ICEE2006 14th Iranian Conference on Electrical Engineering

Exploring the Design of A RISC Asynchronous Processor Using Persia Asynchronous Toolset

M. Mirzaaghatabar **M. Najibi** **K. Saleh** **H. Pedram**
{mirzaaghatabar,najibi, k.saleh, pedram}@ce.aut.ac.ir
Computer Engineering Department, Amirkabir University of Technology (Tehran Polytechnic)
424 Hafez Ave, Tehran 15785, Iran

Abstract: *Asynchronous microprocessors are more flexible to adapt to physical parameters, and have lower power consumption than synchronous microprocessors. In this paper we will introduce the design of an asynchronous microprocessor (V8-uRISC) and explore its design process compared to synchronous design. The processor is synthesized by Persia, an automatic tool for synthesizing asynchronous circuits. We have performed full functional test at various levels of design and synthesis. Our results show that an area overhead is expected for the asynchronous design as the cost for lower power and robustness.*

Keywords: Asynchronous processor, QDI, Persia, Synthesis

1 Introduction

Synchronous circuits have been in use for a long time. These circuits have many problems, including power consumption, vulnerability to changes in the environment, clock skew, and so on. The introduction of asynchronous circuits aims at solving some of these problems.

Although the concept of asynchronous circuits dates back to 1950's, this design method has not been able to acquire popularity due to hazard problems[1]. In recent years, a number of methods based on different timing models have been proposed to develop practical asynchronous circuits[2]. Asynchronous design methods are mainly categorized in two groups, namely, bounded delay, and delay insensitive. Considering the bounded delay models, the circuit delays are computed precisely, and accounted for in the design process. Regarding the delay insensitive models, synchronization between different sections is performed by generating and detecting request and acknowledgement signals.

Advantage of asynchronous circuits are as follows: eliminating the clock skew problem, modularity, lower power consumption, applying average delay instead of worst case delay, quick adaptation to newer technologies, and less vulnerability to changes in voltage and other environmental parameter such as temperature. As a result, an asynchronous processor has the potential to consume less power, expose better performance, and more flexible to adapt to voltage and other parameter changes in comparison to a synchronous processor. Due to these advantages, asynchronous processors have been used in several applications such as contact less intelligent cards[3].

In contrast to the mentioned advantages, there are some drawbacks regarding asynchronous circuits such as complex design procedures, and larger number of transistors. As a result, having an automatic asynchronous design tool is extremely helpful in popularizing asynchronous design methods. We have developed and we are continuing to optimize our *Persia* design tool in this regard. *Persia* is an asynchronous design synthesis tool based on the QDI[1] [4][5] timing model, and it can support GALS[2] design as well. In this paper we present the design of an asynchronous processor. The processor is synthesized by *Persia*[6], and the results are reported.

[1] Quasi Delay Insensitive
[2] Globally Synchronous Locally Asynchronous

In section 2, *Persia* synthesis tool is introduced. Section 3 describes the architecture of V8-uRISC processor, along with its asynchronous design. Section 4 focuses on synchronous and asynchronous design methods, taking advantage of V8-uRISC processor as a tested for comparison, and section 5 presents the synthesis results.

2 Persia Synthesis Tool

Persia is an asynchronous synthesis toolset developed for automatic synthesis of QDI [1] asynchronous circuits with adequate support for GALS [2] [7] systems. The structure of Persia is based on the design flow shown in Figure 1 which can be considered as the following four individual portions: QDI synthesis, GALS synthesis, layout synthesis, and simulation at various levels. QDI and GALS synthesis flows are joint together in the layout stage. The simulation flow is intended to verify the correctness of the synthesized circuit in all levels of abstraction.

