author or can be obtained under https://github.com/ehsan-ali-th/firmcore_to_softcore_appendices.

References

Ahmad, S., Boppana, V., Ganusov, I., Kathail, V., Rajagopalan, V. and Wittig, R. (2016) 'A 16- nm multiprocessing system-on-chip field-programmable gate array platform', *IEEE Micro*, Vol. 36, No. 2, pp.48–62 [online] https://doi.org/10.1109/MM.2016.18.

Ahmed, O. (2018) *Latest FPGAs in the Market*, COEN 6501 – Digital Design and Synthesis [online] http://users.encs.concordia.ca/asim/COEN_6501/Lecture_Notes/FPGA%20Report.pdf (accessed 23 November 2019).

Ali, E. and Pora, W. (2020) 'Implementation and verification of IEEE-754 64-bit floating-point arithmetic library for 8-bit soft-core processors', *8th International Electrical Engineering Congress (iEECON)*, pp.1–4 [online] doi: https://10.1109/iEECON48109.2020.229455.

American Institute of Aeronautics & Astronautics (2014) *Guide for the Verification and Validation of Computational Fluid Dynamics Simulations*, (AIAA G-077-1998(2002)) [online] https://doi.org/10.2514/4.472855.001.

Amiri, M., Siddiqui, F.M., Colm, K., Woods, R. and Rafferty, K. and Bardak, B. (2017) 'FPGA-based soft-core processors for image processing applications', *Journal of Signal Processing Systems*, Vol. 87, No. 1, pp.139–156, ISSN 1939-8115 [online] https://doi.org/10.1007/s11265-016-1185-7.

Ammendola, R., Barbanera, M., Bizzarri, M., Bonaiuto, V., Ceccucci, A., Checcucci, B., De Simone, N., Fantechi, R., Federici, L., Fucci, A., Lupi, M., Paoluzzi, G., Papi, A., Piccini, M., Ryjov, V., Salamon, A., Salina, G., Sargeni, F. and Venditti, S. (2017) 'Performance and advantages of a soft-core based parallel architecture for energy peak detection in the calorimeter Level 0 trigger for the NA62 experiment at CERN', *Journal of Instrumentation*, Vol. 12, No. 03 [online] https://doi.org/10.1088/1748-0221/12/03/C03054.

AN 307 (2018) *Intel® FPGA Design Flow for Xilinx Users*, Updated for Intel® Quartus® Prime Design Suite: 17.1 [online] https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/ an307.pdf.

Antonio-Torres, D., Villanueva-Perez, D., Canepa, E. and Meraz, N.S. (2009) 'A PicoBlaze-based embedded system for monitoring applications', *International Conference on Electrical, Communications, and Computers*, pp.173–177 [online] https://doi.org/10.1109/CONIELECOMP.2009.49.

Anvar, S., Gachelin, O., Kestener, P., Le Provost, H. and Mandjavidze, I. (2006) 'FPGA-based systemon- chip designs for real-time applications in particle physics', *IEEE Transactions on Nuclear Science*, June, Vol. 53, No. 3, pp.682–687.

ARC (2002) *International Completes Integration of Three Subsidiaries Into One Company*, 17 June [online] https://www.design-reuse.com/news/3409/arc-international-integrationsubsidiaries- into-one-company.html (accessed 23 November 2019).

Baklouti, M. and Abid, M. (2014) 'Multi-softcore architecture on FPGA', *International Journal of Reconfigurable Computing*, Vol. 2014 [online] https://doi.org/10.1155/2014/979327.

Barbareschi, M. and Bagnasco, P. (2017) 'Implementation of a reliable mechanism for protecting IP cores on low-end FPGA devices', *International Journal of Embedded Systems*, Vol. 9, No. 4 [online] https://doi.org/10.1504/IJES.2017.086135.

Borawake, S.M. and Chilveri, P.G. (2014) 'Implementation of wireless sensor network using microblaze and picoblaze processors', *Fourth International Conference on Communication Systems and Network Technologies*, pp.1059–1064 [online] https://doi.org/10.1109/CSNT.2014.216.

Bourdeauducq, S. (2013) 'OpenCores project', *Navre AVR clone (8-bit RISC)* [online] https://opencores.org/projects/navre (accessed 11 November 2019).

