An Instruction Set Architecture for Secure, Low-power, Dynamic IoT Communication

Shahzad Muzaffar and Ibrahim (Abe) M. Elfadel

Department of Electrical and Computer Engineering
Khalifa University
Masdar City, P.O. Box 54224, Abu Dhabi, UAE
{shahzad.muzaffar,ibrahim.elfadel}@ku.ac.ae

Abstract. This chapter presents an instruction set architecture (ISA) dedicated to the rapid and efficient implementation of single-channel IoT communication interfaces. The architecture is meant to provide a programming interface for the implementation of signaling protocols based on the recently introduced pulsed-index schemes. In addition to the traditional aspects of ISA design such as addressing modes, instruction types, instruction formats, registers, interrupts, and external I/O, the ISA includes special-purpose instructions that facilitate bit stream encoding and decoding based on the pulsed-index techniques. Verilog HDL is used to synthesize a fully functional processor based on this ISA and provide both an FPGA implementation and a synthesised ASIC design in GLOBALFOUNDRIES $65nm$. The ASIC design confirms the low-power features of this ISA with consumed power around $31\mu W$ and energy efficiency of less than $10pJ/bit$. Finally, this chapter shows how the basic ISA can be extended to include cryptographic features in support of secure IoT communication.

Keywords: Dynamic Signaling, Single-Channel, Low-Power Communication, Clock and Data Recovery, Internet of Things, Domain Specific Architecture, Pulsed-Index Communication, Instruction Set Architecture, Secure Communication

1 Introduction

IoT nodes need to meet two conflicting requirements: high data-rate communication to support bursts of activity in sensing and communication, and low-power to improve energy autonomy. Unfortunately, existing protocols fail to meet these requirements simultaneously. Protocols providing high data rates, such as WiFi, WLAN, TCP/IP, USB, etc. [1–3], are power-hungry and involve complex controllers to handle two-way communications. On the other hand, low-power protocols such as 1-Wire [4] and UART [5] have low data rates.

To fill up the gap and address these two requirements at once, a novel family of pulsed signaling techniques for single-channel, high-data-rate, low-power dynamic communication have been recently proposed under the name of Pulsed-Index Communication (PIC) [6] [7]. The most important feature of this family of

protocols is that they do not require any clock and data recovery (CDR). They are also highly tolerant of clocking differences between transmitter and receiver, and are fully adapted to the simple, low-power, area-efficient, and robust communication needs of IoT devices and sensors. These techniques are reviewed in Section 2 with their advantages and disadvantages clarified. The main issue that this chapter addresses is to provide a flexible framework that enables the implementation of the most suitable PIC technique for a given application. The issue of selecting and implementing a communication interface in a constrained IoT node is a prevalent one, and its solution should contribute to the streamlining of communication subsystem design in IoT devices.

One candidate solution is to program all the protocols on a microprocessor and control their selection and parameters through registers. This is a standard practice that is followed for data transfer protocols such as I^2C, I^2S, SPI, UART, and CAN. Another possible solution is to design ASIC for the newest generation of the protocol and make it backward compatible with older versions as in the case of USB 1.0 through 3.0 [8]. Such methods increase silicon area and power consumption, and do not provide any customization features. Yet another approach is to adopt the principles of hardware-software codesign and provide a special-purpose hardware supporting a tuned or extended Instruction Set Architecture that can be used to configure and implement the various communication protocols of a given family without changing or re-designing the on-chip hardware modules. An example of such approach can be found in Cisco's routers where the main CPU (e.g., MPC860 PowerQUICC processor from Motorala/NXP) includes an on-chip Communication Processor Module (CPM) [9]. The CPM is a RISC microcontroller dedicated to several special purpose tasks such as signal processing, communication interfaces, baud-rate generation, and direct memory access (DMA). The work described in this chapter is inspired with such a solution in that it proposes a flexible, fully programmable communication interface for the PIC family based on a full RISC-like ISA tailored for the efficient and seamless implementation of the PIC protocols.

Specifically, a set of special-purpose instructions and registers along with a compact assembly language is proposed to help perform the specific tasks needed for the generation of pulsed signals and to give access to all the hardware resources. The proposed ISA is called Pulsed-Index Communication Interface Architecture (PICIA) and is meant to help reduce the number of instructions required to implement a PIC family member without impacting the advantageous data rates or low-power operation of the PIC family. Verilog HDL is used to synthesize and verify a fully functional processor based on this ISA over the Spartan-6 FPGA platform. Furthermore, an ASIC design in the GLOBALFOUNDRIES $65nm$ process confirms the low-power operation with $31.4\mu W$ and energy efficiency of less than $10pJ/bit$.

This chapter is an expanded version of an earlier publication of ours [10] and includes an entirely new section, Section 6, on secure IoT transmission using an extended ISA with cryptographic instructions. Other changes include improved figures and additional explanations that are spread throughout this chapter.

