Journal of Information Systems and Telecommunication
Vol.10, No.4, October-December 2022, 279-286
http://jist.acecr.org
ISSN 2322-1437 / EISSN:2345-2773

ARASP: An ASIP Processor for Automated Reversible Logic Synthesis

Zeinab Kalantari[1*], Marzieh Gerami[2] , Mohammad Eshghi[3]

[1]. Department of Computer Engineering, Rafsanjan Branch, Islamic Azad University, Rafsanjan, Iran
[2]. Department of Computer Engineering, ShahreKord Branch, Islamic Azad University, ShahreKord, Iran
[3]. Faculty of Electrical and Computer Engineering, Shahid Beheshti University, Tehran, Iran

Received: 04 Jun 2021/ Revised: 04 Jan 2022/ Accepted: 02 Feb 2022

Abstract
Reversible logic has been emerged as a promising computing paradigm to design low power circuits in recent years. The synthesis of reversible circuits is very different from that of non-reversible circuits. Many researchers are studying methods for synthesizing reversible combinational logic. Some automated reversible logic synthesis methods use optimization algorithms Optimization algorithms are used in some automated reversible logic synthesis techniques. In these methods, the process of finding a circuit for a given function is a very time-consuming task, so it's better to design a processor which speeds up the process of synthesis. Application specific instruction set processors (ASIP) can benefit the advantages of both custom ASIC chips and general DSP chips. In this paper, a new architecture for automatic reversible logic synthesis based on an Application Specific Instruction set Processors is presented. The essential purpose of the design was to provide the programmability with the specific necessary instructions for automated synthesis reversible. Our proposed processor that we referred to as ARASP is a 16-bit processor with a total of 47 instructions, which some specific instruction has been set for automated synthesis reversible circuits. ARASP is specialized for automated synthesis of reversible circuits using Genetic optimization algorithms. All major components of the design are comprehensively discussed within the processor core. The set of instructions is provided in the Register Transform Language completely. Afterward, the VHDL code is used to test the proposed architecture.

Keywords: Reversible logic; Optimization Algorithms; Application Specific Instruction Set Processors; ASIP; RTL.

1- Introduction

Application specific instruction set processors (ASIP) can compromise the advantages of custom ASIC chips and general DSP chips. In other words, ASIP chips utilize high performance and low power of ASIC chips and flexibility of DSP chips [1][2][3][4][5].
There is a tradeoff between cost and speed in ASIPs.
Programmability is the main advantage of ASIPs, which gives more flexibility to software developers. Other advantages are more convenient in the design and debugging process, predictability, and shorter time to market. Hardware and software are two aspects of ASIPs rather than one aspect of the task being dominant. Besides, compared to general-purpose processors, ASIP benefits from having specific instructions to perform a specific task faster and reduce programmer errors. Accordingly, efficiency and programmability are both advantages of ASIPs compared to general-purpose processors.
In this paper, a novel ASIP-based processor for the synthesis of reversible circuits is proposed. This processor is used to synthesize reversible circuits using optimization algorithms. VHDL code is used to simulate and test the proposed architecture. The main objective of the proposed design is programmability as it is the major concept of ASIP. In addition, the suggested structure reduces hardware complexity.
The organization of the rest of the paper is as follows. The next section and subsequent sections present a background on the synthesis of reversible circuits. Section 3 details the proposed ASIP architecture model for the synthesis of reversible circuits. The testing process describes in section 4. Finally, section 5 concludes the paper.

✉ **Zeinab Kalantari**
z.kalantari@iaurafsanjan.ac.ir

2- Background

Reversible logic has applications in low power computing, quantum computing, nanotechnology, optical computing, and DNA computing. The design of the reversible circuits is quietly different from the design of conventional irreversible logic circuits [6] because of the different gates that are available in reversible logic.

The synthesis of reversible circuits differs significantly from synthesis using traditional irreversible gates. Many algorithms have been proposed for the synthesis of reversible circuits [7][8][9][10]. Dengli et al. proposed an improved KFDD based reversible circuit synthesis method [7]. Ahmed et al. suggested a synthesis approach using reorder algorithm [8]. Basak et al. presented an algorithm using the ESOP expressions [9]. Some automated reversible logic synthesis methods, such as genetic algorithms (GAs) are also presented [11][12][13][14][15][16]. These algorithms use optimization algorithms.