CAST Inc. L8051XC1: Legacy-Configurable 8051-Compatible Microcontroller IP Core [online] http://www.cast-inc.com/ip-cores/8051s/l8051xc1/index.html (accessed 15 November 2019).

Chapman, K. (2014) *PicoBlaze for Spartan-6, Virtex-6, 7-Series, Zynq and Ultra Scale Devices (KCPSM6) - Release 9*, Xilinx, Release 9, Xilinx Inc., USA.

Chen, D., Cong, J. and Pan, P. (2006) 'FPGA design automation: a survey', *Foundations and Trends in Electronic Design Automation*, Vol. 1, No. 3, pp.139–169, http://dx.doi.org/10.1561/1000000003.

Claudiu, L., Sebastian, S. and Cristian, B. (2012) 'Smart sensor implemented with PicoBlaze multi-processors technology', *IEEE 18th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, pp.241–245 [online] https://doi.org/10.1109/SIITME.2012.6384384.

Cofer, R.C. and Harding, B.F. (2013) 'Chapter 14 – embedded processing cores in rapid system prototyping with FPGAs', *Accelerating the Design Process – Embedded Technology*, pp.185–209, ISBN 9780750678667 [online] https://doi.org/10.1016/B978-075067866-7/50015-9.

Coonan, T. (2016) *GitHub Project*, risc8 [online] https://github.com/brabect1/risc8 (accessed 20 November 2019).

Cuturela, D. (2009) 'OpenCores project', *pAVR: 8 Bit Controller that is Compatible with Atmel's AVR Architecture* [online] https://opencores.org/projects/pavr (accessed 11 November 2019).

Digital Core Design, DP80390 – High performance MCU for Applications Requiring Code Space Larger than 64 kB. [online] https://www.dcd.pl/product/dp80390/ (accessed 17 November 2019).

Digital Core Design, DRPIC166X [online] https://www.dcd.pl/product/drpic166x/ (accessed 15 November 2019).

Dunkels, A. (2003) 'Full TCP/IP for 8-bit architectures', *Proceedings of the 1st international conference on Mobile systems, applications and services (MobiSys '03)*, pp.85–98, ACM, New York, NY, USA, http://dx.doi.org/10.1145/1066116.1066118.

Eberle, H., Wander, A., Gura, N., Chang-Shantz, S. and Gupta, V. (2005) 'Architectural extensions for elliptic curve cryptography over GF(2/sup m/) on 8-bit microprocessors', *IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)*, pp.343–349 [online] https://doi.org/10.1109/ASAP.2005.15.

Fawcett, B. (1996) 'FPGAs as reconfigurable processing elements', *IEEE Circuits and Devices Magazine*, Vol. 12, No. 2, pp.8–10 [online] https://doi.org/10.1109/101.485906.

Furber, S.B. (1989) *VLSI Risc Architecture and Organization*, 1st ed., CRC Press, ISBN 9780824781514.

Genßler, P.R. (2019) *GitHub.com Project*, PauloBlaze [online] https://github.com/krabo0om/pauloBlaze (accessed 11 November 2019).

Good, T. and Benaissa, M. (2006) 'Very small FPGA application-specific instruction processor for AES', *IEEE Transactions on Circuits and Systems I: Regular Papers*, Vol. 53, No. 7, pp.1477–1486 [online] https://doi.org/10.1109/TCSI.2006.875179.

Guzman, F. (2012) 'OpenCores project', *Natalius 8 bit RISC* [online] https://opencores.org/projects/natalius_8bit_risc (accessed 17 November 2019).

Hassan, M.N. and Benaissa, M. (2009) 'Embedded software design of scalable low-area elliptic-curve cryptography', *IEEE Embedded Systems Letters*, Vol. 1, No. 2, pp.42–45 [online] https://doi.org/10.1109/LES.2009.2034708.

Hays, K.I. (2016) 'OpenCores project', *Open8 uRISC* [online] https://opencores.org/projects/open8_urisc (accessed 20 November 2019).

Hsu, C-F., Chung, I-F., Lin, C-M. and Hsu, C-Y. (2009) 'Self-regulating fuzzy control for forward DC-DC converters using an 8-bit microcontroller', IET *Power Electronics*, Vol. 2, pp.1–12 [online] https://doi.org/10.1049/iet-pel:20070179.