2 Pulsed-Signaling Techniques

Pulsed-signaling techniques are based on the basic concept of transmitting binary word attributes rather than modulated bits. The attributes are quantified, coded as pulse counts, and transmitted as streams of pulses. The key to the success of these techniques is the encoding step whose goal is to minimize the pulse count. At the receiver, the decoding is based on pulse counting by detecting the rising edge of each pulse. These techniques have the distinguished feature that they don't require any clock and data recovery (CDR), which significantly contributes to their low-power and small foot-print hardware implementations. Recently, three techniques based on this concept have been introduced, namely, Pulsed-Index Communication (PIC) [6], Pulsed-Decimal Communication (PDC) [7], and Pulsed-Index Communication Plus (PIC*plus*). With slight differences, these techniques apply an encoding scheme to a data word B to *minimize* the number of ON bits, and *move* them to the Least-Significant-Bit (LSB) end of the packet with the goal of lowering the number of pulses required to transmit the data bits. The encoding process includes a segmentation step where the data is broken into N independent segments of size l bits each (i.e. $N = B/l$). To maximize data rate, these use, on each segment, an encoding combination of bit inversion and/or segment reversion/flipping. For PIC and PIC*plus*, this combination is meant to reduce the number of ON bits and decrease their index values. For PDC, the same combination is meant to reduce the number of ON bits and decrease the decimal number represented by each segment. To facilitate decoding, flag pulses representing the type of encoding performed are added to each segment. Unlike PIC, the PDC segment flags of two consecutive segments and the PIC*plus* segment flags of four consecutive segments are combined in one data word flag and placed in the header. The PDC further applies a third segmentation step post-encoding whose goal is further reduce the number of pulses per segment and, therefore, further increase the data rate.

All the pieces of information including flags, the number of indices, and the indices themselves in the case of PIC and PIC*plus*, or the decimal numbers of each segment in the case of PDC, are transmitted in the form of pulse streams. Within a given packet, segment pulse streams are separated by an inter-symbol delay (α). The receiver counts the number of pulses for each pulse stream and applies the decoding according to the flags received.

3 Pulsed-Index Communication Interface Architecture (PICIA)

As described in Section 2, the PIC family members share many ideas, some of which are used in exactly the same way and others with few changes. Their packet formats are also quite similar. There could be a number of variations that could be introduced in these techniques as per needs and choice. The proposed PICIA can be used to generate not only these standard protocols with tune-able respective communication parameters (i.e. segment size, inter-symbol

delay, pulse width etc.) but it can also be used to develop other customized communications techniques that use the same underlying idea of transmitting information in the form of pulses. The PICIA is described in detail in the next subsections.

Table 1. PICIA Register Set

	Register	**Type**	**Organization**
1	R0-R7	8 bit GP[a]	8-bit Value
2	Ctrl0	8 bit SP[b]	[0, Mode, 3-bit SegNum, 3-bit SegSize]
3	Ctrl1	8 bit SP	8-bit Pulse Width
4	LoadReg	16 bit SP	16-bit Value

[a.]General Purpose [b.]Special Purpose

3.1 Register Set

The PICIA uses three types of registers. The first type includes a set of eight 8-bit registers, *R0 through R7*, which are programmer-accessible general-purpose registers. The second type is that of Control Registers *Ctrl0* and *Ctrl1* which are 8-bit registers used to store protocol configuration parameters such as mode of transaction (transmitter or receiver), segment number, segment size, and pulse width in terms of a number of clock cycles. These control registers are initially set by the programmer through specific instructions but, once set, they become accessible only to the system. The third type is the *LoadReg* register, which is a 16-bit, I/O-dedicated register used to read the I/O port, set the I/O port, and to store the updated results after an instruction is executed. Like the Control Registers, *LoadReg* is a privileged register accessible only to the system. These register types are summarized in Table 1. In the remainder of the text, the word register will always refer to a general-purpose register.

3.2 Instruction Formats

The PICIA instructions are all 16-bit long and are of three different types. The first type, *I-Type 1*, handles one operand at a time and is used in operations such as to read/write the I/O port, set/clear the *LoadReg*, set various communication protocol parameters, and send/receive pulse streams. I-Type 1 is divided into five fragments, as shown in Fig. 1. The 5-bits *Opcode* represents the type of operation. *Type (R/C)* is used to set the type of operand (register or a constant) in an instruction. *Halt PC/WE* is used either to halt the PC during the transmission of pulse streams or to enable the store operation of received pulse-count to a specified register. The bit *E* sets if an extra pulse should be added to the transmitted pulse stream and/or an extra pulse should be removed from the received pulse stream. The last 8-bits long fragment of I-Type 1 is used to indicate a register number or an immediate constant value.

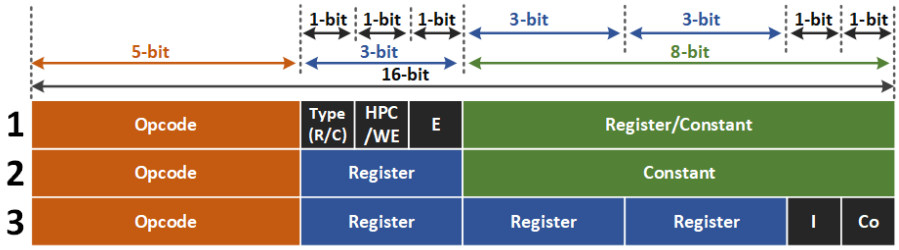


Fig. 1. PICIA Instructions Format

The second type of instruction, *I-Type 2*, needs two operands and is used in operations such as updating a register with a given constant value, and jumping to a specified label in the code depending on the validity of a condition in a register. I-Type 2 is divided into three fragments, as shown in Fig. 1. The 5-bits *Opcode* represents the type of operation. The 3-bits *Register* field is used to indicate one of the general purpose registers and the 8-bits *Constant* field is used to provide a constant value or a label that is present in the code.