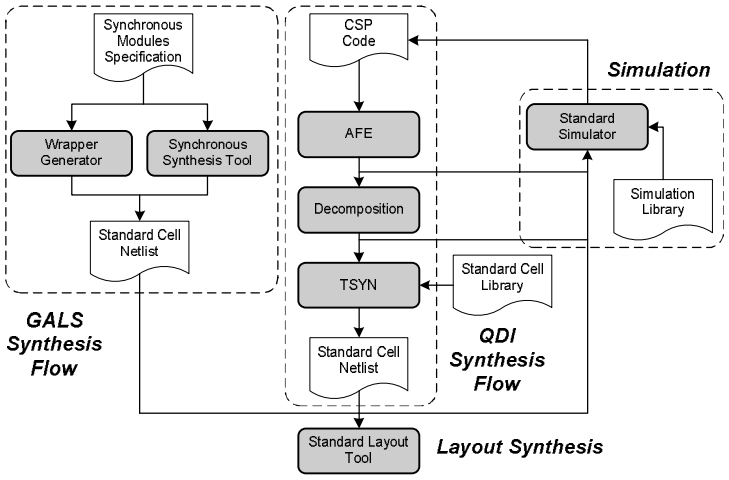

Figure 1: QDI and GALS design flow

CSP is a well-known language for description of concurrent systems which is accepted as a good description language for asynchronous systems. A Circuit in CSP is described as the composition of distinct processes that run in parallel and communicate with each other on channels by message passing. Persia uses Verilog-CSP[3][8], an extension of the standard Verilog which supports asynchronous communication as the hardware description language for all levels of abstractions except the net list which uses standard Verilog. This way the Verilog is powered by some READ and WRITE PLI [4] macros to emulate CSP language communication actions on the channels. The input of Persia is a Verilog description of a circuit that includes READ and WRITE macros for sending and receiving data via communication channels. This description will be converted to a netlist of standard-cell elements through several steps of QDI synthesis flow. For simpler synthesis first arithmetic operations are extracted from the code and the major steps of synthesis only works on the codes without any arithmetic operations. This is done by the AFE which also replaces the arithmetic functions by standard library modules. The two major steps in Persia synthesis are Decomposition and TSYN. In the following subsections we briefly describe the functionality of these three stages.

2.1 AFE

Technology-Mapper, as a part of Template Synthesizer, is only able to synthesize one-bit assignments containing logical operators like AND, OR, XOR, etc. Arithmetic operations are not synthesizable by TSYN, so Persia extracts these operations from the CSP source code and then implements them with pre-synthesized standard templates. This is the role of the first stage of our asynchronous synthesis flow, called Arithmetic Function Extractor or AFE.

AFE extracts each assignment that contains arithmetic operations like addition, subtraction, comparison, etc and generates a tree of standard

[1] Quasi Delay Insensitive
[2] Globally-Asynchronous Locally-Synchronous
[3] Communicating Sequential Processes[9]
[4] Programming Language Interface

circuits which implements the extracted assignment. The communication between the main circuit and the arithmetic circuit is made by introducing new channels and added READ/WRITE macros. As a result, the main circuit will contain only logical assignments and arithmetic computations will be performed in standard unconditional modules that are designed and included in the library. Figure 2 shows the implementation of (a + b) – c using standard unconditional adder and subtractor modules.

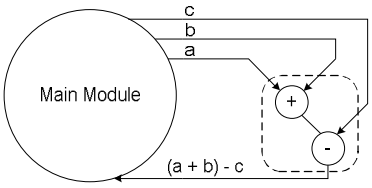

Figure 2: Extracting Arithmetic Function

2.2 Decomposition

Our synthesis approach is based on pre-designed asynchronous four-phased dual rail templates. Each template can be considered as a simple pipeline stage that can has the ability of conditional reading of its inputs our conditional writing to the outputs. The most renown form of these templates are named as PCFB and PCHB0[10].

The high-level CSP description of even very simple practical circuits is not directly convertible to PCFB/PCHB[10]0 templates. The intention of Decomposition stage is to decompose[11] the original description into an equivalent collection of smaller interacting processes that are compatible to these templates and are synthesizable in next stages of QDI synthesis flow. Decomposition also enhances the parallelism between the resultant processes by eliminating unnecessary dependencies and sequences in the original CSP description.