In optimization algorithms, the main process is generating some random circuits and then computing the output truth table of generated circuits. Then the hamming distance between the truth table of generated random circuits and the truth table of the given function is calculated. After that, according to the optimization algorithm the best circuit is selected. These operations are repeated while desired hamming distance is reached.

To implement the automated reversible synthesis algorithm, a background on reversible gates is needed. The next section is illustrated to introduce reversible logic gates. After that, a general algorithm for the synthesis of all reversible circuits is presented.

2-1- Reversible Gates

Since a serious problem in modern VLSI designs is power consumption, Low power circuit design is one of the most attractive subjects for hardware designers. Landauer has shown that for irreversible logic computations, each bit of information lost, generates kTln2 joules of heat energy, where k is Boltzmann's constant and T is the absolute temperature at which computation is performed [17]. Bennett showed that kTln2 energy dissipation would not occur if a computation is carried out reversibly [18]. This part of energy dissipation is independent of what the underlying technology is.

In reversible circuits, no bit of information is lost, and reversible computation in a system can be performed only when the system comprises reversible gates.

In a reversible gate, there is a one-to-one correspondence between its inputs and outputs. As a result, the number of outputs of a reversible gate is the same as the number of inputs, and for each input vector, there is a unique output vector and vice versa.

Some more common gates to design reversible logic circuits are Feynman Gate, FG [19], Toffoli Gate, TG [20], Fredkin Gate, FRG [21] are more common gates to design reversible circuits.

A 2*2 Feynman Gate, also known as controlled-NOT (CNOT), is depicted in Fig.1.a. It implements the logic functions: P = A and Q = A⊗B.

A 3*3 Toffoli Gate has 3 inputs: 2 control inputs, that are copied to the first 2 outputs and one other input that is complemented if all control inputs are 1s and are directly copied to the last output otherwise [20]. A 3- input, 3-output Toffoli Gate is shown in Fig.1.b. The inputs 'A' and 'B' are passed as first and second outputs, respectively. The third output is controlled by 'A' and 'B' to invert 'C'.

A 3*3 Fredkin Gate is depicted in Fig.1.c. Here the input 'A' is passed as the first output. Inputs 'B' and 'C' are swapped to get the second and third outputs, which are controlled by 'A'. If A = 0, then the outputs are simply duplicating of the inputs; otherwise, if A = 1, then the two input lines (B and C) are swapped.

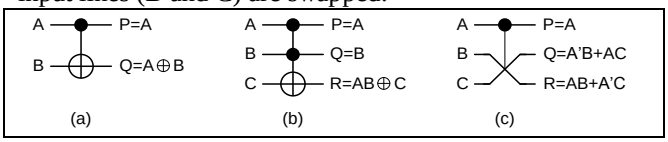

Fig. 1 (a) Feynman gate, (b) Toffoli gate, and (c) Fredkin gate

2-2- Synthesis of Reversible Circuits

Synthesizing a reversible circuit using a searching algorithm is a complex problem with a large amount of searching space. The number of combinations for placing one Toffoli r×r gate in an n×n circuit (0<r<n-1) to synthesize a reversible n×n circuit is expressed by (1):

$$\rho = n.\sum_{r=0}^{n-1}\binom{n-1}{r} = n.2^{n-1} \quad (1)$$

If the number of required gates to design a circuit is *m*, then the number of possible circuits is ρ^m. If Fredkin and Press are added to the set of Toffoli r×r gates, then the number of possible circuits is $(3\rho)^m$. So optimization algorithms, especially GA, are used to find the global minimum or maximum of a function, in an extensive searching space [11][12][13][14]. In the next subsection, a review of automated synthesis is presented.

2-3- Automated Reversible Logic Synthesis

The general algorithm for automated reversible logic synthesis is shown in Fig.2. This processor is used to synthesize reversible circuits.

The algorithm starts by generating a random configuration (a random state) of a circuit with one gate. We consider the hamming distance between the truth table of this circuit and the truth table of a given function as the cost function. Then for each new configuration, it's necessary

to compute the truth table. So the main operations in such algorithms are computing the truth table of each circuit and then comparing its truth table with the destination truth table to select the best circuit, based on the desired algorithm. So in a given iteration, the algorithm generates n new circuits at a time. Each new circuit is derived from the old configuration. The hamming distances of the two circuits are then compared.

After the process of finding a new circuit, comparing it to the current configuration, and either accepting or rejecting, it is done n times.