The third type of instruction, *I-Type* 3, handles two or three operands simultaneously. I-Type 3 is used in operations such as encoding (inversion and reversion with or without condition), combining and splitting encoding flags, and copying register contents or some other information to a specified register conditionally. I-Type 3 is divided into six fragments, as shown in Fig. 1. The 5-bits *Opcode* represents the type of operation. The 3-bits *Register* fields are used to indicate one of the general purpose registers. The combinations of 1-bit *I* and *Co* fields are used to select the source of information to be copied.

3.3 Addressing Modes

The PICIA employs three addressing modes: immediate, register, and auto-decrement. In the immediate mode, the source is either a constant or a label while the destination is one of the general-purpose, special-purpose, or program counter registers. In the register mode, the register contains the value of the operand. The auto-decrement mode is used only for jump operation where the branch to a label is taken and a specified register decrements by one if the register contains a non-zero number.

3.4 Interrupts

There are three interrupts in the PICIA supported processor. First, the I/O interrupt is generated when the data at the I/O port is available. The system remains in a halt state until the I/O interrupt is reached and the system starts the execution of instructions from the very start. Second, the transmitter interrupt is used to indicate the completion of the transmission of one pulse stream. The PICIA processor remains in a halt state, if activated, until transmitter interrupt

is received and the execution continues from where it paused. Third, the receiver interrupt is generated when the reception of one pulse stream completes. The PICIA processor remains in a halt state until the receiver interrupt is received at which time, program execution is continued.

3.5 External I/O

Three external I/O ports are supported by the PICIA processor. One of these ports is the 16-bit data I/O port that is used to read from and write back to the external environment. To transmit and receive the packets in the form of pulse streams, a 1-bit signal I/O port is used. Another 1-bit data ready port is used to source the generation of I/O interrupts and start the execution of instructions.

4 PICIA Assembly Language

Before diving into the PICIA assembly language in detail, it is necessary to understand few relevant interpretations about the instructions and assembly language. These interpretations are shown in Table 2. The left part of the table shows the instruction interpretations where the values of the control bits are indicated along with the corresponding effect or representation. Similarly, the right part of the table does the same but for PICIA assembly language. The PICIA instructions are listed in Table 3 along with a brief description and an example for each. The instruction categories and types are given in Table 4. More details about the PICIA instructions are given in the next subsections.

Table 2. PICIA Interpretation

Instruction Interpretation		**Assembly Interpretation**	
Control Bit	**Value : Effect**	**Symbol**	**Meaning**
Type (R/C)	0 : Register, 1 : Constant	*R*	Register Only
Halt PC	0 : No Halt, 1 : Halt	C	Constant Only
WE	0 : Register Write Disabled 1 : Register Write Enabled	RC	Register or Constant
E	0 : Extra Pulse Disabled 1 : Extra Pulse Enabled	R, Rx, Ry, Rs	Register Number
I	0 : No Indexing, 1 : Indexing	h	0 : No Halt, 1 : Halt
Co	0 : Copy Segment Disabled 1 : Copy Segment Enabled		

4.1 Type 1 Instructions (I-Type 1)

These instructions are concerned with configuration and transmission control operations and use only one operand. The first instruction towards this is *RP*, *read*

from port, that collects the data from the I/O port and stores it in the *LoadReg*. *WP*, *write to port*, reads data from *LoadReg* and updates the I/O port. There is no operand to these instructions as the system accesses the special purpose register internally. *SSS* and *SSN* set the segment size and the segment number respectively in the *Ctrl0* register. The operand for both of these instructions is an immediate constant value. The operand to *SSS* can be any of 0, 1, or 2 that represents a segment size of 4, 8, or 16 bits respectively.

Table 3: PICIA Assembly Language

	Instruction	**Description**	**Example**
Configuration Instructions			
1	RP	Load data from Input Pins to data register.	RP
2	WP	Output the received data from data register to the Pins.	WP
3	SSS C	Set segment size (C = 0,1,2 for 4 bit,8 bit,16 bit).	SSS 1
4	SSN C	Select segment number (C = 0,1,2,3).	SSN 2
5	SM C	Set Mode (C = 0,1 for Transmitter, Receiver). Setting RX mode clears LoadReg, setting TX loads input into LoadReg.	SM 0
6	SW C	Set width of pulse (C = integer specifying cycle count).	SW 2
7	SRD RC	Set Receiver Inter-Symbol Delay equal to RC number of clock cycles.	SRD R0
8	NOP	No operation.	NOP
Encoding/Decoding Instructions			
9	IV Rx,Ry	Inverse the selected segment. Rx=NOI & Ry=Flags (Rx/Ry= R0,R1,... R7).	IV R0,R1
10	IVC Rx,Ry	Inverse conditionally the selected segment if encoding condition satisfy (ON bits >Seg. Size/2). Rx=NOI & Ry=Flags (Rx/Ry= R0,R1,... R7).	IVC R0,R1
11	FL Rx,Ry	Flip selected segment bits. Rx=NOI & Ry=Flags (Rx/Ry= R0,R1,... R7).	FL R0,R1
12	FLC Rx,Ry	Flip conditionally the selected segment bits if encoding condition satisfy (Seg. >Flip(Seg.)). Rx=NOI & Ry=Flags (Rx/Ry= R0,R1,... R7).	FLC R0,R1
13	IVFL Rx,Ry	Invert and Flip selected segment bits. Rx=NOI & Ry=Flags (Rx/Ry= R0,R1,... R7).	IVFL R0,R1

