國立交通大學

電機資訊學院 資訊學程

碩 士 論 文

非同步 AVR 微控器設計及實現

Design and Implementation of an Asynchronous AVR microcontroller

研 究 生：吳信儒

指導教授：陳昌居 教授

中 華 民 國 九 十 四 年 七 月

非同步 AVR 微控器設計及實現

Design and Implementation of an Asynchronous AVR microcontroller

研 究 生：吳信儒　　Student：Wu Shin-Ru

指導教授：陳昌居　　Advisor：Chen Chang-Jiu

國 立 交 通 大 學
電機資訊學院 資訊學程
碩 士 論 文

A Thesis
Submitted to Degree Program of Electrical Engineering Computer Science

College of Electrical Engineering and Computer Science

National Chiao Tung University
In Partial Fulfillment of the Requirements
For the Degree of
Master of Science
In
Computer Science
June 2005
Hsinchu, Taiwan, Republic of China

中華民國九十四年七月

非同步 AVR 微控器設計及實現

學生：吳信儒　　　　　　　　　　　指導教授：陳昌居 教授

國立交通大學電機資訊學院 資訊學程（研究所）碩士班

摘　　　要

非同步電路有諸多的優點，例如時脈歪斜問題的消除、平均效能的表現、以及低功率消耗的特性，在在吸引著我們去深入探討這樣特性的電路結構。尤其是低消耗功率的特性， 對於現今行動裝置產品而言是非常重要的。

本篇論文針對 ATMEL 公司 8 位元 RISC 架構的 AVR 微控制器，以非同步電路的方式將其重新設計與做部分的實現。

我們採用 Sutherland 的微管線（Micropipeline）架構做為設計基礎，修改成為雙軌（dual rail）與延遲無關（delay insensitive）的電路設計，將整個非同步處理器設計結合目前的同步電路工具進行合成， 並將設計下載至 FPGA 晶片以完成實現。

我們完成了算術及邏輯運算指令如 *ADD*、*SUB*、*AND*、*OR*，資料搬移指令如 *MOV*、*LSI*，分支指令如 *JMP* 等主要的指令，以建構成我們的非同步版本的 AVR 微控制器。

Design and Implementation of an Asynchronous AVR microcontroller

Student：Wu Shin-Ru Advisor：Dr. Chen Chang-Jiu

Degree Program of Electrical Engineering Computer Science

National Chiao Tung University

ABSTRACT

There are many advantages which allure us to explore contents of asynchronous circuits on asynchronous circuits, such as elimination of clock skew problems, average case performance and lower system power consumption. Furthermore, low power consumption is one of the most important issues for mobile devices.

This thesis focuses on the implementation of an asynchronous Atmel AVR 8-bit RISC microcontroller.

We implement our design with Sutherland's dual rail DI micropipeline. The whole asynchronous AVR microcontroller was synthesized with current synchronous EDA tools, and realized with FPGA chip.

The following instructions were accomplished: arithmetic and logic instructions, such as *ADD*, *SUB*, *AND*, *OR*; data transfer instructions, such as *MOV*, *LDI*; branch instructions, such as *JMP*. These instructions were used to comprise our asynchronous AVR microcontroller.

Acknowledgements

I am deeply indebted to the following persons.

Dr. Chang-Jiu Chen– the prime advisor on my research and this thesis.

Dr. Fu-Chiung Cheng– introduces the concepts of asynchronous system to me.

My all good friends–talk and discuss about my research or everything.

My mother– gives me the best supports and encouragements.

My wife and my sons – I want to thank my wife Trista Jiang and my son Qian-Han;

Without their patience, concern, and efforts, this research would not have been possible.

And all other people who ever helped me.

Contents

ABSTRACT IN CHINESE 3
ABSTRACT 4
Acknowledgements 5
Contents 6
List of Tables 8
List of Figures 9
Chapter 1 Introduction 10
1.1 Motivations 10
1.2 Contributions 10
1.3 The organization of this thesis 11
Chapter 2 Background 12
2.1 The AVR architecture 12
2.1.1 The AVR architecture overview 12
2.1.2 AVR organization 14
2.1.3 The AVR instruction set 15
2.2 Asynchronous circuits 16
2.2.1 Signaling protocol 16
2.2.2 Data encoding 18
2.2.3 Classes of asynchronous circuits 19
2.2.4 Asynchronous circuits' advantages 20
2.3 Micropipeline 22
2.4 Data dependency graph 24
Chapter 3 Design and implementation 26
3.1 The asynchronous AVR microcontroller architecture 29
3.1.1 Our asynchronous Micropipeline circuits 30
3.1.2 Instruction fetch stage (IF) 31
3.1.3 Instruction decode stage (ID) 31

3.1.4 Execution stage (EX)...32
3.1.5 Write back stage (WB)...33
3.1.6 The register file...34
3.2 Design methodology...34
3.2.1 Data dependency graph of AVR microcontroller...35
3.2.2 Design steps...36
3.2.3 Design environment...37
3.3 Implementation...37
3.3.1 FPGA design issues for asynchronous logic...37
3.3.2 Implementation environment...37
Chapter 4 Testing...39
4.1 Testing configuration introduction...40
4.2 16 bits instruction composition...41
4.3 Memory interface...42
4.3.1 Dual-rail to single-rail circuit...42
4.3.2 Single-rail to dual-rail circuit...43
4.3.3 Return to zero circuits...44
4.4 HDL verification...44
4.5 Physical circuits validation...45
Chapter 5 Conclusions...47
References...49

List of Tables

Table 1. The Design Environment tools 36
Table 2. The Sequence of instruction execution 39
Table 3. The test program of looping addition 45
Table 4. The Implemented AVR instructions 48

List of Figures

Figure 1. AVR Memory Map 13
Figure 2. AVR organization 14
Figure 3. 4 cycle asynchronous signaling protocol 16
Figure 4. 2 cycle asynchronous signaling protocol 17
Figure 5. Two phase bundled data convention 18
Figure 6. Event controlled storage elements........ 22
Figure 7. Muller C element and its properties 23
Figure 8. Micropipeline with 4 stages 24
Figure 9. Five basic elements of data dependency graph and its mapping circuit........ 25
Figure 10.Qelement and its procedure........ 26
Figure 11.The organization of asynchronous AVR microprocessor 28
Figure 12.Dual rail , 4 phase , Delay insensitive Micropipeline with 4 stages 29
Figure 13.The organization of Instruction Fetch Stage 30
Figure 14.The organization of Instruction Decode Stage 31
Figure 15. The organization of Execution Stage 32
Figure 16. The organization of Write Back Stage 33
Figure 17. The instruction content of ADD instruction 34
Figure 18. The DDG of ADD instruction........ 35
Figure 19. The Design and Implementation procedure 36
Figure 20. xilinx prototype board 38
Figure 21. Testing configuration 40
Figure 22. EEPROM 16-bit word configuration 42
Figure 23. Two Rail To One Rail Circuit........ 43
Figure 24. One Rail To Two Rail Circuit........ 43
Figure 25. Return to zero circuits 44
Figure 26. Simulation result in Modelsim SE/EE PLUS 5.4........ 45
Figure 27. Xilinx prototype board with virtex E FPGA Chip........ 46
Figure 28. I/O Card 46

Chapter 1 Introduction

1.1 Motivations

Because of clock less characteristics, the major advantages of asynchronous logic are average case performance and lower power consumption. These advantages that become the most important issues for future applications, especially for 3C products are fully being suitable for embedded system design. That's why we designed this asynchronous processor core to accommodate the advantages above.

The instruction set selected is AVR. AVR is a very popular RISC architecture developed by Atmel. In fact, AVR and 8051 are popular microcontroller for embedded system. However, compared to 8051, AVR has better performance than 8051, almost 12 times faster than the 8051 in synchronous version.

After finishing our asynchronous circuits design, we need to simulate our design to validate its functionality. We used FPGA chip, a flexible programmable chip, to implement our design to make sure it really works.

1.2 Contributions

In this thesis, we propose the design and implementation of an asynchronous AVR microcontroller. We describe the behavior of AVR instruction by using data dependency graph. We also establish several asynchronous FPGA cell libraries in Verilog. Once the design methodology established, we can use the same way to survey and implement other asynchronous CPU core. Finally, we implement this design into the FPGA chip and confirm that the asynchronous design really works.

1.3 The organization of this thesis

In first chapter, we present the motivations and contributions of this research. In chapter 2, we give the background of the AVR architecture, the instruction set, the asynchronous circuits' classification, and the limitations of asynchronous circuit implementation with synchronous FPGA chip. In chapter 3, we propose the architecture of asynchronous AVR microprocessor, design methodology and the existing synchronous tools used in our design. In chapter 4, we introduce the testing environment of our design. Chapter 5 concludes this thesis.

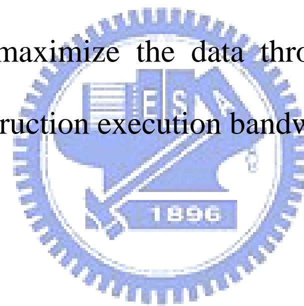
Chapter 2 Background

2.1 The AVR architecture

The AVR microcontroller is a Reduced Instruction Set Computer (RISC) architecture designed by Atmel Norway. Its predecessor is μRISC, first developed in a Diploma Thesis at NTH (NTNU). This core is a standard 8-bit microcontroller, used widely in Atmel products. [7]

2.1.1 The AVR architecture overview

The AVR architecture has two major advantages. The first of all, it provides a load and store multiple instruction to maximize the data throughput. Second, it uses the Harvard architecture to improve the instruction execution bandwidth.



AVR registers

- Program Counter

Program Counter can be used as a pointer to the instruction being executed. Most AVR instructions are 2 bytes long (one 16-bit word). The PC contains 12 bits in AT90S8515 family. With the relative jump and call instructions, the whole 4K address space can be directly accessed.

- Status register

Status Register (SREG) storing the ***ALU*** operation result and can be referenced by the next instruction. SREG is 1 bytes long (one 8-bit), that contains carry, zero, negative, two's complement overflow indicator, condition bit for signed tests instruction set, half carry, transfer bit and global interrupt enable/disable indicator.

- General purpose register

The AVR core has 32 general purpose working registers. All the 32 registers are directly connected to the arithmetic logic unit (***ALU***), allowing two independent registers to be accessed in one single instruction. The data communication with two general purpose registers can be finished in one instruction. Six of the 32 registers can be used as three 16-bits indirect address register pointers for Data Space addressing - enabling efficient address calculations. One of the three address pointers is also used as the address pointer for the constant table lookup function. These added function registers are the 16-bits X-register, Y-register and Z-register.

・I/O register

The I/O memory space contains 64 addresses for CPU peripheral functions for Control Registers, Timer/Counters, A/D-converters, and other I/O functions. The I/O Memory can be accessed directly. In addition, the AVR uses the Harvard architecture concept that separates memories and buses for program and data.