2.3 TSYN

Template Synthesizer, as the final stage of QDI synthesis flow, receives a CSP source code containing a number of PCFB-compatible modules and optionally a top-level netlist and generates a netlist of standard-cell elements with dual-rail ports that can be used for creating final layout. TSYN can synthesize all logical operations including AND, OR, XOR, etc with conditional or unconditional READ and WRITES. In addition, TSYN adds acknowledge signals to I/O ports and converts the top-level netlist to dual-rail form and makes appropriate connections between ports and acknowledge signals. The output of TSYN can be simulated in standard Verilog simulators by using the behavioural description of standard-cell library elements.

3 Asynchronous V8-uRISC 8-bit Microprocessor

In this section, first the V8-uRISC, a simple 8-bit RISC Microprocessor is introduced and then asynchronous design flow of this microprocessor is discussed.

3.1 V8-uRISC Microprocessor Description

The V8-uRISC[12] 8-bit RISC microprocessor is a synchronous general purpose microprocessor which has a small gate count with single clock cycle execution for many instructions to deliver a high performance 8-bit microprocessor with a very small footprint. The V8-uRISC is ideal for 8-bit applications which need a combination of small size, good performance and low power.

The V8-uRISC is supported by a complete set of development tools for every stage of software development, such as a full ANSI C compiler[13], assembler, software simulator and debugger. The V8-uRISC uses eight 8-bit registers to perform all arithmetic and logical operations. Register R0 can be considered as an accumulator and registers R1 through R7 are general purpose.

This processor has a *Von Neuman* structure. It has seven maskable and one non-maskable interrupts. There are 33 opcodes, 4 addressing modes, and 2 user defined opcodes, 8-bit ALU, 16-bit Program Counter and Stack Pointer.

The V8-uRISC instructions have three formats which are given in Table 1. The first byte of all instructions contains opcode and register's number. 76% of the instructions are single byte instructions, such as INC, ADDC and TX0. Two byte instructions that are 14% of all instructions also contain an offset, such as conditional jump. Finally, three byte instructions, 10% of all instruction, contain two bytes address, such as functions or interrupt service routine address.

Table 1: V8-uRISC Instruction Format

7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Opcode					REG																		
Opcode					REG			Byte1															
Opcode					REG			Byte1								Byte2							

3.2 Asynchronous Microprocessor Architecture

As mentioned before, asynchronous circuits are clock-less and different modules interact with each others by handshake signalling. The first step of designing an asynchronous processor or in general any asynchronous circuit is determining the different modules of the circuit. With consideration of instruction's execution flow in V8-uRISC, the microprocessor is partitioned into these modules: Program Counter, Instruction Memory, Decode, Stack Pointer register, Register File (contains 8 register R0 to R7), Data Memory, Execution, Read/Write Control, two Adders one for Execution module and the other for Decode module. The two adder modules are designed in order to reducing the number of adders in circuit after synthesis of circuit. The data path of asynchronous microprocessor is given in figure 3.