After all, if the stopping criteria of the algorithm, zero hamming distance, is not reached, the number of gates will increase and the algorithm is repeated for a new circuit with extra gates.

```
  Set NoG=1   //the number of gates being used for the synthesis
  Set S=S0    // random initial state
Loop1:
   Initiate a random circuit S using NoG
  While (up to max-iteration)
{
   While(required number of circuits not generated)
  {
      Generate new circuit S' by perturbing S;
      For (all rows of truth-table)
              E=E+HD(des[i]&mask , Syn[i]&mask);
     ΔE=E(S')-E(S)
    If (ΔE<=0)
        S=S';
   }
  If (HD!=0)
  {
    NoG=NoG+1;
    Goto Loop1;
  }
  else
        Print circuit;     } }
```

Fig. 2 Algorithm description

3- Architecture Overview

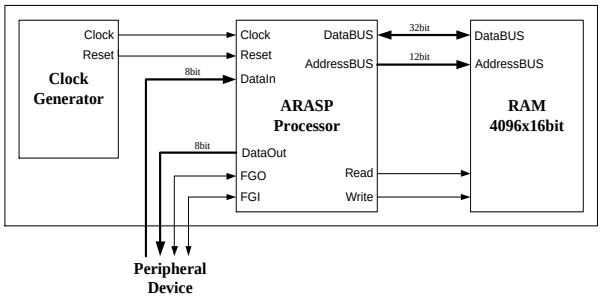

Fig. 3. ARASP processor

This paper introduces an ASIP processor which is useful in the application domain of reversible circuits. This processor is called ARASP, an ASIP processor for automated reversible logic synthesis. The schematic of the ARASP processor is illustrated in Figure 2. The explanation of the proposed register configuration is in Figure 3.

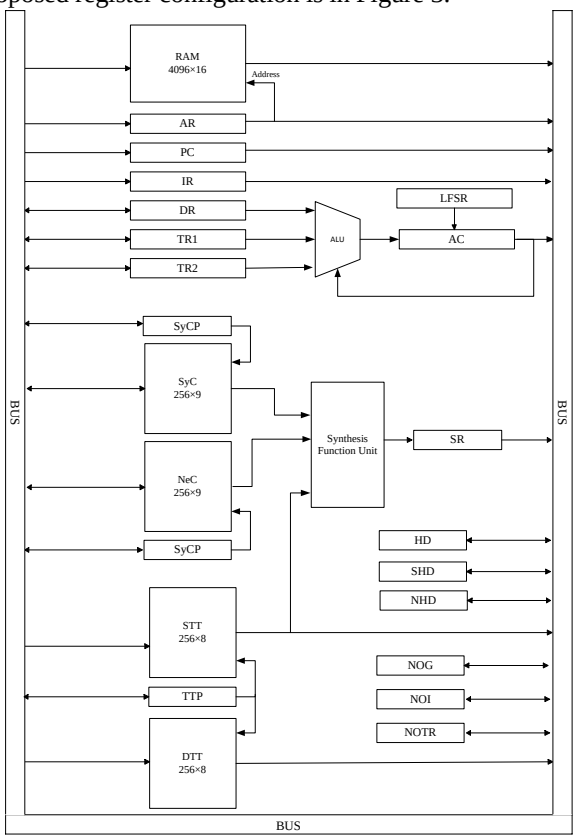

Fig. 4 ARASP register configuration

We proposed a 16-bit wide ASIP architecture with 16-bit integer operations for common arithmetic operations and a specific function unit for the automated synthesis of a reversible circuit.

3-1- Global View on ARASP

Registers have clock inputs that are all connected to the main system clock. Each AC, DR, TR_1, and TR_2 are 16-bit registers that provide operands of ALU. The output of ALU is connected only to the AC. The instruction register, IR, provides the instruction bits for the controller. The 12-bit program counter register is called PC, this register provides an address for current instruction through the memory address register, AR. This register also is a 12-bit binary up counter. The arithmetic logic unit, ALU, is a combinational logic unit with two 16-bit inputs, four flag inputs, and control inputs that specify the integer operations. The output of this unit is connected to the input of AC.