IEEE Standard for Verilog Hardware Description Language (2006) IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001) [online] https://doi.org/10.1109/IEEESTD.2006.99495.

IEEE Standard VHDL Language Reference Manual (2009) IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002) [online] https://doi.org/10.1109/IEEESTD.2009.4772740.

Ivanov, V.N. (2015) 'Using a PicoBlaze processor to traffic light control', *Cybern. Inf. Technol.*, Vol. 15, No. 5, pp.131–139 [online] https://doi.org/10.1515/cait-2015-0023.

Jamil, T. (1995) 'RISC versus CISC - why less is more', *IEEE Potentials*, Vol. 14, No. 3, pp.13–16 [online] https://doi.org/10.1109/45.464688.

Khan, S.A. and Hossain, M.I. (2010) 'Design and implementation of microcontroller based fuzzy logic control for maximum power point tracking of a photovoltaic system', *International Conference on Electrical & Computer Engineering (ICECE 2010)*, pp.322–325 [online] https://doi.org/10.1109/ICELCE.2010.5700693.

Knupp, P. and Salari, K. (2002) *Verification of Computer Codes in Computational Science and Engineering*, 1st ed., Chapman and Hall/CRC, ISBN 9781584882640.

Kocik, P.B. (2007) *PauloBlaze* [online] http://bleyer.org/pacoblaze (accessed 27 November 2019).

Lattice Mico8 Open, Free Soft Microcontroller [online] http://www.latticesemi.com/Products/DesignSoftwareAndIP/IntellectualProperty/ IPCore/IPCores02/Mico8.aspx (accessed 11 November 2019).

Lattice Semiconductor (2017) *iCE40TM LP/HX Family Data Sheet*, DS1040 Version 3.4 [online] http://www.latticesemi.com//media/LatticeSemi/Documents/-DataSheets/iCE/iCE40LPHXFamilyDataSheet.pdf (accessed 20 November 2019).

Lattice Semiconductor [online] https://www.latticesemi.com/en (accessed 13 November 2019).

Lysecky, R. and Vahid, F. (2005) 'A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning', *Design, Automation and Test in Europe*, Vol. 1, pp.18–23 [online] https://doi.org/10.1109/DATE.2005.38.

Makowski, D. et al. (2013) 'Firmware upgrade in xTCA systems', *IEEE Transactions on Nuclear Science*, Vol. 60, No. 5, pp.3639–3646, October [online] https://doi.org/10.1109/TNS.2013.2275073.

Mandala, V. (2011) *A Study of Multiprocessor Systems using the Picoblaze 8-bit Microcontroller Implemented on Field Programmable Gate Arrays*, Electrical Engineering Theses [online] http://hdl.handle.net/10950/59 (accessed 20 November 2019).

Mattson, R. (2004) *Evaluation of PicoBlaze and Implementation of a Network Interface on a FPGA*, Student thesis, Electrical Engineering, Linköping University [online] http://liu.divaportal.org/smash/record.jsf?pid=diva2%3A19730&dswid=-9283 (accessed 20 November 2019).

Mazidi, M.A., Causey, D. and McKinlay, R. (2016) *PIC Microcontroller and Embedded Systems: Using Assembly and C for PIC18*, 2nd ed., MicroDigitalEd, ISBN-10 099792599X, ISBN-13 978-0997925999.

Mazidi, M.A., Gillispie Mazidi, J. and McKinlay, R.D. (2007) *The 8051 Microcontroller and Embedded Systems using Assembly and C*, 2nd ed., Prentice Hall.

McGeer, P.C., Sanghavi, J.V., Brayton, R.K. and Sangiovanni-Vicentelli, A.L. (1993) 'ESPRESSO-SIGNATURE: a new exact minimizer for logic functions', *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 1, No. 4, pp.432–440 [online] https://doi.org/10.1109/92.250190.

Merchant, F., Pujari, S. and Patil, M. (2006) 'Platform Independent 8-bit Softcore for SoPC', *Proceedings of the International Multi-Conference of Engineers and Computer Scientists*, Vol. 2, p.2175.