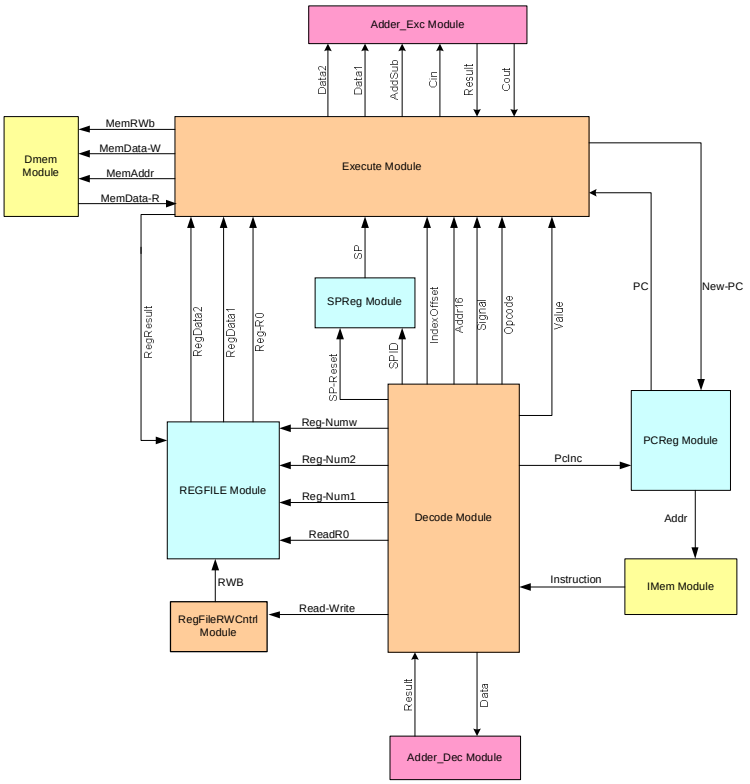

Figure 3: Data path of Asynchronous V8-uRISC Microprocessor

Different modules are related with each other by asynchronous channels and which are connected to their ports. Data communication is done by writing data to the ports and read it from the corresponding port on the other side of the channel. For write we use the following Verilog write macro:

```
'WRITE (Port name, value)
```

If the sender wants to write another data on that port, it would be suspended until the last data read from that port as follows:

```
'READ (Port name, value)
```

The receiver module also remains suspended until a data is written on its counterpart port. Read and Write operation will be implemented by handshake signalling.

Like every other processor, at first an instruction must be fetched from an instruction memory, then it must be decoded and finally executed. The *Program Counter* module, as shown in Figure 4, has two input ports named as PCInc and New_PC, and two output ports, Pc and Pc_exe.

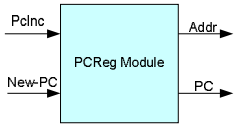

Figure 4: Program Counter module

After initialization of program counter, the PC module waits for a data to be written on the PCInc

port. While there is not a valid data on PCInc port the module would be suspended. After receiving the data on PCInc port, the module will inspect it, and made special decision respect to the PCInc value. For example, if the last instruction is an unconditional jump, the module would assign a new value to program counter register which is read from New-PC port otherwise the program counter register would increase by one to fetch the next instruction or the current instruction's operand. This program is shown in figure 5.

```
always
 begin
   `READ (PCInc, Inc)
   if (Inc == 0)
     `READ (New_PC, Addr)
   else
     Addr = Program_counter + 1;
     Program_counter = Addr;
     `WRITE (Pc, Program_counter)
    if (Inc != 2)
     `WRITE(Pc_exe, Program_counter)
 end
```

Figure 5: Program Counter module program

After writing the proper address on *Addr* port by the *Program Counter* module, this value is read by *Instruction Memory* module which then fetches the instruction or operand from instruction memory and puts it on the 8-bit output port *Instruction*. The structure of this module is shown in figure 6.

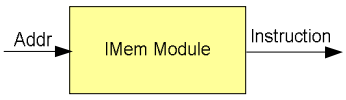

Figure 6: Instruction Memory module

This module has a simple program which is shown in figure 7.

```
always
 begin
   `READ(Addr, Pc)
   x = Ins_Mem [Pc];
   `WRITE(Data,x)
end
```

Figure 7: Instruction Memory module program

The fetched instruction is given to the *Decode* module, which is responsible for decoding the instruction and puts control signals on ports of other modules such as *Program Counter* module or *Execution* module. In similar way, writing on control of data ports of other modules initiated other modules in the asynchronous processor which in turn do proper actions with respect to data values.

For example consider instruction INC R_n which increments the register R_n by one. After decoding this instruction in *Decode* module control signals is made and written on special ports as shown in figure 8.

```
WRITE(Opcode, instruction[5:0])
`WRITE(Signal, i[7:6])
if (instruction [7:3] == 0)//Rn = Rn + 1
 begin
   `WRITE(RegRead_Write, 3)
   `WRITE(ReadR0, 0)
   `WRITE(RegN1, instruction [2:0])//used
   `WRITE(RegN2, instruction [2:0])//no
use
   `WRITE(RegNw, i[2:0])
   `WRITE(PCInc, 1)
end
```

Figure 8: Control Signals

Writing the value of *instruction [2:0]*, the number of register, on the *RegN1* port means that the *Decode* module selects that register from *Register File* module. The value of that register will be read by *Executing* module and increased by one and finally stored to that register in *Register File* module.

