DESIGN OF A LOW-POWER DUAL-RAIL ASYNCHRONOUS 8051 MICROCONTROLLER

XUE CHAO
(B.Eng.(Hons.), NUS)

NATIONAL UNIVERSITY OF SINGAPORE

2010

DESIGN OF A LOW-POWER DUAL-RAIL ASYNCHRONOUS 8051 MICROCONTROLLER

XUE CHAO
(B.Eng.(Hons.), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF ENGINEERING BY RESEARCH

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2010

Acknowledgement

First of all, I would like to thank my supervisor, A/P. Lian Yong, for his continuous support and guidance during my Master candidature. A/P. Lian Yong's insightful advice has helped me greatly throughout this work.

I also would like to thank all my colleagues and friends in the VLSI & Signal Processing Laboratory for helping me in one way or another in my research work. They are Liew Wensin, Eng Weijie, Chua Dingjuan, Pan Junle, Zhang Qi, Niu Tianfang, Li Ti, Ashton Wong, Xu Xiaoyuan, Deepu John, Cheng Xiang and many others. Special thanks also go to the Laboratory officers Ms Zheng Huanqun and Mr Teo Seow Miang for their kind help during my stay in the laboratory.

I would like to dedicate this thesis to my beloved father Xue Fasheng and mother Zhang Fazhen. Their continuous support and encouragement are always the source of confidence that enables me to overcome various obstacles in life.

Contents

Summary

In this thesis, the design of a low-power voltage-scalable asynchronous 8051 microcontroller is presented. It is targeted to function as a general purpose processing unit in a biomedical sensor interface system with a possibly varying supply voltage in the range of 1V to 3.3V. At the top level, the proposed asynchronous 8051 microcontroller can be divided into four major parts: the asynchronous 8051 core, the synchronous peripherals, the custom-designed asynchronous SRAM and the interface wrapper blocks. The design flow starts from coding the asynchronous 8051 core using a dedicated hardware description language in the Balsa framework. After passing behavioral verification, the HDL code is synthesized into a Verilog gate-level netlist by the Balsa framework. This Verilog gate-level netlist of the asynchronous core is then read in by the Synopsys Design Compiler together with other Verilog modules that describe the interface blocks and synchronous peripherals. The Design Compiler optimizes and compiles the individual Verilog files into a unified Verilog gate-level netlist,

which is then imported into the Cadence SOC Encounter together with the LEF (library exchange format) file of the custom-designed asynchronous SRAM block for automatic P&R (placement and routing). A GDS (graphical database system) file of the asynchronous 8051 microcontroller is exported from the SOC Encounter into the Cadence Virtuoso framework. After performing LPE (layout parasitic extraction), the SPICE netlist is passed to Nanosim for final post-layout simulation verification.

The asynchronous 8051core is synthesized using an asynchronous EDA tool called "Balsa" and it adopts the four-phase dual-rail protocol. Four different versions of the asynchronous core are developed during the master candidature. According to the Nanosim post-layout transistor-level simulation data, the last version (referred as Design 4 in this thesis) consumes about 166pJ per each instruction while running at around 0.42MIPS at 1.0V supply for AMS 0.35μm technology. It is able to function properly from the nominal supply voltage of 3.3V down to 1V and below.

The proposed novel interface block to external commercial memory can work with any commercial memory in general after proper configuration of the preset overflow value and driving clock frequency of an internal counter module inside the interface block.

List of Tables

List of Figures

List of Abbreviations

ACC	Accumulator
ALU	Arithmetic and Logical Unit
AMS	AustriaMicroSystem
CISC	Complex Instruction Set Controller
EDA	Electronic Design Automation
GDS	Graphical Database System
HC	Handshake Component
HDL	Hardware Description Language
ISR	Interrupt Service Routine
IF&ID	Instruction Fetch and Instruction Decoding
LEF	Library Exchange Format
MIPS	Million Instruction Per Second
PC	Program Counter
PSW	Program Status Word
ROM	Read Only Memory
SPICE	Simulation Program with Integrated Circuit Emphasis
SRAM	Static Random Access Mem
VLSI	Very Large Scale Integration

Chapter 1

Introduction

At the moment, most of the commercial digital designs are synchronous in nature. In such circuits, there is usually a global clock signal which controls and synchronizes the data movement from one register to another. The correct functioning of these synchronous digital circuits depends on the distribution of the global clock signal. However, as the clock speed increases and the circuit size grows enormously over the years for VLSI digital designs, the proper distribution of the global clock signal becomes increasingly difficult in order to avoid the clock skew problem, which results in unreliable digital circuits. In addition, the power consumption of the clock tree also constitutes a significant amount especially for low-power digital designs. Consequently, there is an increasing

research interest in the field of asynchronous circuits over the years especially in the academic arena. Asynchronous circuits are fundamentally different from synchronous circuits in the way that there is no global clock signal present. Instead, asynchronous circuits makes use of handshaking signals, which acts as local clocks that are not in phase and with varying period, to perform the control and synchronization of data movement as illustrated by Fig 1 below [1]. In this way, the registers in asynchronous circuits are only clocked where and when needed by the handshake signals.