The next part is a function unit for the automated synthesis of a reversible circuit. This unit works as follows. At first, a random circuit with some gates that the NOG (Number OF Gates) register specifies is generated. The SyC (Synthesis Circuit) register bank maintains this random gate. The SyCP (Synthesis Circuit Pointer) register which is an 8-bit up-counter is used to specify each register of this register bank (each reversible gate) in each step. Same as SyC, NeC register bank is used to hold the neighboring circuit of synthesis circuit in each step, the size of this register is as SyC, also NeCP (Neighboring Circuit Pointer) is used to define a register of NeC register bank in each step of the synthesis.
DTT (Destination Truth Table) consists of 256 8-bit registers to maintain the truth table of a given function. As the algorithm generates a reversible circuit in each step, we need a register bank to maintain the truth table of the generated circuit. So STT (Synthesis Truth Table) is used for this aim.
The size of this register bank is the same as DTT. TTP is an 8-bit up-counter that refers to each row of DTT or STT in each step of the algorithm.
As said before the major time-consuming operation in the synthesis of a reversible circuit is calculating the output of the circuit for all combinations of inputs. So the synthesis function unit that is a combinational circuit calculates each row of synthesis truth table of desired circuit, SyC or NeC.
After synthesis of the desired circuit, we need to compare this truth table with the destination truth table, DTT. So we have to calculate the hamming distance between synthesis truth table STT and destination truth table DTT, as the cost function. After that, the calculated hamming distance will set the SHD or NHD depending on the circuit that is synthesized.
The final step in the algorithm is selecting the best circuit. So we need to compare SHD and NHD registers and set HD register with one of these registers.

4- Proposed Architecture

In this paper, the proposed CPU is referred to as ARASP. The proposed processor employs a reduced hardware requirement and application specific instruction set. Due to the size of its data register and buses, ARASP is considered to be a 16-bit processor. It has direct and indirect addressing modes. ARASP also has specific instructions and input-output interrupts.

4-1- Main Memory Organization

The ARASP is capable of addressing 4096 bytes of memory through its 12-bit address lines. This memory is addressed by a register called AR.

4-2- Register Configurations

The main data register of ARASP is AC, which is used in conjunction with most general instructions. This processor has overflow, carry, zero, and sign flags (o, c, z, and s). These flags may be modified by arithmetic operations.
ARASP consists of two parts, global unit, and specific unit. The major components of the global unit are AR, PC, IR, DR, TR_1, TR_2, AC, LFSR, and ALU. Also, the components of the specific part are SyC and NeC register bank that consists of 256 9 bit registers. These register banks hold synthesis and neighboring circuits each consisting of at most 8 gates. DTT and STT register banks hold destination and synthesis truth tables respectively. According to the size of these register banks and hardware restrictions. The desired circuit can have at most 8 inputs (256 8-bit registers). HD, SHD, and HD registers that are 8-bit registers are used for holding hamming distance of the circuit throughout the running synthesis algorithm.

4-3- Instruction Types

The ARASP has a total of 47 instructions totally, and the specific instructions are summarized in Table 3. The Proposed processor has two different types of instruction sets (Table 1). The Memory reference instructions need the main memory address to do their operations and the Non-memory reference instruction set, which needs no memory for their operands. The ARASP'S memory instruction set can be used by direct and indirect addressing modes.

Table 1. Instruction Types and Addressing Modes of ARASP

Instruction Type	M	I	Address	Addressing Mode
Memory	1	0	No	Direct
Memory	1	1	No	Indirect
Others	0	×	Yes	-

As presented in Fig. 5, in memory reference instructions most significant bit of instruction (bit 15) is set, to specify the memory reference instruction type. Bit 14 called I, specifies direct or indirect addressing mode (0 for direct and 1 for indirect). The next 4 bits (bits 10-13) specify the operation of a memory reference instruction. As this type of instruction need a memory word for holding one of the operands, in these types of instructions we should refer to the main memory to read the operand. If it is set to 1 the operand's address is indirect and if it is set to 0 the operand's address is direct.

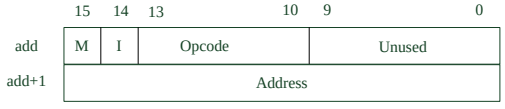

Fig. 5 Memory reference instruction format

These types of instructions occupy a byte whose most significant bit (bit 15) is 0. In this type of instruction, bit 14 specifies output and register instructions or specific instructions (0 for output and register instructions and 1 for specific instructions). The other 4 bits specify operations of instructions (Fig. 6).