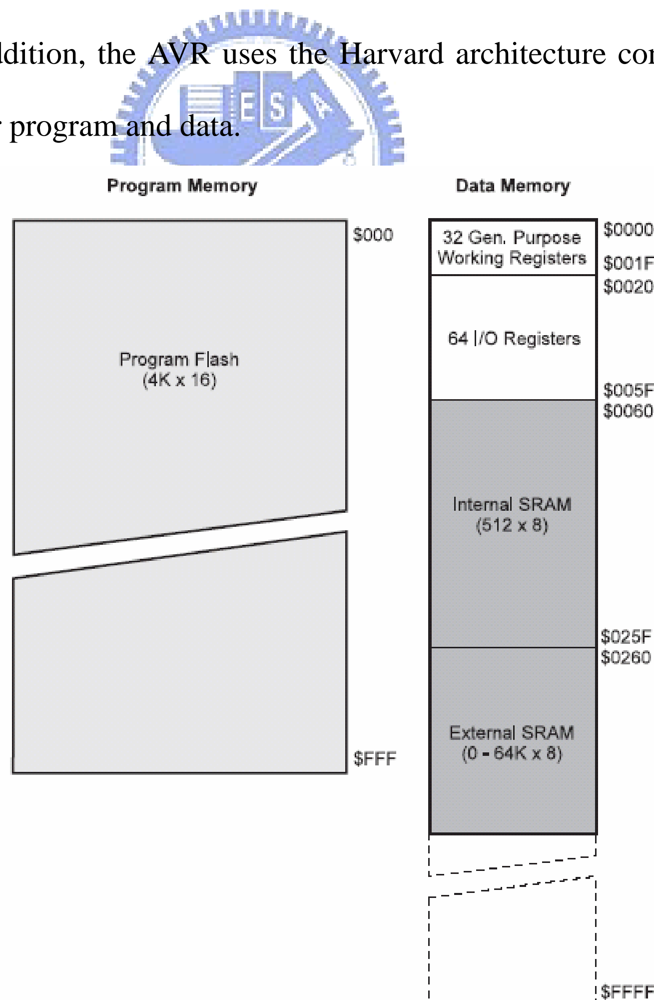


Figure 1 .AVR Memory Maps

2.1.2 AVR organization

Figure 2 shows the organization of AVR [1]. The main components are described as followings:

- The program counter as described above, used as a pointer to the instruction being executed.
- 32 general purpose registers, which store the temporary results. It has two read ports and one write port which can be used to access any register.
- The ***ALU***, supports arithmetic and logic functions between registers or between a constant and a register. Single-register operations are also executed in the ***ALU***.
- The instruction register, the codes are executed with a two stage pipeline. While one instruction is being executed, the next one is pre-fetched from the program memory and stored here.
- The instruction decoder, accepts the instructions from instruction register and issue essential control signals to the CPU and peripheral resources.
- The RAM used as I/O register space, stack space and other operation purposes.

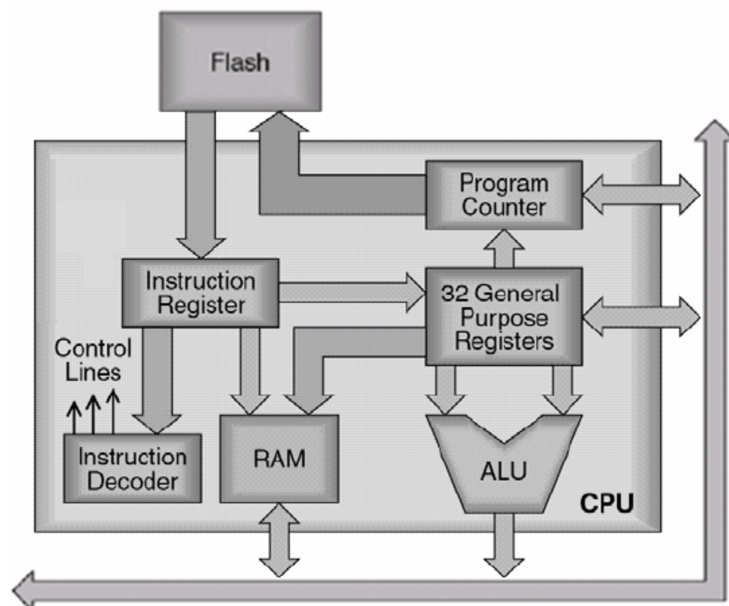


Figure 2 AVR Organization

2.1.3 The AVR instruction set

AVR employs the load-store architecture, which means that the value will be processed inside the registers and the result will be stored into a register. The only operations for memory access are LOAD and STORE instructions.

AVR instruction set categories

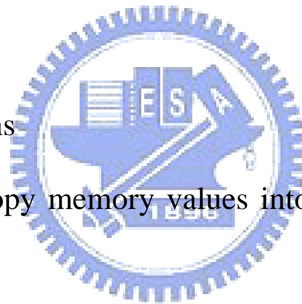
AVR instructions fall into the following four categories:

- Arithmetic and logic instructions

  These instructions can only use the value inside the registers for computation. For example, an *ADD* instruction can add two values in two registers and place the result in another register.

- Data transfer instructions

  These instructions copy memory values into registers or store register values into the memory.



- Branch instructions

  Branch instructions cause the execution path to switch to a different address.

- Bit and bit test instructions

  These instructions can change the register value in bit element or set the status register flag content to affect the next instruction executing.

2.2 Asynchronous circuits

Circuit design styles can be classified into two major categories, namely synchronous and asynchronous. Synchronous circuits can be simply designed as circuits which are sequenced by one or more global, distributed periodic timing signals called clocks. Asynchronous circuits use special "Handshaking" protocol to communicate with each other.

2.2.1 Signaling protocol

Most asynchronous circuit signaling schemes are based on some sorts of protocols involving requests which are used to initiate an action and corresponding acknowledgments, to signal the completion of that action. These control signals provide all of the necessary sequence controls for computational events in the system. Strictly speaking these handshake signals are independent of any global system time and are only concerned with the local relative temporal relationships between two sub circuits sharing a common interface.

There are several choices of how these alternating events are encoded into specific wires. [1]

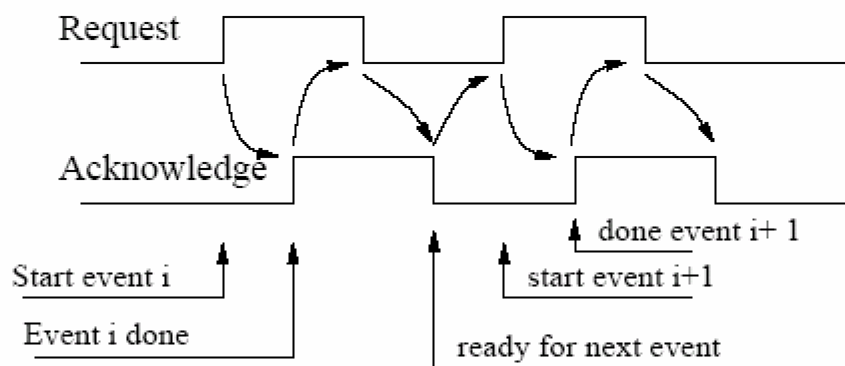


Figure 3 4 phase asynchronous signaling protocol

One common choice is the 4-phase protocol shown in Figure 3. Other names for this protocol are also in common use: return to zero (RZ), 4-phase and level signaling.

In this protocol, 4 transitions (2 on the request and 2 on the acknowledgement) are required to

complete a particular event transition.

4-phase proponents argue that the falling (return to zero) transitions on the request and acknowledgement lines do not usually cause performance degradation because falling transition can be happened in parallel with other circuit operations.

The other common choice is 2-cycle signaling shown in Figure 4, also called transition, 2-phase, or NRZ (non-return to zero) signaling. In this protocol, the waveforms are the same as for 4-phase signaling with the exception that every transition on the request wire, both falling and rising, indicates a new request. The same is true for transitions on the acknowledgement wire.

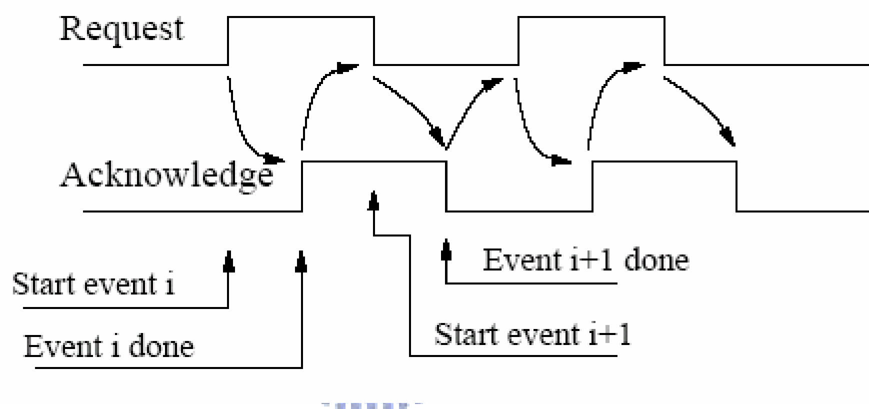


Figure 4 2 phase asynchronous signaling protocol

2-phase proponents argue that 2-cycle signaling is better on both the power consumption and performance standpoints, since every transition represents a meaningful event and no transitions or power are consumed in returning to zero, since there is no resetting of the handshake link.

While in principle aspect this is true, it is also important to notice that most 2-phase interface implementations require more logic than their 4-phase equivalents. The increasing logic complexity may consume more power than that is saved by reducing control transitions.

2.2.2 Data encoding

There are two common choices for how to encode data, one is the use of bundled protocol with either 2- or 4-phase signaling [2].

In this case, for an n-bit data value to be passed from the sender to the receiver, n+2 wires will be required (n bits of data, 1 request bit, 1 acknowledgement bit). The constraint of this protocol is that the propagation times of the control and data lines are either equal or that the control propagates slower than the data signals. If this were not the case, the receiver could initiate the requested action with incorrect values. Fig 5 illustrates the procedure.

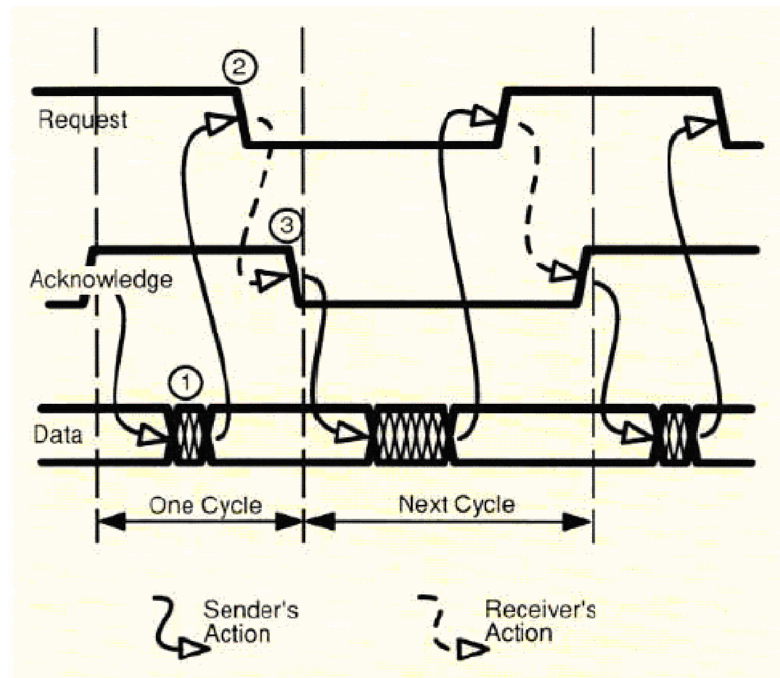


Figure 5 Two phase bundled data convention

The common alternative to the bundled data approach is dual rail encoding. In this case, data and control signals are not separated onto distinct wire paths. Instead, with the dual-rail approach a bit of data is encoded with its own request onto 2 wires. In this case, for an n-bit data value, the wire complexity is 2n+1 wire: 2n for the data and the requests plus an additional acknowledgement signal. In a four cycle variant of this dual rail protocols, sending a bit requires the transition from the idle state to either the valid 0 or valid 1 state and then,

after receiving the acknowledgement, it must transition back to the idle state. The acknowledgement wire must be reset prior to a subsequence assertion of a valid 0 or 1.