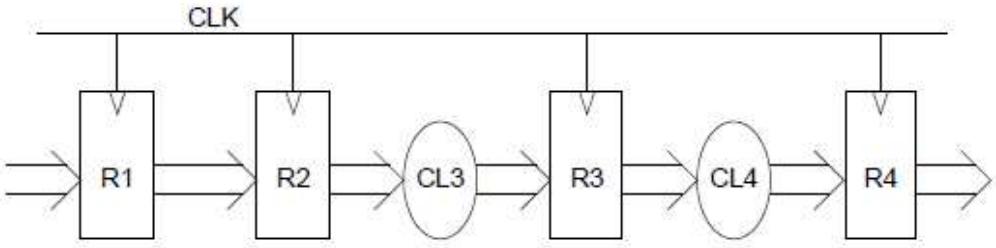


Figure 1.1: Synchronous pipeline stages controlled by clock signal

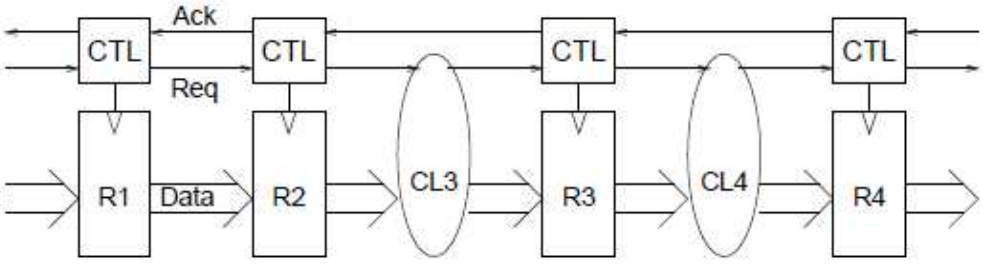


Figure 1.2: Asynchronous pipeline stages controlled by handshake signals

1.1 Synchronous Vs Asynchronous

As mentioned earlier, the main difference between synchronous circuits and asynchronous circuits lies in the data synchronization and communication method adopted. Synchronous circuits use a global clock signal to enforce data synchronization and communication whereas asynchronous circuits use handshake signals to achieve the same purpose. This difference gives the asynchronous circuits some interesting advantages over the synchronous counterparts. Some of the often mentioned advantages are listed below [1].

- No clock skew problem

  This advantage is rather obvious. The absence of a global clock signal eliminates the clock skew problem faced in synchronous circuits.

- Low-power consumption

  The absence of clock tree in asynchronous circuits leads to practically zero stand-by power consumption when the circuits are idle. For some synchronous circuits with special sleep mode operation where the clock oscillator is turned off when the sleep mode is activated, it can also achieve practically zero stand-by power consumption. However it suffers the penalty of a long delay (waiting for the oscillator to stabilize) upon exiting the sleep mode. On the other hand, for asynchronous circuits, the switch between idle and active mode is almost immediate.

- Potentially high operation speed

  The time delay involved in data transfer is determined by local latency for asynchronous circuits as compared to synchronous circuits where it is determined by the global worst case latency.

- Robust towards variations in supply voltages and fabrication process.

  Timing assumption is based on matched delays for bundled data protocol, and for asynchronous circuits that adopt the dual-rail protocol, the circuits are even quasi delay insensitive or completely delay insensitive.

- Less emission of electromagnetic noise

  For asynchronous circuits, the ticking of local clocks (handshaking signals) is generally random, leading to less emission of electromagnetic noise.

On the other hand, asynchronous circuits also come with some disadvantages as compared to synchronous circuits. The handshaking signals that used in asynchronous circuits as a replacement of the global clock signal in synchronous circuits often results in an overhead in terms of area and sometimes operation speed when the delay involved in the control signals are large. In addition, a complete and mature set of CAD tools has been developed for designing VLSI synchronous digital circuits over the years and such CAD tools greatly facilitate the entire design flow. However, for designing asynchronous circuits, there is much less CAD tools support out there for designers, making the design flow much more difficult as compared to synchronous circuits.

1.2 Asynchronous Handshake Protocols

There are several different handshake protocols out there in the field of asynchronous circuits. In this section, I will briefly discuss two most commonly used handshake protocols for designing asynchronous circuits: bundled data protocol and dual-rail protocol. For this project, the dual-rail four phase protocol is used to synthesize the asynchronous core of the 8051 microcontroller in BALSA.