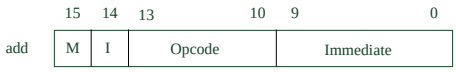

Fig. 6. Non Memory reference instruction format

The fetch, decode and calculation of effective address phases of the instruction cycle could be as follow:

***Interrupt* :**
IEN(FGI+FGO) : R ← 1
RT_0 : AR ← 0
RT_1 : M[AR] ← PC , PC ← 0
RT_2 : PC ← PC+1, IEN ← 0 , R ← 0 , SC ← 0
Fetch :
$\overline{R}T_0$: AR ← PC , PC ← PC+1
$\overline{R}T_1$: IR ← M[AR]
Decode :
$\overline{R}T_2$: D_{31} … D_0 ← IR[10-14] , AR ← IR[0-8] , F ← IR(9) , M ← IR(15)
Address Fetch :
MT_3 : AR ← PC, PC ← PC+1
MT_4 : AR ← M[AR]
M $\overline{I}$ T_5 : nothing
M I T_5 : AR ← M[AR]

4-4- Arithmetic Logic and Shift Unit

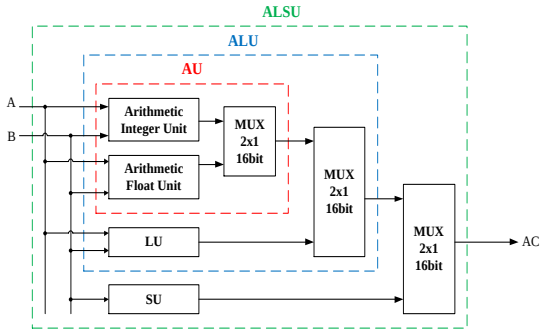

Fig. 7 Arithmetic Logic and Shift Unit

The presented processor supports basic arithmetic, logic, and shift units which are presented in

Table 2. ALU Operators

S4	S3	S2	S1	S0	Unit	Operation	Function
0	0	0	0	0	AU	ADD	A+B
0	0	0	0	1		SUB	A-B
0	0	0	1	0		DEC	A-1
0	0	0	1	1		INC	A+1
0	0	1	×	×		MUL	A×B
0	1	0	0	0	LU	AND	A∧B
0	1	0	0	1		OR	A∨B
0	1	1	0	0		XOR	A⊕B
0	1	1	1	0		NOT	¬B
0	1	1	1	1		PASS	A
1	×	×	0	0	SU	SHL	SHL(B)
1	×	×	0	1		SHR	SHR(B)
1	×	×	1	0		ROL	ROL(B)
1	×	×	1	1		ROR	ROR(B)

The 5-bit opcode (S_0 to S_4) hierarchically selects the proper operation. Besides the main results, five arithmetic flags (Carry, Overflow, Zero, and Sign) are set by the ALU. Each flag obtains the proper value by the Eq.s 2 to 5 considering that the input values are unsigned integer. AC, DR, TR1, and TR2 can be considered as both first and second operands.

C-Flag = Cout when (Op.=ADD|SUB|DEC|INC|SHL|SHR) (2)
O-Flag = '1' when (Op.=ADD|INC|SHL & C_{out}='1') | (Op.= SUB|DEC & C_{out}='0') | (Op.=MUL & 16-bit MSB≠0) (3)
Z-Flag = '1' when (16-bit LSB=0) (4)
S-Flag = '1' when (Op.=SUB|DEC & C_{out}='0') (5)

4-5- Instruction Set

The instruction set for the ARASP is depicted in Table3.
The objective of synthesizing a reversible circuit is to compute a circuit for a given function. So we need to have a destination truth table of a given function. LDTT instruction reads the truth table of the desired function from the main memory to the DTT register bank. This instruction is a specific instruction that needs to refer to the main memory for its operation. Because of the hardware restrictions, it assumes that each function could have at most eight inputs. So the process of reading the rows of the destination truth table from memory reads some rows with the number that NoTR defines. The value of this register is set by SNOTTR instruction according to the value of the NOG register.
The *STRC* instruction stores the generated circuit of the synthesis process of a given function in the main memory.
The *RAND* instruction generates a 16-bit random number.
The *CLRNOG instruction* clears 8-bits of the NOG register, while *INCNOG* instruction increments the value of this register.
The *SNOI* instruction initializes the NOI register by an immediate number.
The *GRNDC* instruction generates a random circuit with some gates that are determined by the value of the NOG register. This instruction gets the value of the NOG register and the number of gates and sets a random value to some registers of SyC register bank according to the specified value of NOG.
The *GNBRCROS* instruction performs the crossover operation on a circuit to generate a new circuit called neighboring circuit from synthesis circuit. The instruction selects two random gates from the synthesis circuit that are in the SyC register bank. After that, it exchanges the position of these two gates to generate a new circuit called a neighboring circuit that is placed on the NeC register bank.
The other instruction that performs mutation operation is *GNBRMUT*. This instruction also generates a random new circuit from the existing circuit by mutation operation. The instruction selects a random gate from the synthesis circuit

that is in the SyC register bank. After that, it exchanges the position of control and main inputs to generate a new circuit called a neighboring circuit that is also placed on the NeC register bank.