Dual rail signaling is insensitive to the delays on any wire and therefore is more robust while assumptions like the bundling constraint cannot be guaranteed. The receiver will need to check for validity of all n-bits before using the data or asserting the acknowledgement.

The downside of the dual-rail approach is often the increasing complexity in both wiring and logic.

2.2.3 Classes of asynchronous circuits

Asynchronous circuits can be divided into the following classes [2].

A delay-insensitive (DI) circuit is designed to operate correctly regardless of the delays on its gates and wires. That is, an unbounded gate and wire delay model is assumed. The concept of a delay insensitive circuit grows out of work by Clark and Molnar in the 1960's on Macro modules [16]. The DI systems have been formalized by [10] and [4]. Many practical DI circuits can be built if one allows more complex components [11, 12]. A complex component is constructed out of several simple gates.

A quasi-delay-insensitive (quasi-DI or QDI) circuit is delay-insensitive except that "isochoric forks" are required [14]. An isochoric forked wire where all branches have exactly the same delay. In contrast, in a DI circuit, delays on the different fork branches are completely independent. The motivation of QDI circuits is that they are weakest compromise to pure delay-insensitivity needed to build practical circuits using simple gates and operators.

A speed-independent (SI) circuit is the circuit that operates correctly regardless of gate delays; wires are assumed to have zero or negligible delay. SI circuits were introduced by David Muller in the 1950's [13].

A self-timed circuit, described by Seitz [2], contains a group of self-timed "elements". Each element is contained in an "equipotential region", where wires have negligible or

well-bounded delay. An element itself maybe is an SI circuit, or a circuit whose correct operation relies on use of local timing assumptions. However, no timing assumptions are made on the communication between regions; that is, communication between regions is delay-insensitive.

2.2.4 Asynchronous circuits' advantages

An asynchronous circuit is one in which synchronization is performed without a global clock. Asynchronous circuits have several advantages over their synchronous counterparts, including [3]:

#Elimination of clock skew problems

As system become larger, increasing amounts of design effort is necessary to guarantee minimal skew in the arrival time of the clock signal at different parts of the chip. In an asynchronous circuit, skew in synchronization signals can be tolerated.

#Average case performance

In synchronous systems, the performance is dictated by worst-case conditions. The clock period must be set to be long enough to accommodate the slowest operation even though the average delay of the operation is often much shorter. In asynchronous circuits, the speed of the circuit is allowed to change dynamically, so the performance is governed by the average case delay.

#Lower system power requirements

Asynchronous circuits reduce synchronization power by no requiring additional clock drivers and buffers to limit clock skew. They also automatically power down unused components. Finally, asynchronous circuits do not waste power due to spurious transitions.

#reduced noise

In a synchronous design, all activity is locked into a very precise frequency. The result is nearly all the energy is concentrated in very narrow spectral bands at the clock frequency and

its harmonics. Therefore, there is substantial electrical noise at these frequencies. Activity in an asynchronous circuit is uncorrelated, resulting in a more distributed noise spectrum and a lower peak noise value.

#Component modularity and reuse

In an asynchronous system, components can be interfaced without the difficulties associated with synchronizing clocks in a synchronous system.

#Better technology migration potential

In asynchronous systems, migration of only the more critical system components can improve system performance on average, since performance is dependent on only the current active path. Also, since many asynchronous systems sense computation completion, components with different delays may often be substituted into a system without altering other elements or structures.

#adaptively to processing and environmental variations

The delay of a VLSI circuit can change with variations in fabrication, temperature and power-supply voltage. Synchronous designs have their clock rate set to allow correct operation under some allowed variations. Due to asynchronous circuits' adaptive nature, it operates correctly under all variations.

2.3 MICROPIPELINE

Micropipeline was introduced by Sutherland to build asynchronous pipelines [9]. A micropipeline has alternating computation stages separated by storage elements and control circuitry. Sutherland's approach uses transition-signaling for control along with bundled data and employs a "capture-pass" latch as a data storage element; and implementation is illustrated in Figure 6.

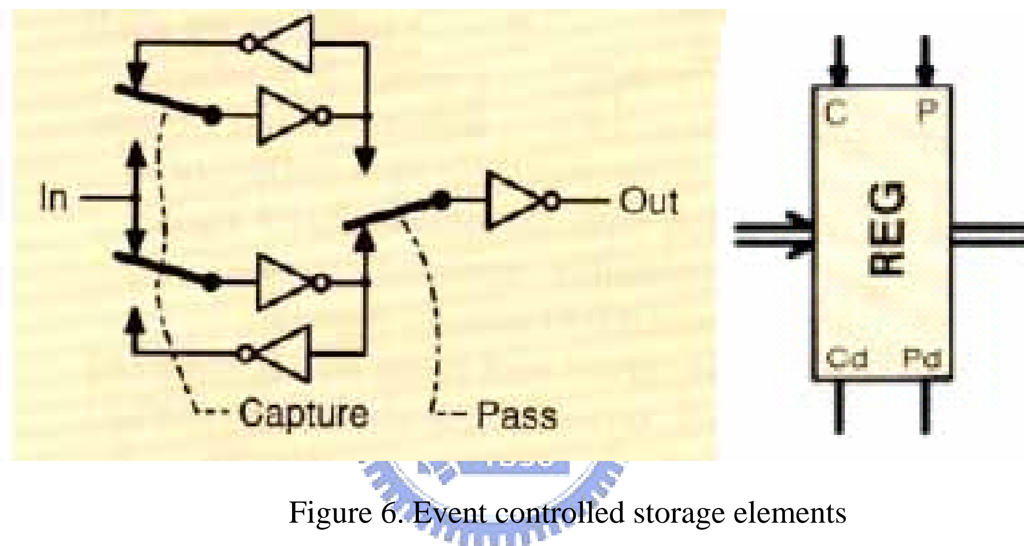


Figure 6. Event controlled storage elements

The data storage element uses two latches side by side and activates them alternately. The switches connecting ***IN*** to latches change over in response to an event on the Capture line while the switch connecting ***OUT*** to the latches changes over in response to pass events.

The capture – pass latch is transparent until an event occurs on its Capture line; this causes the latch to hold any data that was at its input line at that time. The Cd event signal report the data capture operation has completed, and the ***OUT*** represents the captured data. In this moment, any data change will have no effect on the ***OUT*** value. When the ***OUT*** value has been consumed, then a subsequent event on the pass control wire will return the latch to its transparent state, ready for the next data value and input event. The Pd event signals reports the completion of the pass operation.

Capture-pass latch structures can be interconnected to form micropipeline using Muller C-gates to ensure correct operation of the bundled data protocol; Figure 8 illustrates the operation of a micropipeline with 4 stages. A control stage of the pipeline consists of a Muller C-element. A C-element with two inputs and one output behaves as follows. If both inputs are 1, the output is 1; if both inputs are 0, the output is 0. Otherwise, if inputs have different values, the output holds its current value. Figure 7 details a Muller C-element and its properties. The C-elements in the micropipeline behave similarly, except that each has one inverted input.

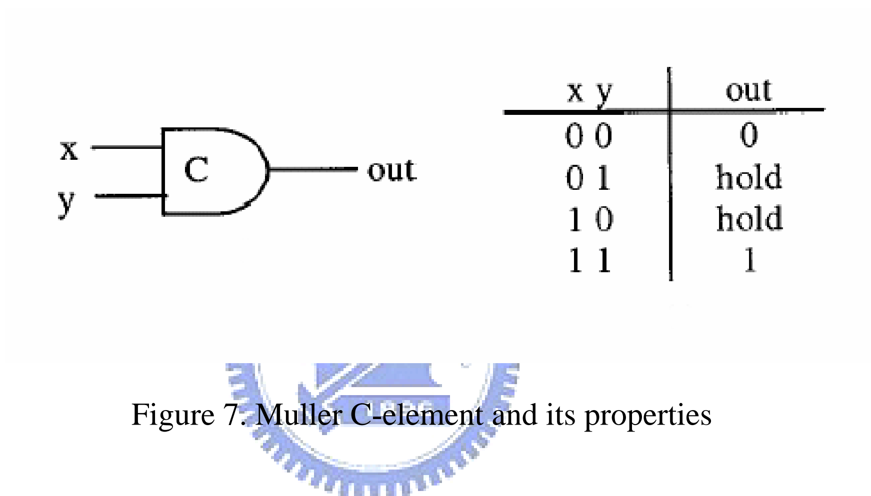


Figure 7. Muller C-element and its properties

Initially, all C-gate outputs are initialized to zero; the inversion is thus necessary to "prime" the C-gate for firing on the first event received on ***R(in)***. This will cause the first stage to capture the data presented at ***D(in)***. On completion of the capture, an acknowledgement is returned to the sender of ***D(in)*** as an event on ***A(in)***; the sender can then respond with new data on ***D(in)***. This acknowledgement is also forwarded to the next stage, via a delay, to the C-gate controlling the second capture-pass stage. The delay element ensures that data is valid at the second stage prior to the issue of a capture event.

The same behavior is repeated at the second stage: On completion of the first stage output value captured, an acknowledgement returned to the first stage and reset the latch to its transparent stage. Also, the acknowledgement is forwarded to the next stage. This process

continues through stages 3 and 4, and a final request appears at ***R(out)***.

Further new data may arrive ***D(in)***, can not process until the first stage capture-pass latch becomes transparent, the same situation on the second stage, and so on. ***A(out)*** acknowledgement event from the receiver must be responses that the ***D(out)*** has been consumed, thus enabling pipeline data to progress continuously.