1.2.1 Bundled Data Protocol

The bundled data protocol is also sometimes called the single-rail protocol as each bit of data is represented by one signal wire. It has separate request and acknowledge signal wires that are bundled with the data. It can be further divided into 4-phase bundled data protocol and 2-phase bundled data protocol. For 4-phase bundled data protocol, the handshake signals need to return to zero before starting the next round of handshake operation. Whereas for 2-phase bundled data protocol, each transition (‘1’-to-‘0’ or ‘0’-to-‘1’) of the handshake signals represents a valid handshake operation.

The 4-phase bundled data protocol works in the following order as illustrated by the timing diagram on the left of Fig. 1.3 [1] below: 1. the sender sets the request line to high when a valid data is ready on the data bus and the acknowledge line from the receiver is low, 2. the receiver sets the acknowledge line to high once it captures the valid data on the data bus, 3. the sender then lowers the request line upon sensing an asserted acknowledge line and the data on the data bus may become invalid, 4. the receiver finally lowers the acknowledge line to indicate the completion of a successful handshake sequence. Steps 3 and 4 constitute the so-called return-to-zero phase of the handshake sequence. This return-to-zero phase is necessary as in the four phase protocol, the handshake signals are level sensitive. In contrast, for 2-phase bundled data protocol as illustrated by the timing diagram on the right of Fig. 1.3, the return-to-zero phase is not needed since the handshake signals in this case are transition sensitive. The 2-phase protocol is potentially faster than the 4-phase protocol since there is no extra return-to-zero phase. However, circuits that are transition sensitive are much more complex and hard to design than circuits that are level sensitive and they often results in a large overhead in terms of circuit area.

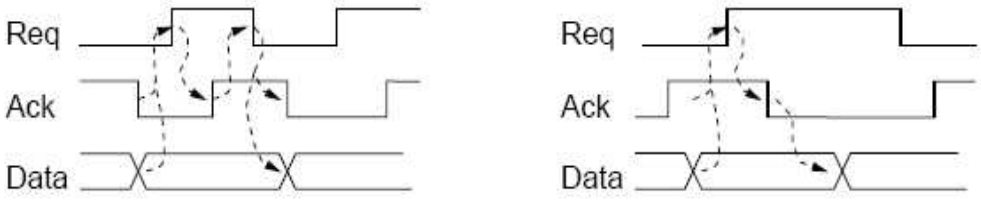


Figure 1.3: Four-phase and two-phase bundled data protocol

For both 4-phase and 2-phase protocols, the correct functioning of the bundled data approach relies on the proper delay matching in the request line. At the sender's side, the request line is set to high after the data is valid. To ensure the receiver captures the correct data, at the receiver's side, the request line should also only goes to logic '1' after the valid data is ready. Since there may be some delays involved in the data bus from the sender to the receiver due to some computation circuit in-between, the request line therefore needs to be delayed properly through inserting buffers to ensure the assertion only occurs after the valid data is ready on the data bus for the receiving side.

Fig. 1.4 [1] below illustrates a simple 4-phase bundled data pipeline with three stages. As shown in the figure, a delay element is inserted in the request line to match the delay of the combinational logic in the data bus. The strange looking gate in the control logic is the so-called Muller C-element, which is used very frequently in the control path of asynchronous circuits. The truth table of the Muller C-element is shown in Fig. 1.5 [1].

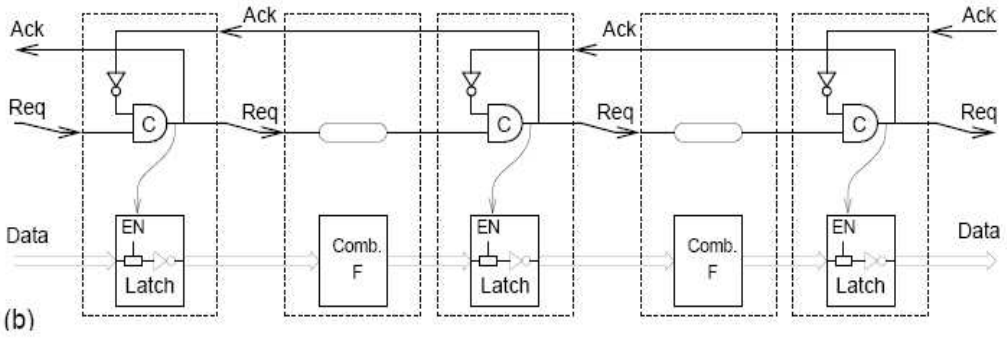
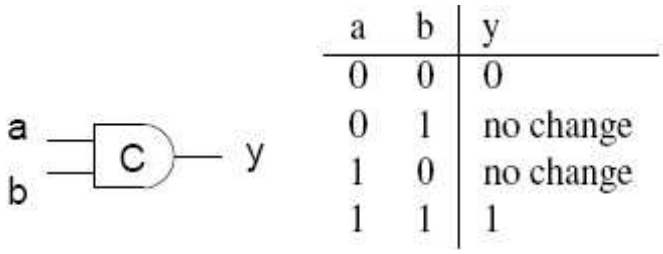