14	CRC R,Rs,I,Co	Copy register conditionally. R= Rs if I=0. R= Rs , if I=1 and LoadReg [Rs]=1 and Co=0. R=0 otherwise. R=Selected Segment, if Co=1. Rs is ignored. (R/Rs= R0,R1,... R7). Can be used to clear the register.	CRC R1,R2,1,1
15	CF R,Rx,Ry	Combine Flags. R={Rx[1:0], Ry[1:0]}.	CF R0,R1,R2
16	SF Rx,Ry,R	Split Flags. Rx=R[3:2], Ry=R[1:0].	SF R1,R2,R0
Transmission Control Instructions			
17	SP h, E, RC	Send RC number of pulses (RC = register number or constant value). Halt PC if h=1 (h=0,1). (Type = 1 then it's a constant). Send one extra pulse if E=1 (E=0,1).	SP 1,1,4
18	SD h, RC	Inter-Symbol delay of RC number of clock cycles. Halt PC if h=1 (h=0,1).	SD 1,4
19	WRI WE, E, R	Wait for receiver pulse stream interrupt. PC halts till the interrupt arrives. Remove one extra pulse count if E=1 (E=0,1). Enable received pulse count write to register R (R= R0,R1,... R7) if WE=1 (WE=0,1).	WRI 1,1,R0
20	SDB C	Sets the index bits or the data bits in the LoadReg as per the received pulse stream. (C=0,1 for indexing and data respectively).	SDB 1
Register/Branch Update Instructions			
21	WR R, C	Write constant value to a register R (R= R0,R1,... R7).	WR R0,8
22	BNZD R, label	Branch to label and decrement R by 1 if the specified register R contains non-zero number. (R= R0,R1,... R7).	BNZD R0,loop

Table 4: I-Types Instructions

Instructions Catagory	**I-Type**
Configuration	1
Transmission Control	1
Register/Branch Update	2
Encoding/Decoding	3

Segment size information helps the system break the data word into smaller independent segments. The operands to *SSN* can be the numbers 0, 1, 2, and 3. *SSN* is used to select the segment that is going to be processed by all the

following instructions in the program until the segment number is changed again. *SM*, *set the mode*, also accesses the special purpose control register *Ctrl0* and sets or clears a bit representing the mode of operation. The operand to *SM* can either be 0 or 1 that represents the transmitter or receiver mode respectively. During transmitter mode, the signal port is used to send the pulses out and, during reception mode, the same port is used to receive the pulses from the external world. If the receiver mode is selected, the *LoadReg* is automatically cleared by the system to make it ready for reception. If the transmitter mode is selected, the *LoadReg* is updated automatically with the data present on I/O port. *SW*, *set pulse width*, sets the count of system-clock cycles for which the pulse remains high. The operand to *SW* is an 8-bit integer number.

The *SP*, *send pulses*, sends a pulse stream consists of a number of consecutive pulses equal in count specified by the operand that could either be a register or an immediate constant number. The argument h is used to decide if the system should halt during the transmission of a pulse stream or not. If 1, halt the system unless the pulse stream transmission is complete, or continue with the next instruction if 0. The argument E to *SP* instruction informs the system if the pulse stream should include the transmission of an additional pulse at the end of stream or not. This is helpful in representing the no-pulse or zero-index condition with only one pulse as it is in the case of PIC and PIC*plus* transmission, unlike PDC where all the pulse streams are transmitted with an additional pulse. If 1, include an extra pulse or send the exact number of pulses if 0. *SD* is a similar instruction but with minor differences. *SD*, *send the delay*, transmits an inter-symbol delay that is equal in length to the specified number of system-clock cycles. All the arguments and operands work in the same way as that of *SP* except that there is no choice of an extra pulse.

To set the expected number of clock cycles per inter-symbol delay during the process of reception, the instruction *SRD* is used which takes either a register number or a constant number as an operand to represent the number of clock cycles. During a reception, the system needs to wait for the incoming pulse stream so that the pulses can be counted to infer the sent information. To fulfil this task, the instruction *WRI*, *wait for receiver interrupt*, is used. The system goes into the halt state when this instruction is executed and returns back to the normal state at the reception of receiver interrupt that is generated when a pulse stream is received completely. The incoming pulses are counted and the count decrements once if the argument E to *WRI* is set. The count is stored in a specified register R if the argument *WE* is set. Among different types of information chunks in a received packet, a pulse streams related to data could either represent the index number of an ON bit (as in PIC or PIC*plus*) or the decimal number for a segment (as in PDC or other custom techniques). The instruction *SDB*, *set data bits*, removes this confusion by informing the system if the received pulse count needs to be stored directly in the *LoadReg* as a segment's content (if C=1) or a bit in the *LoadReg* needs to be set at the index number represented by the count (if C=0). The last instruction in the category of I-Type 1 is *NOP*, *no operation*, that is used when there is a need

to wait for some operation to complete, as in the case of instructions *SP* and *SD*, without halting the system. In this case, there should be enough number of *NOP*s ($PulsCount+2$) to wait for the completion of a pulse stream transmission. All or some of these *NOP*s can also be replaced by other instructions in order to perform useful tasks instead of waiting for transmission.