To compute the truth table for each circuit it is necessary to set each row of the truth table with initial values 0 to 2^n-1. To achieve this goal, SIVDTT is used.

MASK instruction generates a mask pattern for output.

CALCSTT instruction computes the truth table of each generated circuit, SyC or NeC. In other words, this instruction, compute the value of the output for each combination of inputs in a circuit with some gate. By feeding the initial value of a given row of the truth table to the circuit as the first stage and synthesizing them the computation operation starts and then synthesis ALU calculates the output of this gate. The next gate gets the calculated output of the preceding gate and calculates the output. These operations continue while the output of the last gate is computed. This output will be replaced by the value of the given row. CALSTT repeats these operations for all combinations of inputs (all rows of truth table).

To calculate the hamming distance between syntheses or neighboring truth tables and destination truth tables, CALCHD instruction is used.

The *SBC instruction* selects the best circuits, a circuit with less hamming distance, between a circuit and a neighboring circuit.

Finally, SIZHD is used to determine the zero value of the HD register.

NOGTAC instruction is used to transfer the value of the NOG register to the AC.

Table 2. ALU Operators

S_4	S_3	S_2	S_1	S_0	Unit	Operation	Function
0	0	0	0	0	AU	ADD	A+B
0	0	0	0	1		SUB	A-B
0	0	0	1	0		DEC	A-1
0	0	0	1	1		INC	A+1
0	0	1	×	×		MUL	A×B
0	1	0	0	0	LU	AND	A∧B
0	1	0	0	1		OR	A∨B
0	1	1	0	0		XOR	A⊕B
0	1	1	1	0		NOT	¬B
0	1	1	1	1		PASS	A
1	×	×	0	0	SU	SHL	SHL(B)
1	×	×	0	1		SHR	SHR(B)
1	×	×	1	0		ROL	ROL(B)
1	×	×	1	1		ROR	ROR(B)

5- Testing Process

In this paper, the proposed CPU is referred to as ARASP. The proposed

A structural VHDL code in Fig. 8 is used to test and verify the functionality of the given structure. Using the instruction set of the presented ARASP processor, the following code has to be programmed to generate a random circuit for the desired N×N function which is stored from the memory address Addr1.

```
CLRNOG
INCNOG
NOI  N
SNOTTR
LDDTT  Addr1
GRNDC
SISTT
CLCSTT
CLCHD 0
GNBRCMUT
SIVSTT
CLCSTT
CLCHD 1
SBC
```

Fig. 8 VHDL code

6- Conclusion

We showed that synthesizing a reversible circuit using a search algorithm is a complex task with a large number of search spaces. So optimization algorithms, especially Genetic Algorithm, GA, are used to find the global minimum or maximum of a function, in an extensive searching space. As in such algorithms, the process of calculating values of outputs is a time-consuming operation. So, we need a processor to speed up the process of synthesis. As a result, application specific flexibility is mandatory to meet the performance requirements of synthesis reversible circuits.

In this paper, we presented a novel design of the family of ASIP processors in the application domain of reversible circuits. The Providing programmability together with required specific instructions has been the main purpose of the automated synthesis of reversible circuits. The proposed processor that we referred to as ARASP is a 16-bit processor with a total of 47 instructions totally, which some specific instruction has set for automated synthesis reversible circuits. ARASP is specialized for automated synthesis of reversible circuits using optimization algorithms such as GA or simulated annealing.

The design steps of all the main components inside the processor core have been described in detail. Maximum specific instruction, GNBRMUT, needs 29 clock cycles for execution. Structural VHDL code has been used to test the proposed architecture. A pipeline technique could be used to enhance the speed and achieve a high throughput rate as future work.

As future work, the processer can be comprehensively implemented of this processor that will specialize in simulated annealing algorithm. It is suggested that the proposed work will provide a new focus in the reversible field making hardware more specific for such applications.