Totally, in asynchronous circuits design, the designer must determine the different modules of the circuits and their proper interconnection through special input/output ports that needed data would be transferred by writing on and reading from these ports, as mentioned for this microprocessor.

4. Synchronous vs. Asynchronous Design Flow: V8-uRISC Example

Asynchronous design of a circuit is usually done from the scratch and it requires changing designers' way of thinking about synchronization. The most remarkable differences between asynchronous design techniques in contrast to synchronous design techniques are arisen by elimination of the global clock which has a significant effect on the design method. As in synchronous methodology, transferring data from one module to another requires an implied agreement between the sender and receiver, in asynchronous methodology data transfer itself also takes care of synchronization. As an example, in a synchronous processor for a data exchange to take place both sender and receiver designs are affected; the designer must implement the agreement such that the state of both sender and receiver at the moment of the data transfer to be correct. In contrast, an asynchronous designer is

only required to write the data value on the output port of the sender regarding only the state of the sender, and read the port of the receiver considering the receiver's state only; no inter-module agreement is needed.
This kind of independency in designing of separate modules in the system has many benefits for the designer:

- Simpler design can lead to less time-to-market for complex designs
- Independent design of the modules increases the modularity and design reuse
- Scalability and incremental design are also improved

The following code samples, which are simplified version of the fetch unit in synchronous and asynchronous processors, demonstrate how asynchronous communication can simplify the design. The code fragment in figure 9 shows the RTL code of the fetch unit for a hypothetical synchronous processor.

```
R'T0: IR ← MEM [PC], PC ← PC + 1,…
(A+B)R'T1: IR ← MEM [PC], PC ← PC + 1,…
B R'T2: IR ← MEM [PC], PC ← PC + 1,…
T3 : . . .
```

Figure 9: RTL description of fetch

Control signals A and B respectively show two and three byte instructions, and T0, T1,… indicate sequential cycles of the global clock. Signal R indicates external interrupts which is not involved in our discussions. As it can be seen in the code for this simple design, all one, two, or three byte instructions take three cycles to start execution. It means that 2 cycles of the fetch can be useless so some performance degradation is expected. Expert synchronous designers however can use a more complicated approach in the fetch unit to avoid this performance loss but only at the cost of increasing complexity. The equivalent asynchronous specification of the same fetch unit is shown in figure 10.

```
‘Read(…)
if( op == 2 )
  begin
  ‘Read(…)
  . . .
end
else if( op == 3 )
  ‘Read(…)
  ‘Read(…)
  . . .
end
```

Figure 10: CSP description of fetch

In asynchronous fetch unit for all one, two, or three byte instructions execution is started right after the completion of the fetch operation.
Applying a change in the asynchronous code is as simple as adding a `READ or `WRITE operation to the code while in synchronous code the designer must deal with more difficulties especially for more complex state machines. For illustration consider the situation of adding a new instruction to a pre-designed processor. For asynchronous version it is only needed to add the following code fragment and there is no other change required.

```
else if( op == 4 )
  . . .
  ‘Read(…)
  ‘Read(…)
  ‘Read(…)
  . . .
end
```

Figure 11: Added code

As explained in this section, the nature of asynchronous circuits leads to simpler and more scalable codes in contrast to synchronous codes. This is mainly resulted from eliminating the need for global synchronization by means of local handshaking, and distributed nature of the control parts of asynchronous circuits.
However asynchronous circuits can not be synthesized without powerful automatic synthesis tools. This fact that implementation of asynchronous circuits is generally a more complex task emphasizes the role of automatic synthesis.

5. Result

In this section, we'll present the synthesis result of the decoder module using the *Persia* synthesis tool. We can see the synthesis result in Table 2. The first column of table contains the cell names in *Persia* synthesis tool library[14], the second column shows the number of each cells in the design, and the third column shows the number of transistor in each cell.
We need 37125 cells or equivalently 269610 transistors to implement the decoder module. *Persia* output contains a net list of its standard library cell which can be implemented by available layout tools. *Persia* is the first available fully automatic synthesis tool for asynchronous circuits, however it is not completely optimized; more optimization is needed especially after process decomposition. As one of the major

optimization tasks we mention the process recombination of fine grain processes resulted from the decomposition phase [15]. This optimization eliminates most of the unnecessary completion detection circuits which comprises nearly half of the small sized PCFB modules. Following the optimization process the number of transistors and the area of circuit is expected to be higher than the synchronous circuit. This overhead is the cost we for asynchronous advantages such as: lower power consumption.