4.2 Type 2 Instructions (I-Type 2)

I-Type 2 is the smallest set of instructions. As mentioned earlier, these instructions handle two operands at a time and are concerned with register and/or branch update operations. One of the operands is a register and the other is an immediate constant. One of these instructions is *WR*, *write register*, that is used to store an immediate constant value to a specified general purpose register. The second instruction is the jump instruction *BNZD*, *branch and decrement if not zero*. The instruction takes two arguments, a register to check the condition and a label to jump to. If the content of the specified register is a non-zero value, the program counter jumps to the label and the register value decrements once. The *BNZD* is helpful in writing conditional loops.

4.3 Type 3 Instructions (I-Type 3)

These instructions are concerned with encoding and decoding and use either two or three operands, but all of these operands must be registers. The five instructions, described next, are used in encoding the selected segment. *IV*, *invert*, is used to complement the bits of the selected segment unconditionally and the resulting new segment replaces the corresponding segment in *LoadReg*. The operand register *Rx* stores the new number of ON bits (NOI) in the resulted segment and register *Ry* stores the corresponding flags to represent the encoding type, as per encoding description in PIC and PDC overview. The *IVC*, *invert conditionally*, works the same way as *IV* works but only if the condition of encoding is true. The condition, as mentioned earlier in the overview section, is that the number of ON bits in the selected segment should be greater than half the segment size. The *Rx* and *Ry* get updated with new NOI and Flags respectively. The *FL*, *flip*, and *FLC*, *flip conditionally*, work exactly the same way as *IV* and *IVC*, respectively, except for the base operation that is the bit wise reverse/flipping instead of inversion. The condition here for *FLC* is to check whether the content number of the selected segment is greater than the flipped content number of the same segment. If the condition is true, it means the ON bits are at the higher number of indices, hence, they represent a big decimal number and both of these can be reduced by relocating the ON bits to the lower index numbers. The fifth instruction that takes part in encoding is *IVFL*, *invert and flip*. The *IVFL* works in the same way as the other aforementioned four instructions work except it applies both the inversion and flipping together unconditionally.

The instructions *CF*, *combine the flags*, and *SF*, *split the flags*, are used for PDC, but can be used for any customized technique through PICIA. *CF* takes

two operands, *Rx* and *Ry*, representing two flags to be combined and stores the result in the third operand register *R*. The first two LSBs of both *Rx* and *Ry*, in the same order, are combined to generate four LSBs in *R*. Similarly, *SF* splits the combined flags in a specified register *R* into two separate flags and stores these in registers *Rx* and *Ry*. The *Ry* takes the first two LSBs of *R* and *Rx* takes the next two LSBs of *R*.

The last and the most complex instruction of PICIA is *CRC*, *copy register conditionally*. Based on the given settings for *I* and *Co*, the instruction performs four different copy operations, as shown in Table 5 where *X* is the don't-care and [*Rs*] represents the index number of *LoadReg*. *CRC* can be used for a simple register to register copy because the instruction copies a register *Rs* to *R* if both *Co* and *I* are cleared. If *Co* is cleared and *I* is set, the source to be copied is decided by the bit of *LoadReg* located at the index number represented by the contents of register *Rs*. If the *LoadReg* bit at index *Rs* is cleared, 0 is copied to register *R*, or simply *Rs* is copied to *R* otherwise. This operation is helpful in generating PIC pulse streams. Remember, PIC selects the ON bits only in data and transmits their index numbers in the form of pulse streams. Therefore, *CRC* with such a configuration helps in finding if the target bit is ON or not. If the bit is ON, the index number of it needs to be transmitted that is present in register *Rs* and that is why it is copied to *R*. If the bit is OFF, nothing is there to transmit and that is why 0 is copied to register *R*. Hence, the index numbers of the ON bits can be transmitted in a loop. If *Co* is set, *I* becomes don't care and the contents of the selected segment are copied to register *R*. This is helpful in generating PDC pulse streams as, unlike PIC, it transmits the contents of the sub-segments in the form of pulse streams. Hence using such a configuration for *CRC*, all segments of the data word can be selected and transmitted one-by-one in a loop. All the configurations of *CRC* instruction can be used to generate any other customized transmission techniques based on the idea of transmitting the information in the form of pulse streams.

Table 5. *CRC* INSTRUCTION FUNCTIONALITY

Co	**I**	**LoadReg[Rs]**	**Description**
0	0	X	R=Rs
0	1	0	R=0
		1	R=Rs
1	X	X	R=Selected Segment

5 Experimental Verification and Results

Verilog HDL is used to describe a fully functional processor based on the proposed ISA and a full experimental setup is implemented on the Xilinx Spartan-6 FPGA platform. The prototype platform is used to verify the functionality

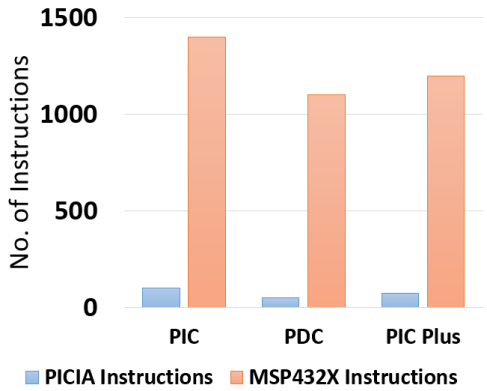


Fig. 2. PIC Family Implementation: PICIA vs. MSP432x

and performance of proposed PICIA. Extensive simulations and real-time hardware verification are performed to verify the results. A clock rate of 25 MHz is used for PICIA testing system. In the experimental flow, the PICIA processor's transmitter sends the 16-bit data starting at 0 with an increment of 1 at each transmission. The PICIA processor's receiver resends the same data back. The returned and original data words are compared to verify the complete round-trip chain.