MicroCore Labs, MCL51 Core [online] http://www.microcorelabs.com/mcl51.html (accessed 21 November 2019).

MicroCore Labs, MMCL65 6502 Core [online] http://www.microcorelabs.com/mcl65.html (accessed 21 November 2019).

Microsemi IP Module - Core8051 [online] http://soc.microsemi.com/products/ip/search/detail.aspx?id=541 (accessed 17 November 2019).

Morse, R., Ravenel, B.W., Mazor, S. and Pohiman, W.B. (1980) 'Intel microprocessors – 8008 to 8086', *Computer*, Vol. 13, No. 10, pp.42–60 [online] https://doi.org/10.1109/MC.1980.1653375.

Mukaro, R. and Carelse, X.F. (1999) 'A microcontroller-based data acquisition system for solar radiation and environmental monitoring', *IEEE Transactions on Instrumentation and Measurement*, Vol. 48, No. 6, pp.1232–1238 [online] https://doi.org/10.1109/19.816142.

Nade, J.B. and Sarwadnya, R.V. (2015) 'The soft core processors: a review', *International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering (IJIREEICE)*, Vol. 3, No. 12, pp.197–203 [online] https://doi.org/1010.17148/IJIREEICE.2015.31241.

Ni, X., Zhang, H., Wang, D. and Luo, J. (2017) 'Implementation of dynamic reconfigurable interpolator for open architecture CNC by using FPGA', *International Journal of Embedded Systems*, Vol. 9, No. 1 [online] https://doi.org/10.1504/IJES.2017.081727.

Nie, Z., Li, Z., Wang, L., Guo, S., Deng, Y., Rangyu and Dou, Q. (2020) 'Laius: an energy-efficient FPGA CNN accelerator with the support of a fixed-point training framework', *International Journal of Computational Science and Engineering*, Vol. 21, No. 3 [online] https://doi.org/10.1504/IJCSE.2020.106064.

Ortega-Sanchez, C. (2011) 'MiniMIPS: an 8-Bit MIPS in an FPGA for educational purposes', *2011 International Conference on Reconfigurable Computing and FPGAs*, pp.152–157 [online] https://doi.org/10.1109/ReConFig.2011.62.

Perera, D.G. and Li, K.F. (2019) 'A design methodology for mobile and embedded applications on FPGA-based dynamic reconfigurable hardware', *International Journal of Embedded Systems (IJES)*, Vol. 11, No. 5 [online] https://doi.org/10.1504/IJES.2019.10018522.

Pham, H., Pillement, S. and Piestrak, S.J. (2013) 'Low-overhead fault-tolerance technique for a dynamically reconfigurable softcore processor', *IEEE Transactions on Computers*, Vol. 62, No. 6, pp.1179–1192 [online] https://doi.org/10.1109/TC.2012.55.

Possa, P., Schaillie, D. and Valderrama, C. (2011) 'FPGA-based hardware acceleration: a CPU/accelerator interface exploration', *18th IEEE International Conference on Electronics, Circuits, and Systems*, pp.374–377 [online] https://doi.org/10.1109/ICECS.2011.6122291.

Rodríguez-Andina, J.J., Valdés-Peña, M.D. and Moure, M.J. (2015) 'Advanced features and industrial applications of FPGAs – a review', *IEEE Transactions on Industrial Informatics*, Vol. 11, No. 4, pp.853–864 [online] https://doi.org/10.1109/TII.2015.2431223.

Romeo, D., LaMagna, J., Hogan, I. and Squire, J.C. (2018) 'An 'Introduction to soft-core processors and a biomedical application', *IEEE Potentials*, Vol. 37, No. 2, pp.13–18 [online] https://doi.org/10.1109/MPOT.2017.2733341.

Romero-Troncoso, R.D.J., Ordaz-Moreno, A., Vite-Frias, J.A. and Garcia-Perez, A. (2006) '8-bit CISC microprocessor core for teaching applications in the digital systems laboratory', *IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006)*, pp.1–5 [online] https://doi.org/10.1109/RECONF.2006.307782.

Santa, F.M., Rodríguez, W.S. and Sánchez, F.R. (2018) '8-bit softcore microprocessor with dual accumulator designed to be used in FPGA', Vol. 22, No. 56, pp.40–50 [online] https://doi.org/10.14483/22487638.12976.