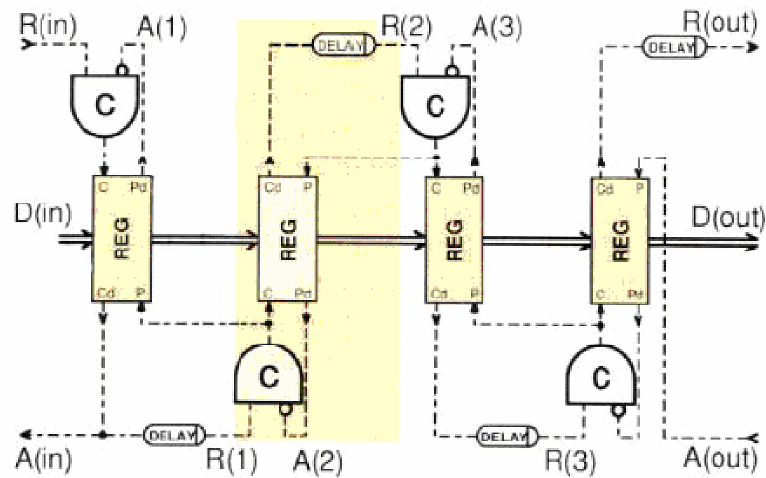
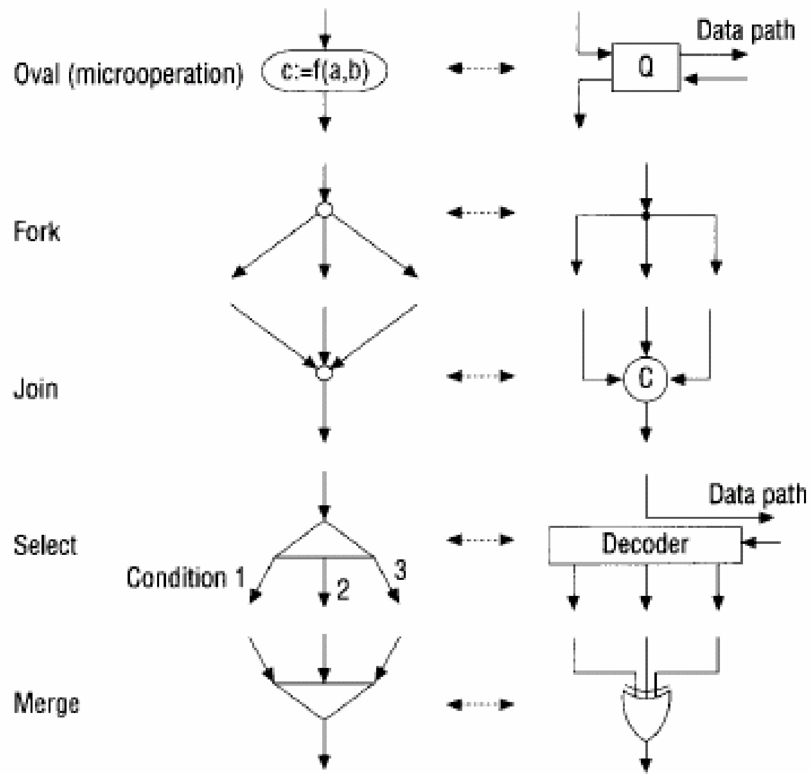


Figure 8. Micropipeline with 4 stages

1896

2.4 Data dependency graph

Given an instruction set and hardware resources such as registers and functional units, we can specify the behavior of each instruction by using DDG [24]. For high-speed operation, the microprocessor should execute as many micro operations as possible in parallel. Naturally, a dependency relation between two micro operations may require one micro operation to be completed before the other is initiated. DDG can be used to represent such dependency relations. DDG is constructed by five types of primitive elements: Oval, Fork, Join, Select and Merge. These elements are shown in Figure 9.



E S A
1896

Figure 9. Five basic elements of data dependency graph and its mapping circuit

Chapter 3 Design and implementation

3.1 The asynchronous AVR microcontroller architecture

According to the analyzing the AVR instruction set, we propose an organization for our asynchronous AVR microprocessor. There are four stages in our asynchronous AVR, instruction fetch (***IF***) stage, instruction decode (***ID***) stage, execution (***EX***) stage and write back (***WB***) stage. The dot lines represent the acknowledgement event, the coarse lines represent request event and the thin lines represent dual-rail data.

The whole controller process is governed by the control components (Q element) in the left side. The motion of the Q element is that, the request event is received in Q element, and then a complete 4-phase handshake procedure sends out to accomplish the data transmission. Figure 10 illustrates Q element and its procedure. The Q element details are in [15].

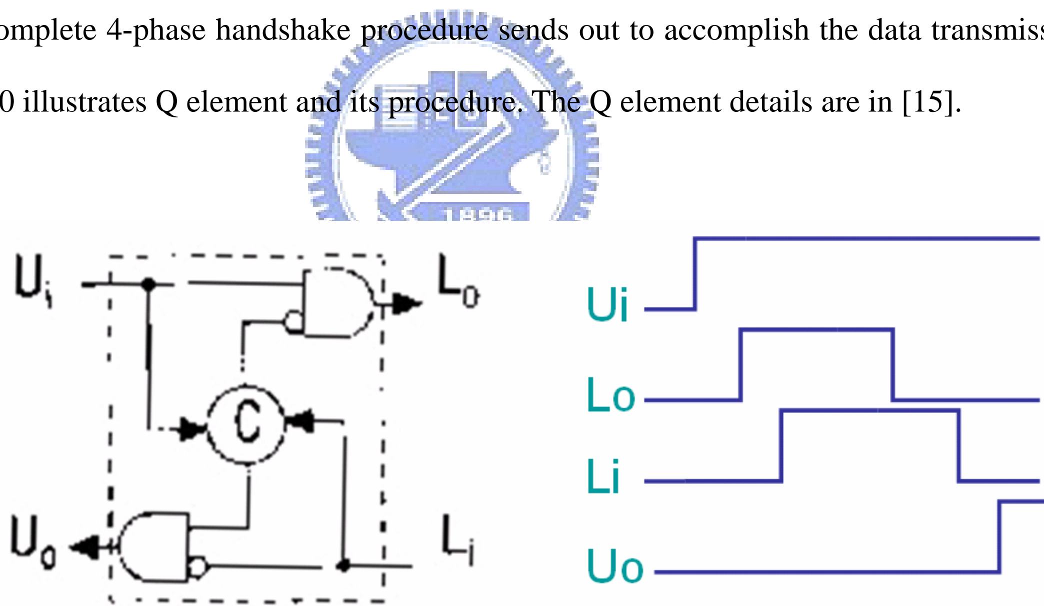


Figure 10. Q Element and its procedure

The complete asynchronous AVR microcontroller procedure was introduced below: Once the reset signal received in the AVR microcontroller, the internal register content was set to zero, and waiting the next request event signal. After receiving the request event, the first stage Instruction Fetch stage was activated to accomplish the three sequence procedure

below ：

Once the request event signal received in ***Q1*** control component, ***Q1*** transfers the action to the ***Q1_1*** control component, then ***Q1_1*** repeat the same action to transfer request event to ***PC*** register. The ***PC*** register acknowledgement the content value to be the instruction memory address signal. The instruction code is then decided from the outside instruction memory output dual rail format instruction. After the instruction register received the instruction code, it acknowledges ***Q1_1*** element to complete the first procedure in instruction fetch stage.
The second procedure is that the ***PC*** register contents transmit to the ***NPC*** register. After the ***NPC*** register received the ***PC*** data, it acknowledges ***Q1_2*** element to complete the second procedure in instruction fetch stage. The ***Q1_3*** issues the request event signal to let the ***NPC*** register and ***INC*** register contents transmit to the ***INC*** adder to accumulate the ***PC*** value. Then the new pc value transmits to the ***PC*** register to complete the third procedure in instruction fetch stage.

The procedures in instruction decode stage are described below. Once the request event signal received in ***Q2*** control component, ***Q2*** sends the request event signal to instruction register to show the instruction codes. After the decoder receiving the instruction code, it sends the control lines to the corresponding peripheral resources, such as register files to fetch the register contents. And it sends some related decode information, such as opcode sub-index and destination register index to next stage. After the next stage received enough data, it acknowledges ***Q2*** element to complete the procedure in instruction decode stage.

The procedures in execution stage are described below. Once the request event signal received in ***Q3*** control component, ***Q3*** sends the request event signal to execution stage register to show the corresponding data. After the ***ALU*** receiving the demand data, it starts to figure out the result and pass it to the write back stage register. After the next stage received the enough data, it acknowledges ***Q3*** element to complete the procedure in execution stage.

The procedures in write back stage are described below. Once the request event signal

received in ***Q4*** control component, ***Q4*** sends the request event signal to write back stage register to show the corresponding data. After the register file received the destination register index and its contents, it starts to refresh the register contents. Once the fresh procedure finished, the register file acknowledges ***Q4*** element to complete the procedure in write back stage. The ***Q4*** component then acknowledges initial request event signal to complete the one instruction procedure. Figure 11 illustrates the asynchronous AVR microcontroller.

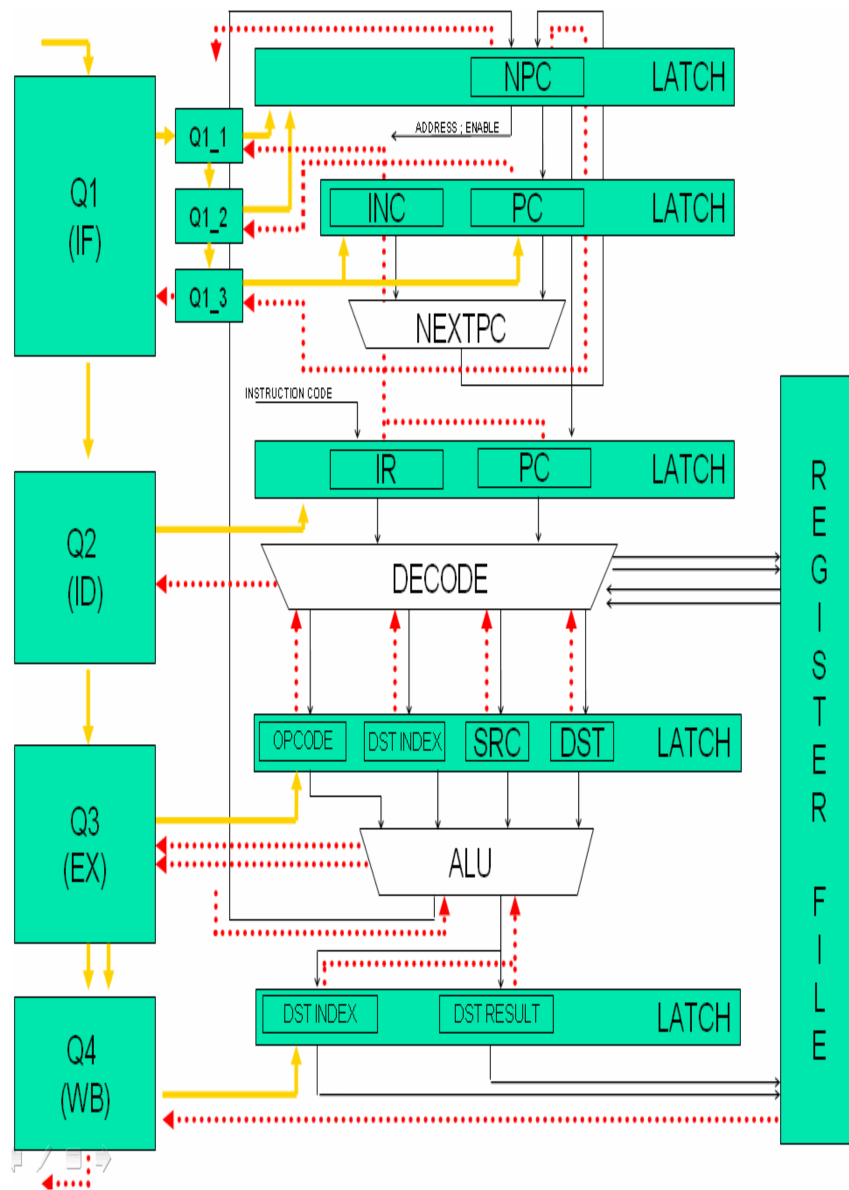


Figure 11 the organization of asynchronous AVR microprocessor

3.1.1 Our asynchronous Micropipeline circuits

The original Micropipeline structure comprises the two side-by-side register with two phase bundled data convention. In order to ensure the correctness in FPGA environment implementation, some modification is needed. The modified Micropipeline structure comprises one register with four phases, dual rail data convention. Figure 12 illustrates dual rail, 4 phases, delay insensitive Micropipeline with 4 stages.