Figure 1.4: Three-stage pipeline adopting four-phase bundled data protocol



a	b	y
0	0	0
0	1	no change
1	0	no change
1	1	1

Figure 1.5: Truth table of the Muller-C element

1.2.2 Dual-rail Protocol

The dual-rail protocol uses two wires to represent one bit of data as shown in the table below [1]. One of the wires is called the true line, represented by "d.t" in the table below, and the other is called the false line represented by "d.f". When both wires are at logic '0', it represents an empty state (the data is not valid). When the true line is logic '1' and the false line is logic '0', it represents a valid logic '1'. When the true line is logic '0' and the false line is logic '1', it represents a valid

logic ‘0’. During normal operation, the two wires will not be at logic ‘1’ simultaneously.

	d.t	d.f
Empty ("E")	0	0
Valid "0"	0	1
Valid "1"	1	0
Not used	1	1

Figure 1.6: Dual-rail data encoding

Unlike the bundled data protocol, the dual-rail protocol only has one separate handshake signal (a request or acknowledge line depending on the type of channels) while the other hidden handshake signal is encoded within the dual-rail data bus.

When the sender is a push channel (a push channel activates the communication, also called the active channel), there is only one acknowledge line as shown in Fig. 1.7 [1] below. The hidden request line is actually encoded within the dual-rail data bus. For a 1-bit wide dual-rail data bus, the hidden request line can be simply represented by the output of an “OR” operation of the “d.t” and “d.f” lines. For an N-bit wide dual-rail data bus, an N-bit input Muller C element is involved and it will be elaborated on later in this chapter.

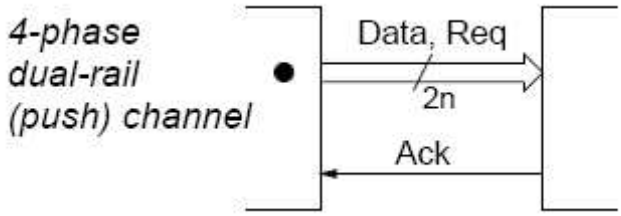


Figure 1.7: A push channel adopting four-phase dual-rail protocol

Due to the presence of empty and valid states in dual-rail protocol, the receiver side is able to distinguish a valid data from an empty data without the help of a separate request line (in the bundled data approach, the absence of the empty states requires an extra delay-matched request line to indicate the validness of the incoming data). In this way, the dual-rail protocol does not suffer the delay matching problem faced in the bundled data approach. This protocol is very robust as it is insensitive to the delays involved in the wires connecting the two communicating parties.

When the sender is a pull channel (a pull channel waits for the communication to be activated, also called the passive channel), there is only one request line. The hidden acknowledge line is encoded in the dual-rail data bus just as the hidden request line for a push channel mentioned earlier.

The dual-rail approach also can be further divided into 4-phase dual-rail and 2-phase dual-rail protocols. For a push channel using 4-phase dual-rail protocol,

there is always an empty state in-between two valid data. The handshake sequence is illustrated by Fig. 1.8 [1] below and goes as follows: 1. the sender issues a valid data on the data bus, 2. the receiver sets the acknowledge line to logic '1' once it captures the valid data on the data bus, 3. the sender then issues an empty data on the data bus after capturing a logic '1' in the acknowledge line, 4. the receiver accordingly lowers the acknowledge line upon detecting an empty data on the data bus, completing one handshake cycle.

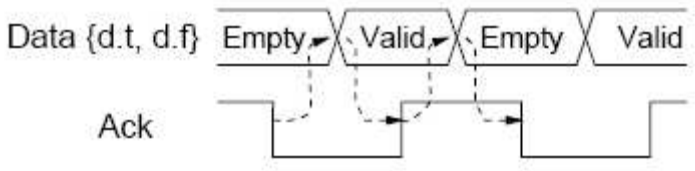


Figure 1.8: Handshake sequence of four-phase dual-rail data protocol

For the 2-phase dual-rail protocol, there are no empty state in-between valid states. Each valid data is followed immediately by another valid data once it is acknowledged. Therefore it has a shorter handshake cycle and can potentially leads to faster circuits than the 4-phase dual-rail protocol. However, as the handshake process is transition-triggered in nature, the resulting circuits are usually very complicated and hard to design.

Fig. 1.9 [1] illustrates a 3-stage 1-bit wide pipeline using the 4-phase dual-rail protocol. As shown in the figure, the acknowledge signal for each stage is simply

generated through an “OR” operation of the two data wires since the {1, 1} state of the data wires is not used. When the acknowledge signal is asserted, it indicates that the valid data (logic ‘1’ or logic ‘0’) on the data bus from the sender has been captured. On the other hand, when the acknowledge signal is low, it indicates an empty state is captured on the data bus.