In another experiment, the software aspects of two implementations are compared. In one implementation, the PIC family member techniques are developed for TI's MSP432X processor family. The reason for choosing the MSP432X in our experiments is that it is an ultra low-power RISC processor, and so it provides an appropriate off-the-shelf choice for comparing the PIC assembly programs using our PICIA processor vs. those of MSP432X. The second implementation used PICIA assembly language to develop the same techniques to run on the implemented processor. Both implementations use a 25MHz clock. The number of instructions required to implement these techniques using MSP432X is approximately 1300 to 1400 on average whereas PICIA needs only 50 to 100 instructions. This is a notable reduction by a factor of 13 to 28, approximately. The data rates offered by the MSP432X implementation are also reduced significantly, approximately by a factor of 100. On the other hand, the data rates are preserved by the implementation of communication techniques using PICIA. The software implementation comparison is shown in Table 6 and Fig. 2.

An example showing how PICIA reduces the number of instructions is illustrated in Fig. 3. At the left side of the figure, an encoding example implemented in C for PDC is presented. If the encoding is implemented using a RISC ISA, around 150 instructions would be required. On the other hand, if the same encoding is implemented using PICIA, only 15 instructions are required. A sample pseudo code in Fig. 3 highlights the flow of the program and the involved PICIA instructions.

```
/* Segmentation */
seg0 = TxData & SEG_MASK;
seg1 = (TxData >> 8) & SEG_MASK;
/* Encoding */
seg0 = PDC_Encode(seg0, 0);
seg1 = PDC_Encode(seg1, 1);
/* Combine Flags */
CFlags = ((Flags1 << 2) & 0x0C) | Flags0;
/* Sub-Segmentation */
subSeg0 = seg0 & SUB_SEG_MASK;
subSeg1 = (seg0 >> 4) & SUB_SEG_MASK;
subSeg2 = seg1 & SUB_SEG_MASK;
subSeg3 = (seg1 >> 4) & SUB_SEG_MASK;

byte PDC_Encode(byte SegData, byte segNo) {
    byte countOfOnes = 0;
    byte SegDataFlipped = 0;
    byte Flags = 0x00;
    countOfOnes = ((SegData >> 7) & 1) + ((SegData >> 6) & 1)
                + ((SegData >> 5) & 1) + ((SegData >> 4) & 1)
                + ((SegData >> 3) & 1) + ((SegData >> 2) & 1)
                + ((SegData >> 1) & 1) + (SegData & 1);
    if (countOfOnes > ON_BITS_LIMIT) {
        SegData = ~SegData;
        Flags = 0x02;
    }
    SegDataFlipped = reverse(SegData);
    if (SegDataFlipped < SegData) {
        SegData = SegDataFlipped;
        Flags = Flags | 0x01;
    }
    if (segNo == 0) {
        Flags0 = Flags;
    } else {
        Flags1 = Flags;
    }
    return SegData;
}
char reverse(char b) {
    b = (b & 0xF0) >> 4 | (b & 0x0F) << 4;
    b = (b & 0xCC) >> 2 | (b & 0x33) << 2;
    b = (b & 0xAA) >> 1 | (b & 0x55) << 1;
    return b;
}
```

~150 RISC Instructions

~15 PICIA Instructions

1- SSS: Select Segment Size = 4
2- SSN: Select Segment Number 0
3- IVC: invert conditionally Seg0
4- FLC: flip conditionally Seg0
5-13: Repeat 2-to-4 steps 3 times for Seg 1,2, & 3
14- CF: Combine Flags of Seg 0 & 1
15- CF: Combine Flags of Seg 2 & 3

Fig. 3. PICIA Code Reduction Example

We have also synthesized the PICIA processor system using GLOBALFOUNDRIES $65nm$ technology and estimated that PICIA hardware consumes around $31.14\mu W$ with a gate count of about 4700 gates. The power consumption results are promising as they remain well within the power budget of a full-hardware implementation of stand-alone pulsed-signaling techniques. Additionally, the consumption of hardware resources is comparable, data rates are preserved and the

Table 6. RESULTS

	Implementation	
	PICIA	**Stand-alone**
Software Implementation Comparison		
Avg. No. Of Instructions	50-100	1300-1400
Avg. Data Rate (Mbps)	≈4.1-7.1	≈0.041-0.071
Hardware Synthesis Comparison		
Power (μW)	≈31.14	≈19-26.6
Avg. E$_b$ (pJ/bit)	≈4.2-7.6	≈2.7-6.5
Area (gate count)	≈4700	≈2100-2400

required number of instructions is reduced. Moreover, PICIA offers a customizable solution. The PICIA solution differs in that it offers a fully programmable communication interface that is specifically geared to the realization of pulsed-transmission techniques.