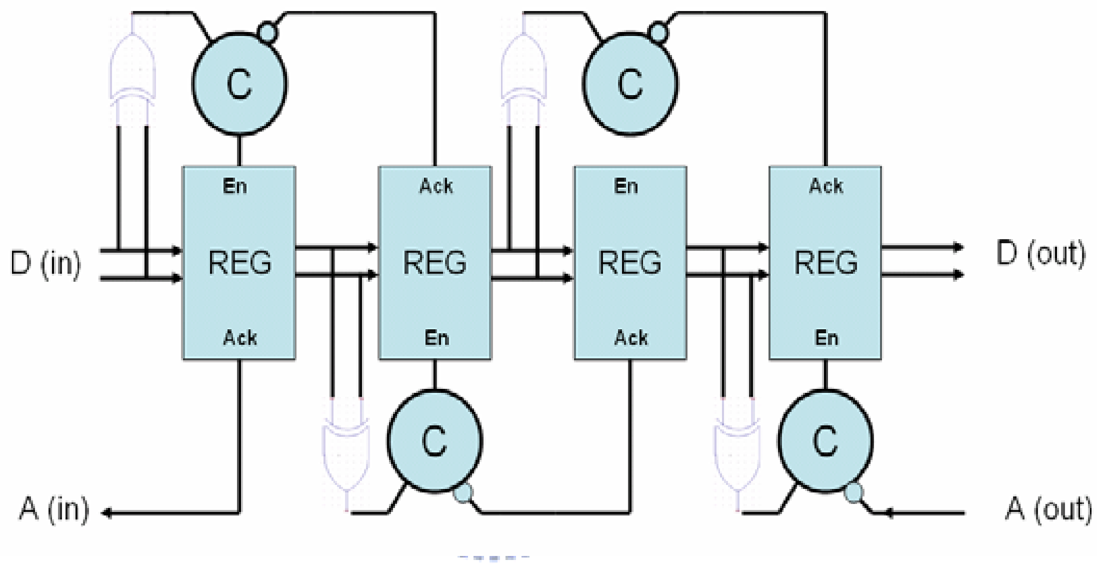


Figure 12. Dual-rail, 4 phase, delay insensitive Micropipeline with 4 stages

The differences between original Micropipeline and our micropipeline are described below：

We use dual rail data protocol to replace single-rail bundled data protocol. In order to detect to data arrived, we use a *xor* gate to sense the data variation. The most advantage is that we do not need to consider the synthesis and place & route problem arising in FPGA environment. The delay insensitive structure is insensitive to this constraints in which ensure the implementation accuracy.

3.1.2 Instruction fetch stage (IF)

There are three registers, one adder and three procedure control components in instruction fetch stage. ***PC***, ***NPC*** and ***INC*** comprise the three registers. Program Counter (***PC***) stores the current program memory address value; ***NPC*** stores the next program memory address value. The ***NPC*** and ***PC*** value are set to dual rail zero after the initial reset signal arrived. The ***NPC*** register data width is eight bits and the same as the ***PC*** register. The ***INC*** registers stores the program counter increment, its value is one. The ***NEXTPC*** is an adder using calculate the next program memory address. The ***NEXTPC*** result is stored in ***NPC*** register.

The three procedure control components ***Q1_1***, ***Q1_2*** and ***Q1_3*** govern the instruction fetch stage procedure. These control components ensure the correct flow; the first procedure is sending the memory address to instruction memory to fetch the instruction code; the second and the third procedure is refreshing the ***NPC*** value.

The ***NPC*** register provides two input port; one port is used for receiving the ***NEXTPC*** result and the other port is used for receiving the branch instruction result from ***ALU***. After the ***ALU*** figures out the new branch address, ***NPC*** register could be update immediately to improve the performance. Figure 13 illustrates the organization of instruction fetch Stage.

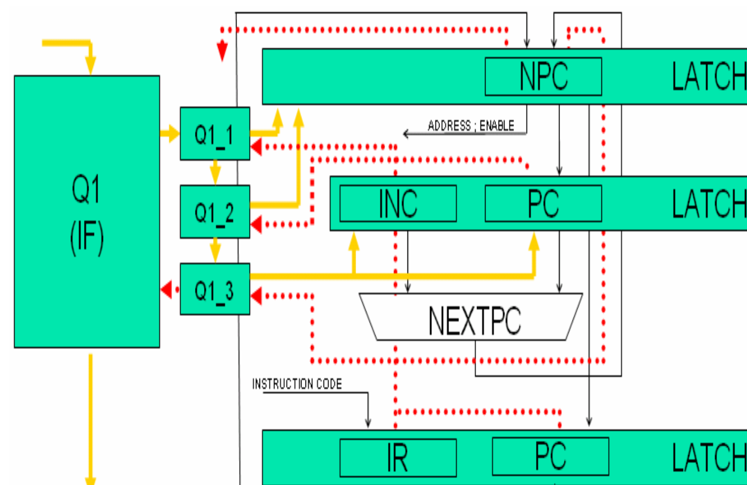


Figure 13. The organizations of instruction fetch Stage

3.1.3 Instruction decode stage (ID)

There are two registers and one decoder in instruction-decode stage. Figure 14 illustrates the organization of instruction decode stage.

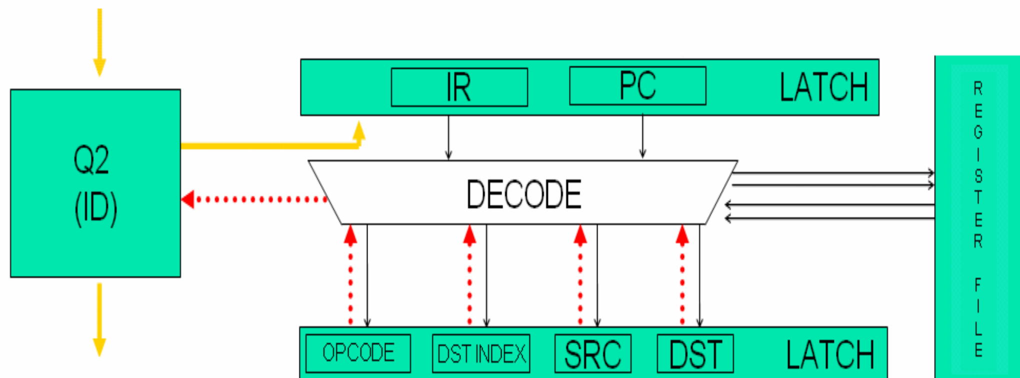


Figure 14 The organization of instruction decode stage

The widths of the instruction register are 16 bits as wide as the AVR instruction set format. According the instruction contents, the decoder sends the necessary control lines to corresponding peripheral resources to accomplish the procedure. For example, the instruction *ADD R1, R3* means adds the R1 register value and the R3 register value, stores the result back to the R1 register. When the decoder received the instruction code, it sends request signal to the register files. The register files responses the R1 register value and R3 register value. The next latch acknowledges to ***Q2*** to accomplish the instruction procedure in this stage. The decoder also sends the destination register index and the micro-instruction index to the next latch to serve the corresponding action.

3.1.4 Execution stage (EX)

There are four registers and one ***ALU*** in execution stage. Figure 15 illustrates the organization of execution Stage.

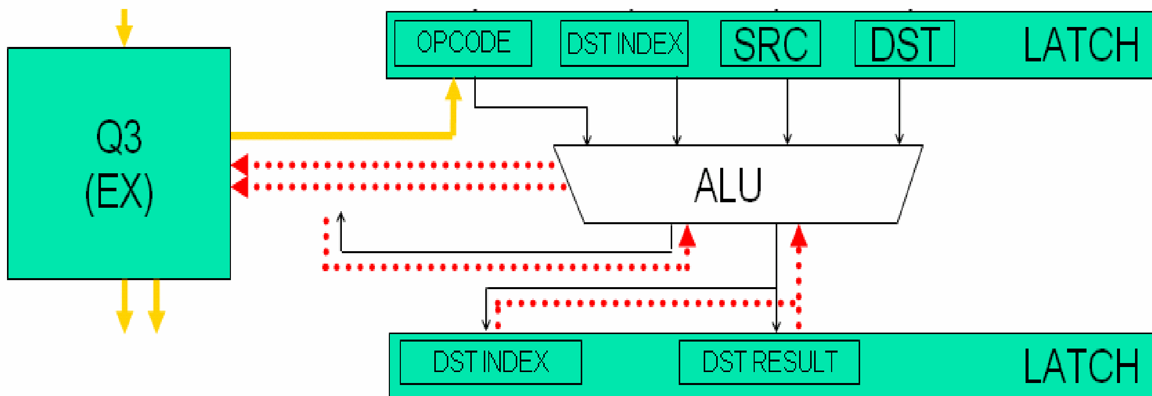


Figure 15. The organization of execution stage

The ***Q3 (EX)*** send the request signal to execution stage latch and show the opcode (micro-instruction index), source register value and destination register value to ***ALU***. It also passes the destination register index to the next latch. The next latch acknowledges to ***Q3*** to accomplish the instruction procedure in this stage.

In order to simplify the basic design the implementation of the arithmetic logic unit provides only the integer functions of addition and subtraction. The configuration of the ***ALU*** is based on an implementation described in [5].

The ***ALU*** provides a branch instruction output port. The new branch address result was send from this port to ***NPC*** register. When ***Q3 (EX)*** receives the acknowledgement from ***NPC***, it will influence the next control component ***Q4*** procedure and bypass the write back stage procedure. It means that write the result back to the register files is no necessary. It could improve the performance.

3.1.5 Write back stage (WB)

There are two registers in write back stage. Figure 16 illustrates the organization of execution stage. The two register contents are destination register index and destination register value. The ***Q4 (WB)*** send the request signal to write back stage latch and show the destination register index and destination register value to register file. The register file

acknowledges to ***Q4*** to accomplish the instruction procedure in this stage.