Sharat, B., Chandra, V., Kolin, P., Balakrishnan, M. and Lavenier, D. (2017) 'Hardware acceleration of de novo genome assembly', *International Journal of Embedded Systems*, Vol. 9, No. 1 [online] https://doi.org/10.1504/IJES.2017.081729.

Silvaco HCS08 Processor: 8-bit HCS08 Microcontroller Core Implemented in Freescale's MC9S08xx Family Devices [online] https://www.silvaco.com/products/IP/hcs08/index.html (accessed 17 November 2019).

Silvaco M8051EW and M8051W: High-Performance Versions of the Popular 8051 8-bit Microcontroller [online] https://www.silvaco.com/products/IP/m8051ewm8051w/index.html (Accessed 17 November 2019).

Smith, D.J. (1996) 'VHDL and Verilog compared and contrasted-plus modeled example written in VHDL, Verilog and C', *33rd Design Automation Conference Proceedings*, pp.771–776 [online] https://doi.org/10.1109/DAC.1996.545676.

Teubner, J. and Woods, L. (2013) *Data Processing on FPGAs, Synthesis Lectures on Data Management*, ISBN 9781627050609 [online] https://doi.org/10.2200/S00514ED1V01Y201306DTM035.

Thacker, B.H., Doebling, S.W., Hemez, F.M., Anderson, M.C., Pepin, J.E. and Rodriguez, E.A. (2004) *Concepts of Model Verification and Validation*, Los Alamos National Lab., Los Alamos, NM (US) [online] https://doi.org/10.2172/835920.

Tong, J.G., Anderson, I.D.L. and Khalid, M.A.S. (2006) 'Soft-core processors for embedded systems', *International Conference on Microelectronics*, pp.170–173 [online] https://doi.org/10.1109/ICM.2006.373294.

VAutomation Inc. (2018) *V8-uRISC 8-bit RISC Microprocessor*, Product Specification, VAutomation, Inc. [online] http://ebook.pldworld.com/_semiconductors/Xilinx/AppLINX%20CD-ROM/Rev.7%20(Q3-1998)/docs/wcd0002a/wcd02aaa.pdf (accessed 23 November 2019).

Wang, X. and Ziavras, S.G. (2015) 'A multiprocessor-on-a-programmable chip reconfigurable system for matrix operations with power-grid case studies', *International Journal of Computational Science and Engineering*, Vol. 10, Nos. 1–2 [online] https://doi.org/10.1504/IJCSE.2015.067043.

Wharton, J. (1980) *An Introduction to the Intel MCS-51 Single-Chip Microcomputer Family, Application Note AP-69*, May, Intel Corporation, Application Note AP-69, May, Intel Corporation, USA.

Xilinx (2011) *7 Series FPGA Libraries Guide for Schematic Designs - UG799 (v 13.2)* [online] https://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/7series_scm.pdf (accessed 20 November 2019).

Yang, Y. (2010) 'Implementation of a colorful RGB-LED light source with an 8-bit microcontroller', *5th IEEE Conference on Industrial Electronics and Applications*, pp.1951–1956 [online] https://doi.org/10.1109/ICIEA.2010.5515525.

Zanikopoulos, A., Harpe, P., Hegt, H. and van Roermund, A. (2005) 'A flexible ADC approach for mixed-signal SoC platforms', *IEEE International Symposium on Circuits and Systems*, Vol. 5, pp.4839–4842 [online] https://doi.org/10.1109/ISCAS.2005.1465716.

Zavala, A.H., Nieto, O.C., Ruelas, J.A.H. and Domínguez, A.R.C. (2015) 'Design of a general purpose 8-bit RISC processor for computer architecture learning', *Computación y Sistemas*, Vol. 19, No. 2, pp.371–385, ISSN 1405-5546 [online] https://doi.org/10.13053/CyS-19-2-1941.

Zaykov, P. (2007) 'MIMD implementation with PicoBlaze microprocessor using MPI functions', *Proceedings of the 2007 International Conference on Computer Systems and Technologies (CompSysTech '07)*, Article 4 [online] https://doi.org/10.1145/1330598.1330604.