6 Securing PICIA

This section presents a possible extension of the PICIA to support of secure PIC communication [11]. An advantage of the proposed extension is that it does not require any modification in the PICIA instruction format as it employs the very same instruction types of Section 4 to add instructions dedicated to cryptographic functions. The security layer extension of PICIA offers a programmable environment to select not only a suitable encryption algorithm but also to choose among various execution options of the selected algorithm with the goal of trading off transmission security with data rate. Specifically, the PICIA security layer has the following features:

1. Support of multiple encryption algorithms such as simple XOR, MA5/1[11] and AES.
2. Encryption gating in case the crypto function is not needed.
3. Configurable encryption hardware to tune the number of clock cycles used in data encryption. A tradeoff between the number of crypto clock cycles and the required crypto hardware resources is implemented through the iterative use of a smaller crypto unit. In such case, the unused crypto hardware units are gated.

In the following subsections, the security features of the extended PICIA architecture are highlighted.

6.1 Extended Register Set

Two new registers are added to the PICIA register set in support of the security layer extension, as shown in Table 7. The first register is the Control Register *Ctrl2* which is an 8-bit register used to store configuration parameters of the

Table 7. Security Layer Registers in Addition to the Regular Registers of Table 1

5	Ctrl2	8 bit SP[a]	[Enable SL[b], 3-bit Enc.[c] Algorithm 4-bit Enc. Speed]
6	EncIniKey	16×16-bit SP	256-bit Initial Key Array of sixteen 16-bit registers

[a.]Special Purpose [b.]Security Layer [c.]Encryption

security layer such as enabling the security layer, selection of the encryption algorithm, and the speed of encryption in terms of number of clock cycles. The programmer initially sets the control register through a specific instruction but, once set, it becomes accessible only to the system. The second register is the 256-bit *EncIniKey* register, organized as an array of sixteen 16-bit registers, and used to store the initial encryption key. Like the Control Register, *EncIniKey* is a privileged register accessible only to the system.

6.2 Extended Instruction Set

Three new instructions are added to the PICIA assembly language in support of the security layer. They are shown in Table 8. These instructions deal with the configuration and control of the security layer. The first instruction is *ESL*, *enable security layer*, which activates the security layer and updates the *Ctrl2* register. The operand *En* is a one-bit modifier whose *ZERO* value signifies normal PIC transmission without encryption. Its *ONE* value enables encryption ahead of transmission. The second *ESL* operand, *Alg*, is a 3-bit operand that selects the encryption algorithm that should be used. There can be a maximum of eight hardware blocks in the PICIA processor system, each representing a particular encryption algorithm. In our current implementation, an *Alg* of 0 selects a simple XOR operation, while a value of 1 selects MA5/1, a modified, PIC-compatible version of the symmetric A5/1 encryption algorithm [11]. The third *ESL* operand, *ES*, is used to set the speed of the encryption process in terms of the number of clock cycles. This instruction assumes that the encryption techniques implemented within the PICIA processor support changing the number of clock cycles used to generate a full encrypted data word. For example, if MA5/1 is selected to use one clock cycle, the full encryption hardware would be utilized. If the same algorithm is chosen to use four clock cycles, then one-fourth of the hardware would be used, and the rest would be gated to save power. The *ES* operand takes an unsigned integer value in the range of 0 to 15. The number of encryption clock cycles is calculated as $n_C = 2^{ES}$. Through this operand, a trade-off between crypto latency and power can be easily programmed into the configuration of the security layer.

As described earlier, the length of the key register *EncIniKey* is 256 bits. The same register can also be used for initializing shorter keys, e.g, the 128-bit initial key of MA5/1. There is therefore a need for introducing instructions for key-length setting and *EncIniKey* register initialization. Instructions *LPI*, *lock*

Table 8. Security Layer Instructions in Addition to those of Table 3

	Instruction	Description	Example
Security Layer Instructions			
23	ESL En,Alg,ES	Enable security layer. Enable if En=1, disable if En=0. Alg selects encryption algorithm (0:XOR, 1:MA5/1,...7:OtherAlgo7). ES sets encryption speed in terms of number of clock cycles/encryption-iteration. (Number of clock cycles (n_C) = 2^{ES}.	ESL 1,0,1
24	LPI	Lock previously executed instruction. Unless unlocked, all the next instructions are considered as 16-bit constant values for the locked instruction.	LPI
25	UPI	Unlock the locked instruction.	UPI
Mapping			
Security layer instructions are mapped to I-Type 1			

previous instruction, and *UPI*, *unlock previous instruction*, are introduced for that very purpose. *LPI* locks the previously executed instruction in the control unit while keeping all the generated control signals active unless unlocked using *UPI*. In other words, these two instructions define the start and end of the user's key section in the assembly program and must follow the *ESL* instruction. All the 16-bit binary numbers between these two instructions are considered segments of the full initial key. These segments are stored in the *EncIniKey* register using an internal 4-bit offset register. The offset register defines the row index of a 16×16 array version of the *EncIniKey* register. The offset register is cleared when the *LPI* instruction is executed and is incremented when a 16-bit segment is stored successfully. An example of *EncIniKey* initialization is shown in Table 9, where the current offset represents the *EncIniKey* offset value before the execution of a given instruction and the updated offset represents the *EncIniKey* offset value after its execution.