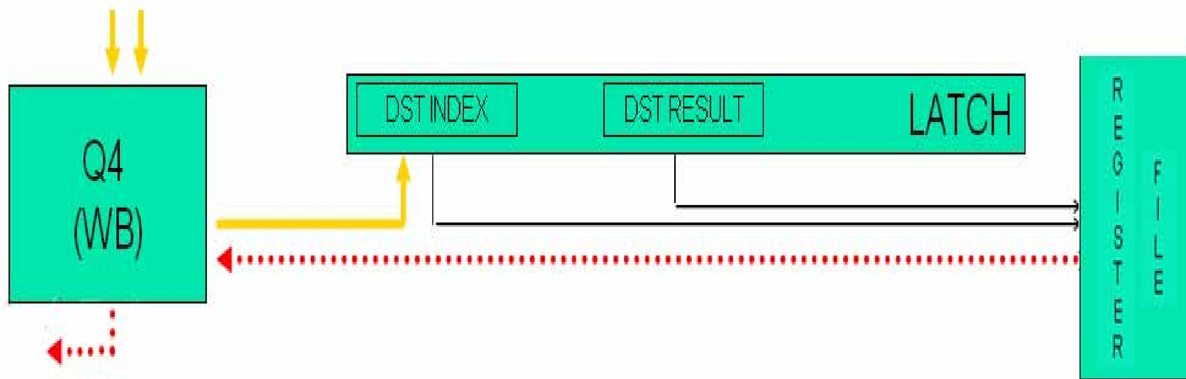


Figure 16 The organization of write back Stage

3.1.6 The register file

The register files comprises 32, 8-bit wide general purpose registers. It provides two output ports for read and one write port for write. The hardware organization helps the data manipulation between registers more efficiency. In order to observe the register contents, we connect the register R31 data output port to outward. This function will be demonstrated below.

3.2 Design methodology

In this section, we introduce the design procedure and its development tools. Section 3.2.1 introduces the DDG of instruction – a method to describe the micro operation of AVR instruction. Section 3.2.2 introduce the design methodology and explain how to combine the synchronous CAD tool to explore asynchronous circuits.

3.2.1 Data dependency graph of AVR microcontroller

After defining the asynchronous AVR hardware organization, we implement the AVR instruction set to the organization. The DDG is used to describe the AVR instruction set micro operation. Every part of DDG is mapped to the corresponding hardware [8].

The instruction *ADD Rd, Rs* is used to explain how DDG describe the AVR instruction set. The meaning of the instruction *ADD Rd, Rs* is that add the value of register Rd and the value of register Rs, and stores the result in the destination register Rd. Its operation and instruction format is shown in Figure 17.

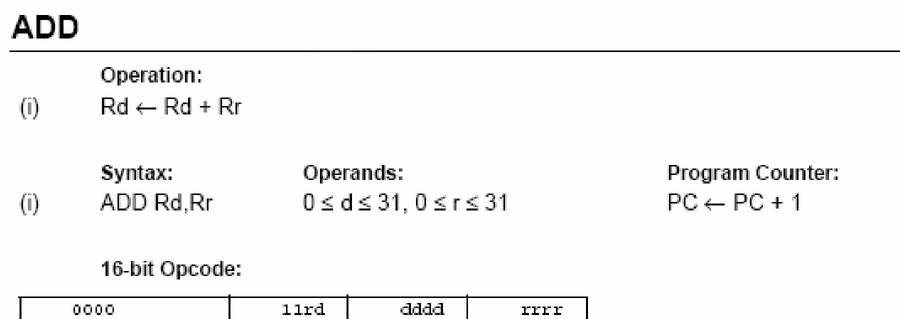


Figure17. The instruction content of *ADD* instruction

According the organization of AVR microcontroller, the DDG of *ADD Rd, Rs* instruction is shown in Figure 18. Because the first stages of all AVR instructions are the same, we only

show the DDG of *ADD Rd, Rs* instruction from instruction decode stage to write back stage.

After the decoder receiving the instruction code, it provides the Rd and Rs register contents to the ***ID_EX*** latch. The execution stage adds these two values and stores it in ***EX_WB*** latch. The write back stage sends the Rd data back to the register file. The whole procedure is the same as we described in section 3.1.

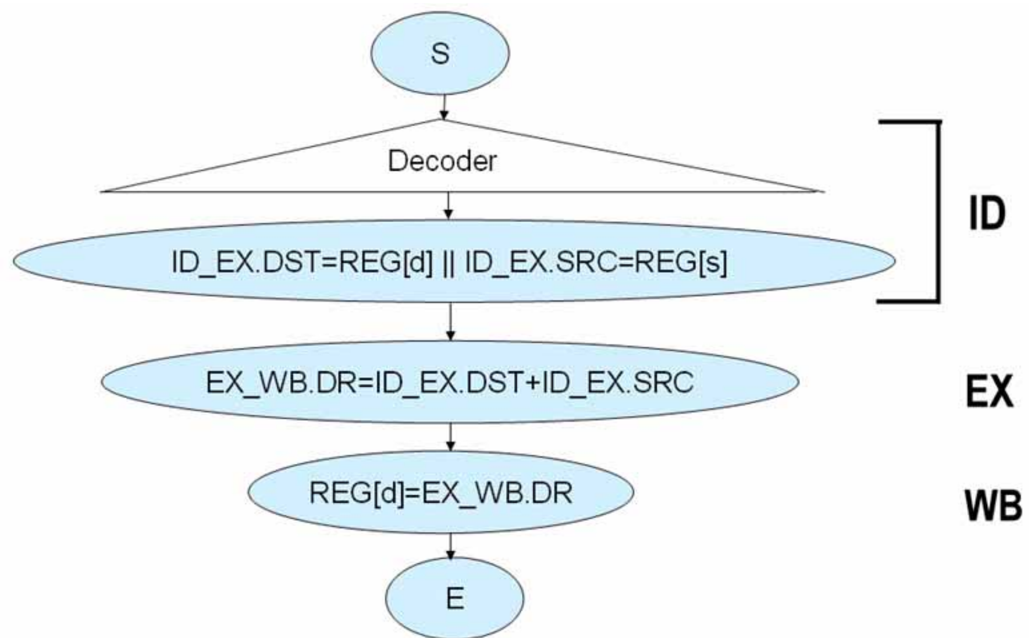


Figure18. The DDG of *ADD* instruction

3.2.2 Design steps

Figure 19 illustrates the asynchronous design steps and how to integrate the existing tools in our design. We describe the design steps below:

Referencing to [6], we created enough delay-insensitive FPGA elements with Verilog language and verify its function with Modelsim software. According to the specification, AVR organization and its DDG of AVR instruction set, we wrote the AVR organization HDL with Verilog language. The Modelsim software was used to verify the correctness of AVR in HDL. Finally we download our design to the physical chip to validate the AVR functions.

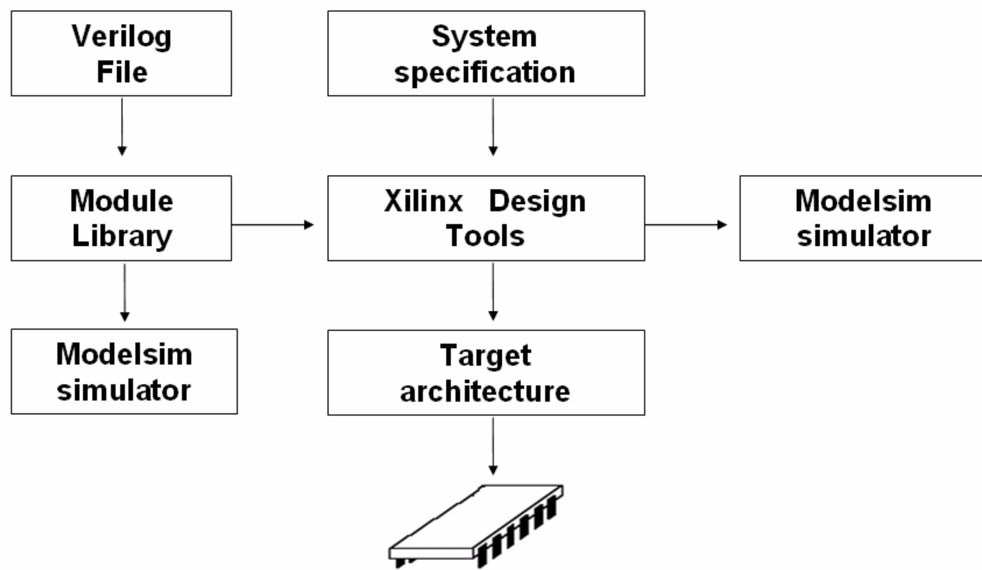


Figure19. The design and implementation procedure

1896

3.2.3 Design environment

Table1 illustrates the existing tools using in our design and its purpose.

Synchronous Tools Name	Purpose
Xilinx ise 4.2.03i	Project Management Download design into Target Board
Synplify Pro 7.6	Logic synthesis
Modelsim SE/EE PLUS 5.4	Simulation and verification

Table 1. The design environment tools

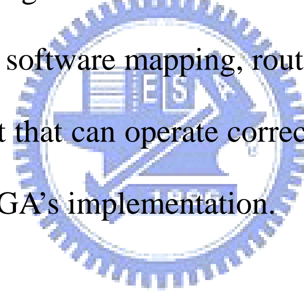
3.3 Implementation

In this section, we introduce implementation environment. Section 3.3.1 introduces the issue for using FPGA to implement asynchronous circuit. Section 3.3.2 introduces the target board and selected FPGA Chip.

3.3.1 FPGA design issues for asynchronous logic

It is no doubt that FPGA is an extremely effective means of performing fast development and test of digital circuits. The employment of large amounts of simple logic gates and datapaths that can be rapidly programmed and reprogrammed until a desired solution has been found is a very cost effective method of hardware design.

Asynchronous circuits design is sensitive to the hazard that was introduced from the FPGA working phases, such as software mapping, routings and placements. The characteristic of delay-insensitive (DI) circuit that can operate correctly regardless of the delays on its gates and wires is suitable for the FPGA's implementation.



3.3.2 Implementation environment

Xilinx prototype board

The Xilinx prototype board (model: AFX BG560-100) is used as the target board. It helps to implement our asynchronous AVR microcontroller design without other additional efforts. The traditional approach to experimenting with new devices involving wiring together some ICs on a circuit board is becoming impractical and ineffective. Instead, with new high-density devices on custom PC boards represents a substantial investment of time and money. This prototype boards from device manufacturers can meet this requirement for experimentation. More details about Xilinx prototype board can be found in [17]; Figure20

shows the photo of physical xilinx prototype board.