Appendix

Table 3. ARASP instruction set

I_i	Instruction	Name	Description	Ins. Reference	IR(9)	OpCode
I_0	INP	Input	AC(L)←INPR	I/O		000000
I_1	OUT	Output	OUTR←AC(L)	I/O		000001
I_2	SKI	Skip if FGI	FGI: PC←PC+1	I/O		000010
I_3	SKO	Skip if FGO	FGO: PC←PC+1	I/O		000011
I_4	ION	IEN On	IEN←1	I/O		000100
I_5	IOF	IEN Off	IEN←0	I/O		000101
I_6	CLA	Clear Accumulator	AC←0	Register		000110
I_7	CLE	Clear E	E←0	Register		000111
I_8	CMA	Complement Accumulator	AC←$\overline{AC(L)}$	Register		001000
I_9	CME	Complement E	E←$\overline{E}$	Register		001001
I_{10}	INC	Increment Accumulator	AC←AC+1	Register		001010
I_{11}	ROL	Rotate Left Accumulator	AC←ROL AC	Register		001011
I_{12}	ROR	Rotate Right Accumulator	AC←ROR AC	Register		001100
I_{13}	SPA	Skip if Positive Accumulator	$\overline{S}$:: PC←PC+1	Register		001101
I_{14}	SZA	Skip if Zero Accumulator	Z: PC←PC+1	Register		001110
I_{15}	SZE	Skip if Zero E	$\overline{E}$: PC←PC+1	Register		001111
I_{16}	HLT	Halt	SC←Disable	Register		010000
I_{17}	AND	AND	AC← M[AR]∧AC	Memory		100000
I_{18}	OR	OR	AC← M[AR]∨AC	Memory		100001
I_{19}	XOR	XOR	AC ←M[AR]⊕AC	Memory		100010
I_{20}	ADD	Addition	AC(L)←M[AR]+AC	Memory	Int/Real	100011
I_{21}	SUB	Subtraction	AC(L)←M[AR]-AC	Memory	Int/Real	100100
I_{22}	MUL	Multiplication	AC(L)←M[AR]×AC	Memory	Int/Real	100101
i_{23}	DIV	Division	AC←AC/DR	Memory	Int/Real	100110
I_{24}	MOD		AC←AC % DR	Memory	Int	100111
I_{25}	POW	Power	AC←AC^{DR}	Memory	Int/Real	101000
I_{26}	EXP	ex	AC←e^{DR}	Memory	Real	101001
I_{27}	LDA	Load Accumulator	AC(L)←M[AR]	Memory		101010
I_{28}	STA	Store Accumulator	M[AR]←AC	Memory		101011
I_{29}	JMP	Jump	PC←AR	Memory		101100
I_{30}	BSR	Branch and Save Return-address	M[AR]←PC, PC←AR	Memory		101101
I_{31}	DSZ	Decrement and Skip if Zero	M[AR]←M[AR]-1 Z: PC←PC+1	Memory		101110
I_{32}	RAND	Generate a random number	AC←LFSR	Specific		010001
I_{33}	CLRNOG	Clear Number of Gate	NOG←0	Specific		010010
I_{34}	INCNOG	Increment Number of Gate	NOG←NOG+1	Specific		010011
I_{35}	SNOI	Set Number of Inputs	NOI←immediate	Specific		010100
I_{36}	SNOTTR	Set Number of Truth Table Rows	NOTR←2^{NOI}	Specific		010101
I_{37}	GRNDC	Generate Random Circuit	SyC[0]←Random Number SyC[NOG-1]←Random Number	Specific		010110
I_{38}	GNBRC	Generate a Neighbor of Circuit (Select a random gate and exchange its main control and one of its input)	NeC←Perturbing SyC	Specific		010111
I_{39}	SIVDTT	Set Initial Value for Destination Truth Table	STT[0]←0 . . . STT[2^{NOG}-1] ←2^{NOG}-1	Specific		011000
I_{40}	MASK	Generate a Mask For Output	MASK ←	Specific		011001
I_{41}	CALCSTT	Calculate Synthesis Truth Table		Specific		011010
I_{42}	CALCHD	Calculate Hamming Distance Between Synthesis Truth Table and Destination Truth Table	SHD/NHD ← Hamming Distance	Specific		011011
I_{43}	SBC	Select Best Circuit		Specific		011100
I_{44}	SETTEMP	Set Temperature	TEMP ← immediate	Specific		011101
I_{45}	DECTEMP	Decrement Temperature	TEMP ← TEMP -1	Specific		011110

References

[1] K. Kucukcakar, "An ASIP design methodology for embedded systems," in Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES'99)(IEEE Cat. No. 99TH8450), 1999: IEEE, pp. 17-21.