6.3 Instruction Format

There is no change to the PICIA instructions format given in Fig. 1 as a result of adding of the crypto instructions. All the new assembly language instructions, described in previous subsections, are of the I-Type 1 instruction format. As shown in Fig. 4, the only change we need to account for is in terms of operand values. In particular, the *[Alg,ES]* operands are added to the field "Register/Constant" and the *En* operand, which controls the enabling of the secirity layer, is added to "E" field. The instruction opcode directs the instruction decoder to activate the control signals as per the issued assembly language command.

Table 9. Key Initialization Examples

16-bit Key	64-bit Key	256-bit Key	Current Offset	Updated Offset
...	...	...	...	...
ESL 1,1,0	ESL 1,1,0	ESL 1,1,0	8	9
LPI	LPI	LPI	0	0
0xF192	0xF192	0xF192	0	1
UPI	0x11AB	0x11AB	1	2
...	0xA9F6	0xA9F6	2	3
	0x3313	0x3313	3	4
	UPI	.....	...	...
	...	.....	...	...
		0x46F4	15	0
		UPI	15	0
		...	15	0

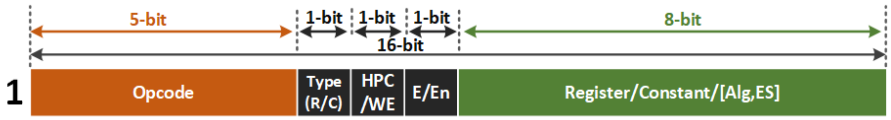


Fig. 4. Additional operand values in the PICIA I-Type 1 Instructions

7 Conclusions

The Pulsed-Index Communication Interface Architecture (PICIA) is a RISC-style special purpose ISA for single-channel, low-power, high data rate, dynamic, and robust communication based on pulsed-signaling protocols. It is designed to facilitate the efficient generation of compact assembly code that is specific to such communication interfaces. This hardware/software co-design capability can be used to embed not only an existing PIC family member but also any custom nonstandard PIC protocol without changing the underlying hardware while greatly reducing the number of required instructions. Furthermore, such communication interface implementation will result in minimal to no impact on the data rates, power consumption, or the reliability of the protocols. The PICIA processor has been synthesized in GLOBALFUONDRIES $65nm$ technology and has been found to consume only $31.14\mu W$, which translates into an energy efficiency of less than $10pJ$ per transmitted bit. To support secure communication, the basic PICIA has been extended to provide a programmable environment for selecting a suitable encryption algorithm and controlling its latency at execution. PICIA's micro-architecture and the optimized hardware blocks that compactly implement its RISC-style ISA are the subject of a separate publication.

Acknowledgments

This work has been supported by the Semiconductor Research Corporation (SRC) under the Abu Dhabi SRC Center of Excellence on Energy-Efficient Electronic Systems (ACE^{4}S), Contract 2013 HJ2440, with customized funding from the Mubadala Development Company, Abu Dhabi, UAE.

References

1. S. Dayu, X. Huaiyu, S. Ruidan, and Y. Zhiqiang, "A Geo-related IoT Applications Platform based on Google Map," *7th International Conference on e-Business Engineering (ICEBE)*, pp. 380–384, Shanghai, China, November 2010.
2. J. Byun, S. H. Kim, and D. Kim, "Lilliput: Ontology-based platform for IoT social networks," *IEEE International Conference on Services Computing*, pp. 139–146, Anchorage, AK, USA, June-July 2014.
3. Jenq Muh Hsu and Chin Yo Chen. "A Sensor Information Gateway Based on Thing Interaction in IoT-IMS Communication Platform," *10th International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP)*, pages 835–838, Kitakyushu, Japan, August 2014.
4. MAXIM, *OneWireViewer User's Guide, Version 1.4*, 2009.
5. C. dos Reis Filho, E. da Silva, E. de L. Azevedo, J. Seminario, and L. Dibb, "Monolithic data circuit-terminating unit (DCU) for a one-wire vehicle network," *Proceedings of the 24th European Solid-State Circuits Conference (ESSCIRC '98)*, pp. 228–231, Hague, Netherlands, September 1998.
6. S. Muzaffar, A. Shabra, J. Yoo, and I. M. Elfadel, "A Pulsed-Index Technique for Single-Channel, LowPower, Dynamic Signaling," *Design, Automation and Test In Europe (DATE'15)*, pp. 1485–1490, Grenoble, France, March 2015.
7. S. Muzaffar, and I. M. Elfadel, "A Pulsed Decimal Technique for Single-channel, Dynamic Signaling for IoT Applications," *25th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC 2017)*, pp. 1–6, Abu Dhabi, UAE, October 2017.
8. R. Teja, B. R. Jammu, M. Adimulam, and M. Ayi, "VLSI implementation of LTSSM," *International conference of Electronics, Communication and Aerospace Technology (ICECA 2017)*, pp. 129–134, Coimbatore, India, April 2017.
9. linux-mips.org, "Cisco Systems Routers," 2012. [Online]. Available: https://www.linux-mips.org/wiki/Cisco.
10. S. Muzaffar, and I. M. Elfadel, "An Instruction Set Architecture for Low-power, Dynamic IoT Communication," *26th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC 2018)*, Verona, Italy, October 2018. To appear.
11. S. Muzaffar, O. T. Waheed, Z. Aung, and I. M. Elfadel, "Single-clock-cycle, Multilayer Encryption Algorithm for Single-channel IoT Communications," *IEEE Conference on Dependable and Secure Computing (DSC 2017)*, pp. 153–158, Taipei, Taiwan, August 2017.