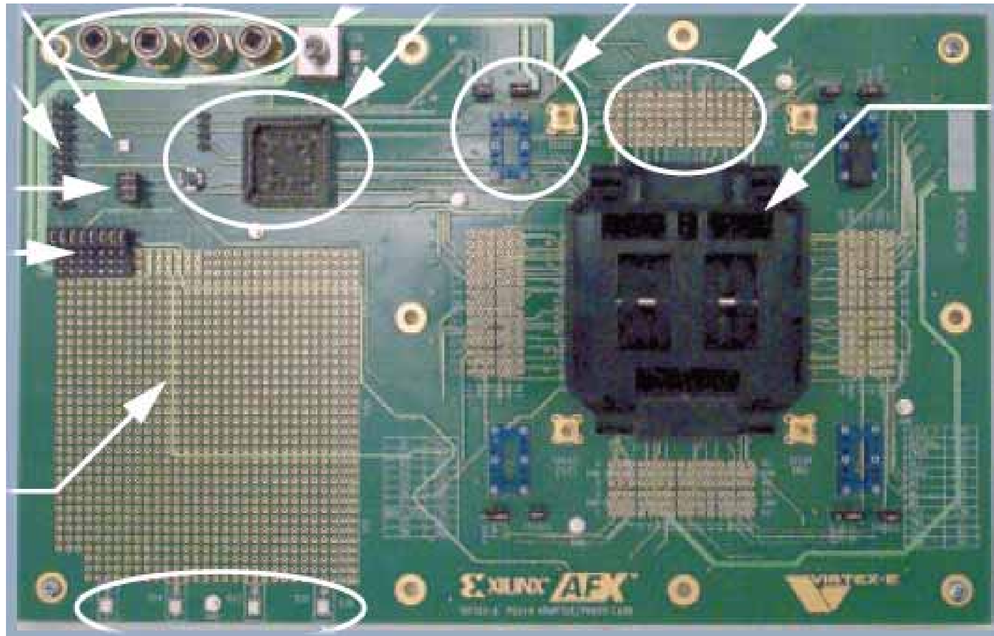
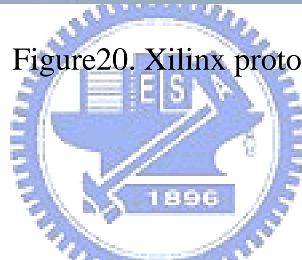


Figure20. Xilinx prototype board



FPGA chip

The target FPGA chip is XCV 1600E BG560 Virtex™-E 1.8 V; The main feature of Virtex E chip are fast, high density system gate count and low power consumption. More details about Virtex FPGA Chip can be found in [18].

Chapter 4 Testing

In order to test the implementation of AVR, a simple program was designed that could be used to exercise the paths. The general purpose of this program was to load data from an external memory, manipulate that data through ***ALU*** and then change the register file contents. Resultant values could be observed by the register file. Table2 shows a typical sequence of execution of instructions.

Instruction	Fetch	Decode	Execution	Write Back
1	Load1	-	-	-
2	-	Load1	-	-
3	-	-	Load1	
4	-	-	-	Load1
5	Load2	-	-	-
6	-	Load2	-	-
7	-	-	Load2	-
8	-	-	-	Load2
9	Alu1	-	-	-
10	-	Alu1	-	-
11	-	-	Alu1	-
12	-	-	-	Alu1
13	Jump1	-	-	-
14	-	Jump1	-	-
15	-	-	Jump1	-
16	-	-	-	Jump1

Table 2.Sequence of instruction execution

There are instances in the execution of a number of instructions in AVR that requires the concurrent access of the register file or other such logic. As AVR has not been designed to function in a concurrent manner, e.g. only sequential processing of instructions can occur in any stage; even with the modified micropipeline structure was introduced.

The obvious conclusions from above statements are that there are likely to be a large performance compromise but rather as a prototype device to investigate how existing tools could cope with such a task.

The AVR test program along with a number of other numeric constants, e.g. values representing the data that would be actually found in the external memory, has been connected to the AVR. The general configuration of this external logic with the associated connectivity can be seen in Figure 21, and introduced below.

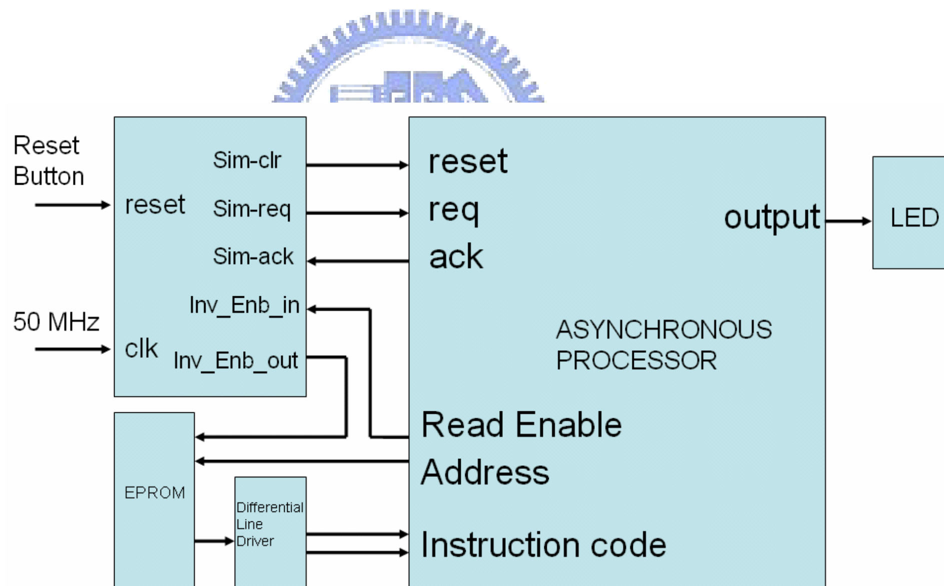


Figure21. Testing configuration

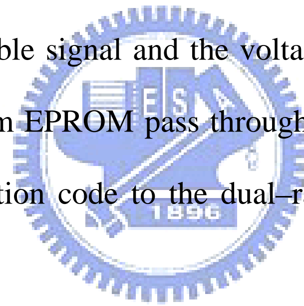
4.1 Testing configuration introduction

In Figure 21, the right block represents the asynchronous AVR. In order to observe the register contents, we extend the general-purpose register R31 data to outside, named OUTPUT, as you can see in the right part of the asynchronous AVR. We connect a LED to its

OUTPUT port to indicate the register content. The left upper block is used for controlling the testing procedure and it receives a 50 MHz signal to internal timing. When this control block detects the reset signal (active low), the minimum 20ms reset signal will send into the asynchronous chip to ensure the internal register situation from the sim_clr port.

Sim_req request event signal arise later to start the first instruction operation. The AVR feedback the acknowledgement events to control block after complete inner operation. The low Sim_req request event signal response the acknowledgement from AVR. The request and acknowledgement situation are idled when transaction is fully completed. The next instruction repeat the procedure described above around.

The instruction fetch stage of AVR will send out an EPROM Enable signal and address signal. Due to the characteristic of the EPROM 2764 active low enable signal is different from the AVR active high enable signal and the voltage level transformation is needed. The instruction code generated from EPROM pass through the one rail to two rail circuits where transfer the single rail Instruction code to the dual–rail instruction code and finally, to the AVR microcontroller.



4.2 16 bits instruction composition

The type of EPROM unit employed was an 8K MD2764 device with 8-bit addressing. In order to present the 16-bits instruction or data value to an AVR instruction input, two of these devices would be enabled in parallel in order to construct that 16-bit word. The structural organization of these EPROM units is shown in Figure 22.

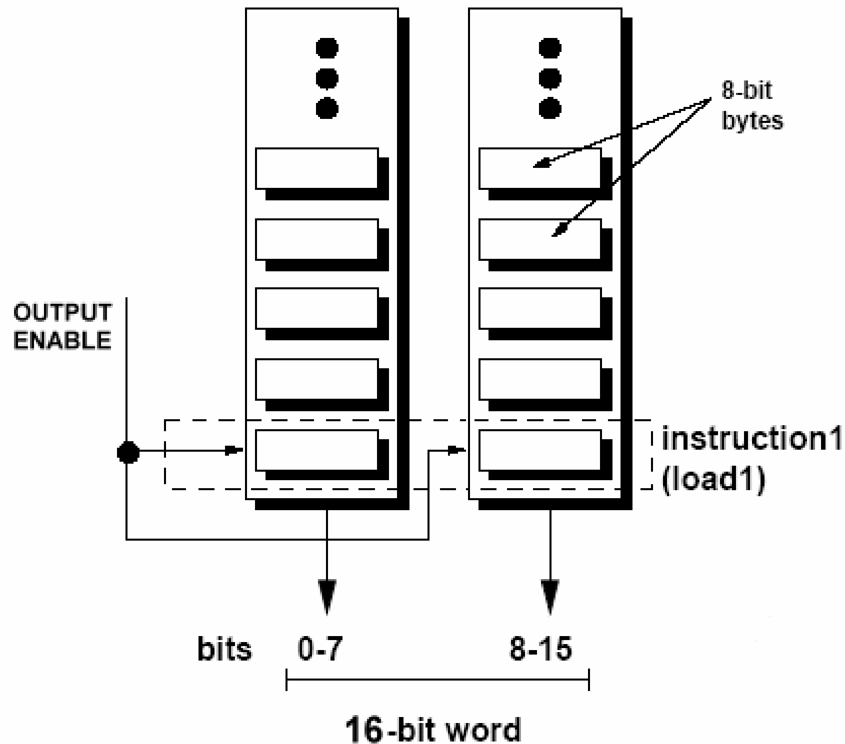


Figure22. EPROM 16-bit word configuration

4.3 Memory interface

In order to connect to synchronous components, some modification must be inserted to meet synchronous requirement to assure the correct function, and introduced below.

4.3.1 Dual-rail to single-rail circuit

The internal register in our AVR is designed for dual rail format. Every data-bit in dual rail format is represented by two one-bit latches. The address data provided from the NPC register is positive part of dual rail data. Furthermore, the EPROM strobe signal is provided by the combination of dual rail address data. The combination organization is constructed by the Muller-C circuits. The strobe signal is guaranteed to be late to the address data which the EPROM output the correct instruction code. Figure23 illustrates that the two-rail data constructs the strobe signal with Muller-C circuits and Y signal is directed from the positive

part of the dual rail X data. More details can be found in [13].

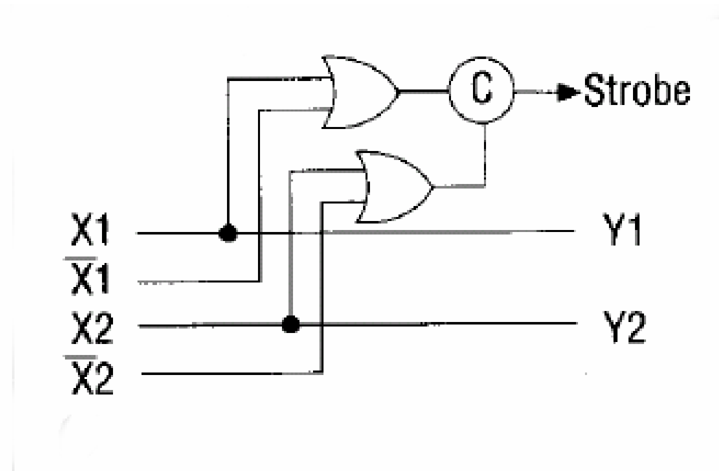


Figure23. Two Rails to One Rail Circuit

4.3.2 Single-rail to dual-rail circuit

The instruction register is dual-rail format. The instruction code generated from EPROM must pass through the one-rail to two-rail circuits. The differential line driver IC (AM26LS31) is used to the transformation circuits. Figure24 illustrates that the two bit single-rail data is transferred to two dual-rail data. The strobe signal is used to control the dual-rail format data. The Z data is valid with the strobe signal is high. The Z condition is high impedance with the strobe signal is low. More details can be found in [13].