[2] M. Gries and K. Keutzer, Building ASIPs: The Mescal Methodology. Springer Science & Business Media, 2006.

[3] R. F. Mirzaee and M. Eshghi, "Design of an ASIP IDEA crypto processor," in 2011 IEEE 2nd International Conference on Networked Embedded Systems for Enterprise Applications, 2011: IEEE, pp. 1-7.

[4] K. Shahbazi, M. Eshghi, and R. F. Mirzaee, "Design and implementation of an ASIP-based cryptography processor for AES, IDEA, and MD5," Engineering science and technology, an international journal, vol. 20, no. 4, pp. 1308-1317, 2017.

[5] M. Venkanna, R. Rao, and P. C. Sekhar, "An Efficient Design of ASIP Using Pipelining Architecture," in International Conference on Intelligent Computing and Applications, 2019: Springer, pp. 117-128.

[6] D. Große, X. Chen, G. W. Dueck, and R. Drechsler, "Exact SAT-based Toffoli network synthesis," in Proceedings of the 17th ACM Great Lakes symposium on VLSI, 2007, pp. 96-101.

[7] D. Bu and P. Wang, "An improved KFDD based reversible circuit synthesis method," Integration, vol. 69, pp. 251-265, 2019.

[8] T. Ahmed, A. Younes, and A. Elsayed, "Improving the quantum cost of reversible Boolean functions using reorder algorithm," Quantum Information Processing, vol. 17, no. 5, pp. 1-16, 2018.

[9] A. Basak, A. Sadhu, K. Das, and K. K. Sharma, "Cost Optimization Technique for Quantum Circuits," International Journal of Theoretical Physics, vol. 58, no. 9, pp. 3158-3179, 2019.

[10] Z. Kalantari, M. Eshghi, M. Mohammadi, and S. Jassbi, "Low-cost and compact design method for reversible sequential circuits," The Journal of Supercomputing, vol. 75, no. 11, pp. 7497-7519, 2019.

[11] M. Lukac, M. Perkowski, and M. Pivtoraiko, "Evolutionary approach to quantum and reversible circuits synthesis," Artificial Intelligence Review Journal, vol. 20, no. 3–4, pp. 361–417, 2003.

[12] M. Lukac, M. Pivtoraiko, A. Mishchenko, and M. Perkowski, "Automated synthesis of generalized reversible cascades using genetic algorithms," 2002.

[13] M. Haghparast, M. Mohammadi, K. Navi, and M. Eshghi, "Optimized reversible multiplier circuit," Journal of Circuits, Systems, and Computers, vol. 18, no. 02, pp. 311-323, 2009.

[14] M. Mohammadi and M. Eshghi, "Heuristic methods to use don't care in automated design of reversible and quantum logic circuits," Quantum Information Processing, vol. 7, no. 4, pp. 175-192, 2008.

[15] M. Y. Abubakar and L. T. Jung, "Synthesis of Reversible Logic Using Enhanced Genetic Programming Approach," in 2018 4th International Conference on Computer and Information Sciences (ICCOINS), 2018: IEEE, pp. 1-5.

[16] T. Atkinson, A. Karsa, J. Drake, and J. Swan, "Quantum program synthesis: Swarm algorithms and benchmarks," in European Conference on Genetic Programming, 2019: Springer, pp. 19-34.

[17] R. Landauer, "Irreversibility and heat generation in the computing process," IBM journal of research and development, vol. 5, no. 3, pp. 183-191, 1961.

[18] C. H. Bennett, "Logical reversibility of computation," IBM Journal of Research and Development, vol. 17, no. 6, pp. 525-532, 1973.

[19] R. P. Feynman, "Quantum mechanical computers," Foundations of Physics, pp. 507-531, 1986.

[20] M. P. Frank, "Introduction to reversible computing: motivation, progress, and challenges," in Proceedings of the 2nd Conference on Computing Frontiers, 2005, pp. 385-390.

[21] E. Fredkin and T. Toffoli, "Conservative logic," International Journal of theoretical physics, vol. 21, no. 3, pp. 219-253, 1982.