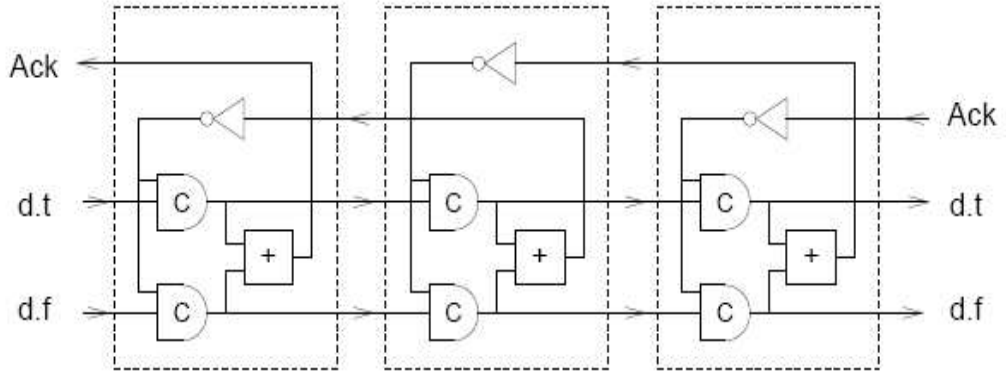


Figure 1.9: Single bit three-stage pipeline adopting four-phase dual-rail protocol

When the width of the data bus is greater than one, the acknowledge signal for each stage should only be asserted after all the individual data bits captured on the data bus are valid. In general, for an N-bit data bus, the generation of the acknowledge signal for each stage is illustrated in Fig. 1.10 [1] below. In the actual implementation, when N is large, the N-input Muller C-element in the figure is usually replaced by a tree structure of Muller C-element with 2 to 3 inputs each. This tree structure of Muller C-element used for completion detection of a valid N-bit data results in a delay in the generation of the acknowledge signal

and this delay is closely related to the size of the data bus. When N is large, this completion detection tree of Muller C-element leads to a significant delay in the acknowledge line which slows down the handshake operation.

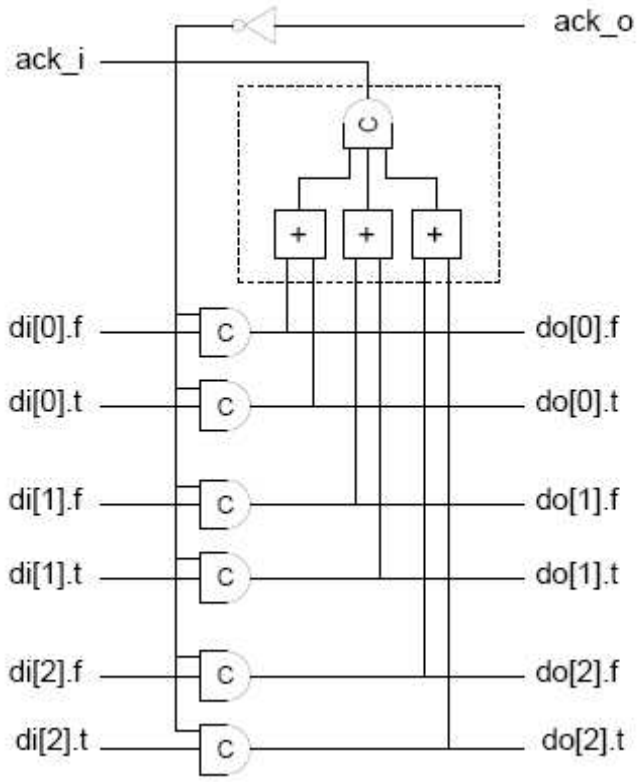


Figure 1.10: Generation of acknowledge signal for dual-rail protocol

1.2.3 Bundled Data Protocol Vs Dual-rail Protocol

The two handshake protocols discussed above have their respective advantages and disadvantages. Both of them have their own applications. For comparison

purpose, the pros and cons of the two protocols will be briefly elaborated in this section.

For the bundled data approach, its advantages are:

- It is less power consuming as compared to the dual-rail protocol since only a single wire is used to represent one bit of data which leads to a much smaller circuit for the same design.
- In general, when proper delay matching is performed, this approach tends to operate faster than the dual-rail protocol especially when the width of the data bus is large since it does not suffer from the delay involved in the completion detection tree for the case of a dual-rail protocol.