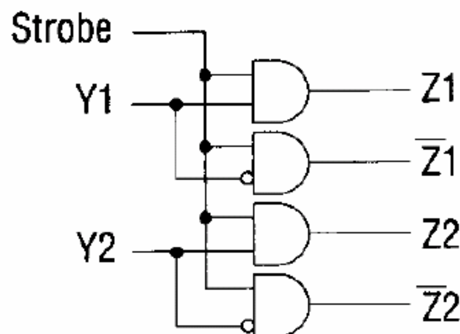


Figure24. One-rail to two-rail circuit

4.3.3 Return to zero circuits

The ultimate condition of four-phase handshake protocol is that request and acknowledgement signals are set to the idle (L) condition. To achieve the rule standard, the extra circuits are needed. We connect the return to zero circuits to the output of the single-rail to dual-rail circuits output. The handshake between instruction fetch stage and EPROM is ended with the strobe signal is low. The single-rail to dual-rail circuits output condition is high impedance with the strobe signal is low. The dual-rail data is low with the return to zero circuits because of the electronic rules. It meets the four-phase return to zero handshake protocol. Figure 25 illustrates the return to zero circuits.

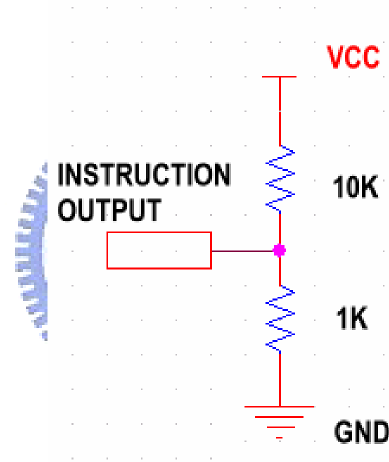


Figure25. Return to zero circuits

4.4 HDL verification

The Modelsim software is used to simulate the system level behaviors of AVR and verify the AVR instruction set function. Figure26 illustrates a looping addition function.

The clr signal is send from outside to reset the internal condition in AVR. The req_AIC is send from outside to start an instruction procedure. After AVR finished one instruction, it sends acknowledgement ack_AIC signal outside. The EPROM_AD and EPROM_ENABLE are issued from instruction fetch stage to outside EPROM. Its feedback instruction data in

dual-rail format are INSTR_Id0 and INSTR_Id1.The REG_31_OUT is the register 31 contents.

After receiving the reset signal, the contents of the register R31 are cleared to zero. The contents are updated the first instruction executed. And it decrease one after the *ADD* instruction (INSTR_id1 =0ffe) executed. The REG_31_OUT is complement format to match up the outside LED circuits.

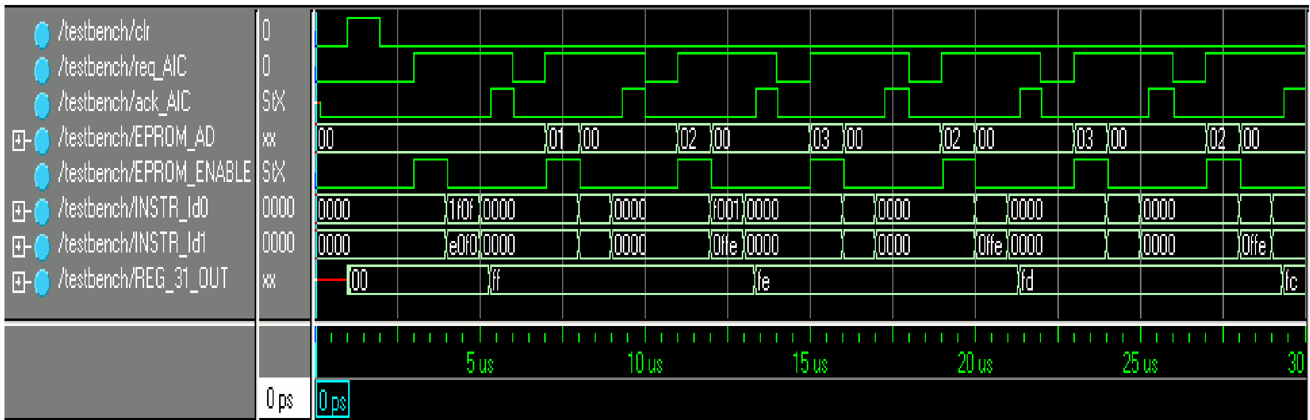


Figure26. Simulation result in Modelsim SE/EE PLUS 5.4

4.5 Physical circuits' validation

The simulation software is used to verify the hardware behaviors. The verified design is downloading to the physical circuits to check the function. Table3 show the test program.

MEMORY ADDRESS	ASSEMBLY CODE	Description
000000	LDI R31 , 0	R31 = 0
000001	LDI R30 , 1	R30 = 1
000010	ADD R31 , R30	R31 = R31 + R30
000011	JMP 00000010	Branch to 00000010

Table 3. The test program of looping addition

We can validate the physical circuits with our downloading design by observing the LED status. The correct function of the physical circuits has the following meanings. First, the software simulation is reliable with the delay insensitive model. Second, peripheral device with the protocol transformation circuits are also working correctly. Finally, it proves correctness of the design methodology we described. Figure27, Figure28 illustrates the physical circuits. The Xilinx prototype board in Figure 27 comprises Virtex E FPGA Chip which used to implement the AVR microcontroller core and control procedure circuits. The I/O Card in Figure 28 comprises the EPROM, Memory interface, Output LED and Remote RESET button.

Figure27. Xilinx prototype board with Virtex E FPGA Chip

Figure28. I/O Card

Chapter 5 Conclusion

In this thesis, we propose an asynchronous AVR microcontroller and implement it using FPGA chip with asynchronous circuit models. The AVR microcontroller is the Reduced Instruction Set Computer (RISC) architecture and the core is a standard 8-bit microcontroller, widely used in Atmel products.

We describe the behavior of AVR instruction by using data dependency graph. In addition, we also establish several asynchronous FPGA cell libraries in Verilog. Following the verification in simulation software and validation in physical circuits, these cells also proved to work correctly in FPGA environment. With the delay insensitive model, we do not worry about the synthesis and place & routing variation. The design can be downloaded to any FPGA chip arbitrarily without altering it.

We did not implement all 90 instructions in AVR specification. The implemented instructions are listed in table 4.

Owing to lack of asynchronous design tools, we use the existing synchronous tools, such as Modelsim and Xilinx ISE4.2i, to accomplish the design and simulate it. The successful combination with existing synchronous tools is demonstrated. But the development period is long and slow and complication increasing with the scale. In order to shorten the development time, the asynchronous design tools are needed.

We modify Sutherland's Micropipeline to dual-rail and delay insensitive Micropipeline and it could be implemented in FPGA chip without losing the asynchronous characteristics.

Once the design methodology established, we can use the same way to survey and implement other asynchronous CPU cores. The asynchronous AVR is not intended to be a fully custom designed microcontroller such as the AMULET processors, but it is as a

prototype device to investigate how existing tools could cope with such a task. In the future, a fully custom designed microcontroller will be realized in such way.

Instruction type	Instruction counts	Instruction lists
Arithmetic and Logic instructions	20	ADD ,ADC,SUB,SUBI,SBC ,SBCI,AND, ANDI,OR ,ORI ,EOR ,COM,NEG,SBR, CBR ,INC ,DEC ,TST ,CLR ,SER
Branch Instructions	25	RJMP,JMP,CP,CPC,CPI,BRBS,BRBC, BREQ,BRNE,BRCS,BRCC,BRSH,BRLO, BRMI,BRPL,BRGE,BRLT,BRHS,BRHC, BRTS,BRTC,BRVS,BRVC,BRIE,BRID
Data Transfer instructions	2	MOV,LDI
Bit and Bit-test instructions(17/31)	17	SEC,CLC,SEN,CLN,SEZ,CLZ,SEI,CLI, SES,CLS,SEV,CLV,SET,CLT,SHE,CLH, NOP

Table 4. Implemented AVR instructions

Reference:

[1]Al Davis and Steven M. Nowick, An introduction to asynchronous circuit design, technical report, 1997.

[2]C. Mead and L. Conway, Introduction to VLSI systems, chapter 7, 1980.

[3]Chris J. Myers, Asynchronous circuit design, Wiley Interscience Publication

[4]David L. Dill, Trace theory for automatic hierarchical verifications of speed independent circuits, ACM Distinguished Dissertations, MIT Press, 1989.

[5]F. C. Cheng, S. H. Unger, M. Theobald and W.-C.Cho, Delay-insensitive carry look-ahead adders, In Proc. International Conference on VLSI Design, pages 322-328, 1997

[6]F. C. Cheng, Asynchronous Systems & System-on-a-Chip Design Lecture, http://www.cse.ttu.edu.tw/~cheng/courses/soc.htm

[7]Gaute Myklebust, Embedded systems and uC PowerPoint www.atmel.com

[8]Han-Chun Lin, Design of an Asynchronous Thumb Microprocessor, Master Thesis, Department of Computer Science and Engineering, Tatung University, July, 2002.

[9]I. E. Sutherland, Micropipeline, Communications of the ACM, 32(6):720-738, June, 1989.

[10]Jan T. Udding, A formal model for defining and classifying delay insensitive circuits, Distributed Computing, pp. 197-204, 1986.

[11]Jo C. Ebergen and Parallelaham Birtwistle, Higher Order Workshop, pp. 85-104, SpringerVerlag, 1991.

[12]M. B. Josephs and J. T. Udding, An overview of DI algebra, In Proc. Hawaii international conf, system sciences, volume I. IEEE computer society press, Jan 1993.

[13]R. E. Miller, Combinational Circuits, volume 1 of Switching Theory, 1965.

[14]Steven M. Ban, Introduction to performance analysis and optimization of asynchronous circuits, PhD thesis, California Institute of Technology, 1991.

[15]Takashi Nanya, Yoichiro ueno, Hiroto Kagotani, Masashi Kuwako and Akihiro Takamura, TITAC: Design of a quasi delay insensitive microprocessor, IEEE Design & Test of Computers, 11(2):50-63, 1994.

[16]Wesley A. Clark, Macro modular computer systems In AFIPS Conference, Volume 30, pages 335-336, Spr. 1967.

[17]Xilinx Prototype Platforms User Guide for Virtex and Virtex-E Series FPGAs, www.xilinx.com

[18]Virtex™-E 1.8 V Field Programmable Gate Arrays Production Product Specification, www.xilinx.com