Table 2: Decoder module synthesized result

Module	Inst.	Trans.
GATE_INV	4504	2
VDD_GATE	11600	0
GND_GATE	2982	0
GATE_NOR2	1471	4
C_Element5	481	14
C_Element3	284	10
C_Element2	656	8
GATE_NAND2	2093	4
Asym_C_Element2	233	4
C_Element2_Reset1	16	9
InputAck	664	11
GATE_OR2	214	6
C_Element2_Reset0	16	9
GBuf	4673	19
cGBuf	3051	16
InitBuf	172	20
cWiredOr2	335	18
WiredOr2	3027	21
cOr2	254	18
C_Element4	56	12
cNand2	324	18
PCFB_Binary_Comparator_EQ8	19	0
Total:	**37125**	**269610**

6. Conclusion

This paper demonstrates the design of an eight bit asynchronous microprocessor (V8-uRISC) from its architectural design, to behavioural coding and synthesis. We also mentioned major differences in synchronous and asynchronous designing styles by the real examples encountered in design of V8-uRISC. Due to the elimination of a single global clock, synchronization between different parts is easier, and this leads to more scalable designs. In contrast, synchronization in synchronous designs is achieved by the clock signal that makes scalability a difficult task. We also demonstrated that how communication between asynchronous modules is performed by handshaking signals on the ports.

Finally we utilized the Persia asynchronous design tool to synthesize the decoder module of V8-uRISC microprocessors, and demonstrated the overhead as a cost for the advantages that we achieved.

References

[1] S.H. Unger, “Asynchronous Sequential Switching Circuits”, *Wiley-Interscience*, New York, 1969

[2] Scott Hauck, “Asynchronous design methodologies: An overview”, *Proceedings of the IEEE*, 83(1):69-93, January 1995

[3] A. Abrial, J. Bouvier, M. Renaudin, P. Vivet, “A Contactless Smart-Card Chip based on an Asynchronous 8-bit Microcontroller”, *France Telecom*, 2000.

[4] J. Sparso, S. Furber, “*Principles of Asynchronous Circuit Design – A System Perspective*”, Kluwer Academic Publishers, 2002.

[5] Alain J. Martin, “Synthesis of Asynchronous VLSI Circuits”,*Caltech, CS-TR-93-28*, 1991.

[6] http://www.async.ir

[7] J. Muttersbach, T. Villiger, and W. Fichtner. Practical Design of Globally-Asynchronous Locally-Synchronous Systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 2000.

[8] A. Seifhashemi, H. Pedram, "Verilog HDL, Powered by PLI: a Suitable Framework for Describing and Modeling Asynchronous Circuits at All Levels of Abstraction", *Proc. Of 40th DAC, Anneheim, CA, USA*, June 2003

[9] C. A. R. Hoare, “Communicating Sequential Processes”, *Communication of ACM 21, 8, pp 666-667*, 1978

[10] A.M. Lines. Pipelined Asynchronous Circuits. *M.Sc. Thesis, California Institute of Technology*, June 1995, revised 1998.

[11] C. G. Wong , A. J. Martin, "Data-Driven Process Decomposition for the Synthesis of Asynchronous Circuits", Proceedings of ICECS, 2001

[12] http://www.ece.neu.edu/info/vhdl/class/actual/FullDocument.htm

[13] http://www.htsoft.com/

[14] K. Saleh, **“**Automatic Synthesis of PCFB Templates to Standard-Cell Library (TSYN)”, Technical Report AUT, May 2005

[15] K. Saleh, H. Pedram, M. Naderi, “Power Reduction Techniques for Asynchronous Circuits”, *Proceedings of the 12th Iranian Conference on Electrical Engineering (ICEE2004)*, May 2004