However, the bundled data approach suffers from a significant drawback: its correct operation relies on the correct delay matching in the request line. If the delay in the request line is not matched correctly to the delay in the data bus, the entire circuit may malfunction. Usually, a safety margin should be included when performing the delay matching. This makes the circuit more reliable but at the expense of a penalty in the operation speed. This approach is in general not as robust as the dual-rail approach towards variations in the fabrication process and supply voltages.

For the dual-rail protocol, the advantage is very obvious. This approach is very robust towards variations in the fabrication process and supply voltages since it is

insensitive to the delays in the wires connecting the two communicating parties. This makes it very suitable for applications that robustness is the top priority and a wide range of operating supply voltages is desired.

The disadvantages of the dual-rail protocol are:

- It consumes more power and area since two wires are used to represent one bit of data, resulting in a much larger circuit.
- It is generally slower than the bundled data approach especially if a wide data bus is used in the design. This is mainly due to the large delay involved in the control logic for completion detection as explained previously

In this work, the dual-rail four phase protocol is chosen instead of the bundled data four phase protocol (In the current version of the Balsa system, it only supports four phase protocols. The two phase protocols are not supported) in order to take advantage of the robustness of the dual-rail circuits. Also, as the supply voltage is potentially variable so that the asynchronous 8051 can operate in different modes with varying operation speed, the dual-rail protocol renders the circuit the desired robustness towards a scalable supply voltage.

1.3 Asynchronous Circuit and its Applications

Despite the fact that in today's world synchronous circuits still dominate the field of VLSI designs, there are many researches on asynchronous circuits going on especially in the academic field. A lot of successful asynchronous designs have been published over the years [2] [3] [4] [5] [6]. Many of the published designs have shown superior performance over their synchronous counterpart in terms of speed, power consumption or the Et^2 metric [7]. The appealing advantages offered by asynchronous circuits have led to the formation of several start-up companies such as Handshake Solutions [8] [9] and Theseus Logic [8] [10]. In addition, some successful commercial asynchronous products have been launched over the years such as the Telephony controllers (P83CL882, P87CL888) and Pager baseband controllers (PCA5007, PCA5010) offered by Philips [8].

.

1.4 Motivation for This Work

This work [11] is initially intended to integrate with a front-end analog signal acquisition unit to form a wearable biomedical sensor interface block which captures ECG or EEG signals from the human body and transmit the raw data or processed data to a remote host computer through wireless communication.

In the present market, there are available commercial devices that monitor the human ECG signals. However, such devices are usually quite large and needs to get the supply from the main power line. Therefore the patients using such devices have very limited space to move around and this can be very inconvenient. If the current hand-held device could be replaced with a wearable device that operated on batteries, it would be much more convenient for the patients wearing the device to walk around at easy.

In order to make sure the device is operational on batteries, the power consumption of the device must be kept low so that a longer operation time can be expected. The microcontroller, which is a significant source of power consumption for the sensor interface block, therefore should have the desirable characteristic of low-power consumption. Also as the microcontroller may need to operate in different modes through adjusting the supply voltage level, it should be functional in a wide range of supply voltage. Hence, an asynchronous microcontroller adopting the dual-rail four phase protocols is chosen to be the desired microcontroller for the sensor interface block.

The asynchronous microcontroller presented in this work closely follows the structure of a standard synchronous 8051 microcontroller invented by Intel. The 8051 microcontroller uses a CISC (Complex Instruction Set Controller) structure and is widely used in today's world. With over 200 instructions available for use, the 8051 is adequate to work as a general purpose microcontroller.

The asynchronous core of the 8051 microcontroller is synthesized using the Balsa [12] system which is an asynchronous design tool from the University of Manchester. Standard peripheral units such as the serial port and interrupt controller will be included in the overall design and they remain as synchronous circuits. The serial port is required as the data may need to be transmitted out through the USART unit. The interrupt controller is necessary as there may be external interrupts coming from other blocks which communicate with the microcontroller. In addition, since external SRAM may be used as temporary data storage memory, the asynchronous 8051 designed should be able to interface with commercial SRAM blocks.

In short, this work aims to design a low-power asynchronous version of the 8051 microcontroller which works as a local processing and control unit in a bio-medical sensor interface block which is powered by batteries.

1.5 Organization of This Thesis

This thesis presents the design of a low-power asynchronous 8051 microcontroller which is fabricated using the AMS 0.35μm technology. The organization of the thesis goes as follows. Chapter 1 briefly covers the background of asynchronous

circuits and the motivation for this work. Chapter 2 introduces the Balsa system which is an asynchronous design tool provided by the University of Manchester. Chapter 3 briefly discusses the main features of a standard synchronous Intel 8051 microcontroller. Chapter 4 elaborates on the proposed asynchronous 8051 microcontroller design with interface to external commercial memory. Chapter 5 compares four different structures of the asynchronous 8051 core designed in the Balsa framework. Chapter 6 presents the post-layout transistor-level Nanosim simulation results of the four different asynchronous cores and compares them with some existing designs. Chapter 7 finally concludes this thesis with a short summary.

Chapter 2

Balsa

Balsa [12] is the name of the free software developed by the University of Manchester. It provides a fully automatic approach for synthesizing asynchronous circuits through describing the asynchronous circuits using a dedicated high-level hardware description language which shares the same name as the software itself. In the latest version 3.5 of the Balsa system, the user can implement asynchronous designs in silicon or FPGA form with the necessary vendor-specific tools such as Cadence design framework with the back-end cell library for the desired technology and the Xilinx design software. It supports both the 4-phase bundled data approach and 4-phase dual-rail approach. Any kind of 2-phase protocols are not supported in the current version. This section intends to give a very brief

introduction on the Balsa framework used in this work, more details can be found in the tutorial guide on Balsa [12].

2.1 Introduction to the Balsa Framework

The Balsa system adopts a syntax-directed compilation method which compiles the source code written in the Balsa language into an intermediate file, the so-called breeze format, which consists of a number of communicating handshake components (there are about 46 handshake components available in the current 3.5 version of Balsa). In this approach, the compilation process is transparent to the user due to the one-to-one mapping of Balsa language construct to intermediate circuits composed of handshake components [13]. Therefore, it is possible for experienced user to optimize the circuit at the source code level through foreseeing the resultant circuits after compilation.

For illustration purpose, a short piece of source code describing a simple single-stage buffer unit is shown in the figure below. After performing a compilation of the piece of source code in Balsa, an intermediate breeze file which is made up of handshake components is generated. A picture view of the breeze file is shown in the figure. The one-to-one mapping relationship of Balsa language construct to

handshake components is shown clearly by the two figures. The filled circles represent active channels while empty circles represent passive channels.

```
import [balsa.types.basic]

procedure buffer1 (input i : byte; output o : byte) is
  variable x : byte
begin
  loop
    i -> x        -- Input communication
    ;             -- Sequence operator
    o <- x        -- Output communication
  end
end
```

Figure 2.1: A Balsa procedure describing a simple buffer

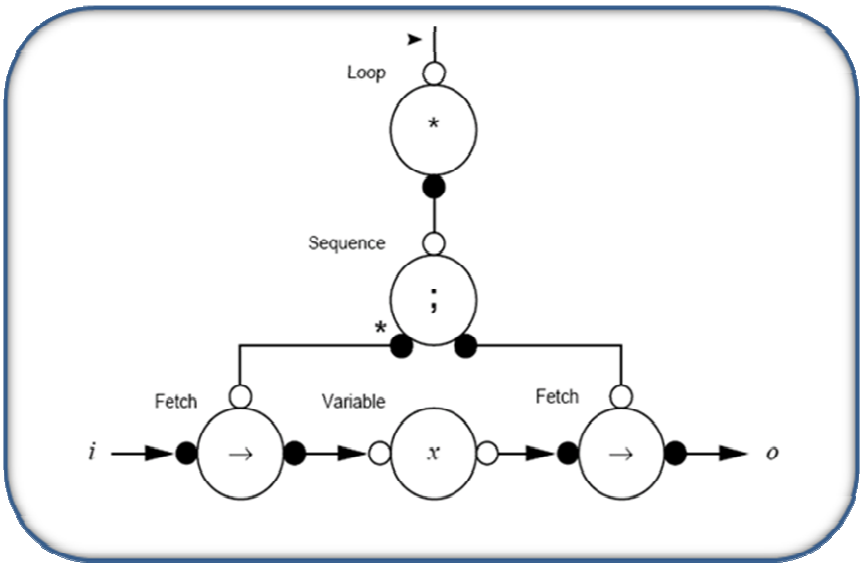


Figure 2.2: Netlist of handshake components describing a simple buffer

An overview of the design flow for the Balsa system is shown in the figure below [12]. The asynchronous design is first described in the Balsa language, thorough a compilation, the Balsa description is transformed into the intermediate breeze description which is a netlist composed of various handshake components. Behavioral simulation can be formed on this handshake component (HC) netlist using the Balsa behavioural simulation system for initial verification. A gate-level netlist consists of the standard cells of a specific technology can be generated from the HC netlist based on a mapping file which replaces each handshake component with the corresponding combination of standard cell units from the target technology. Functional simulation can be performed on the gate-level netlist to verify the functionality of the asynchronous design. After passing the functionality test, the gate-level netlist can be imported into commercial place and route tools such as Cadence SOC Encounter to generate the physical layout of the asynchronous design. Post-layout simulation can then be carried out to verify the asynchronous design at the final stage.

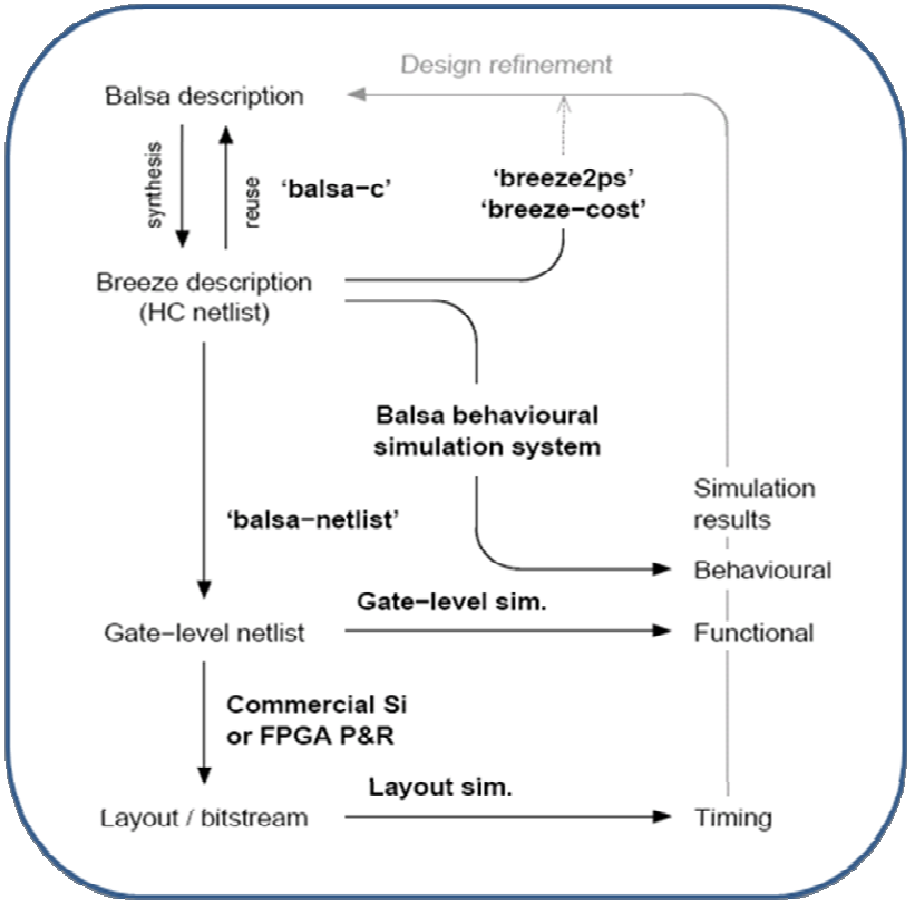


Figure 2.3: Design flow of the Balsa framework

For illustration purpose, the gate-level as well as handshake component circuit of a modulo-10 counter is shown in the figure below [12]. Those parts in blue depict the circuit at the handshake components level while the black parts shows the gate-level description of the counter. From this figure, it can be seen clearly that the intermediate handshake components in Balsa are actually made up of standard digital cell units or even just connecting wires.

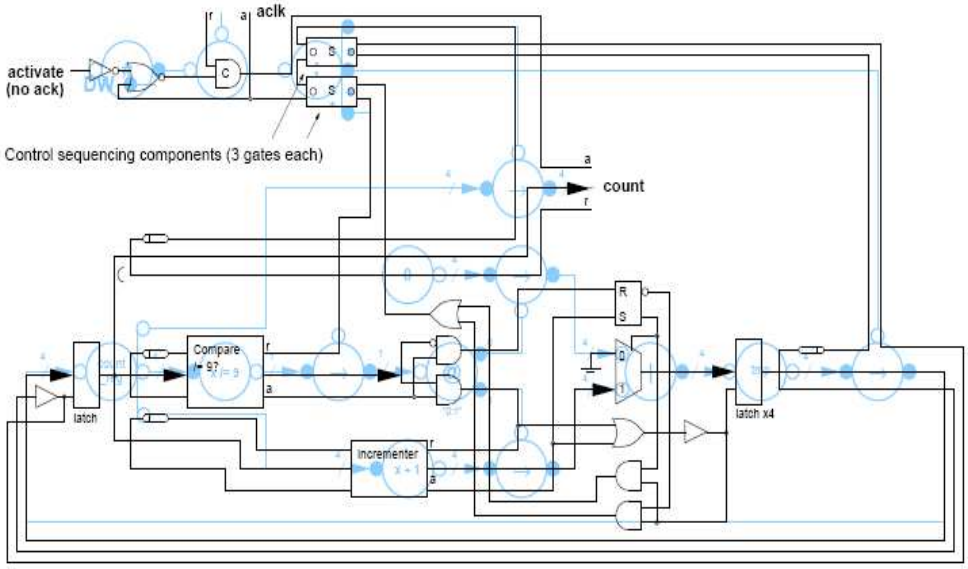


Figure 2.4: Gate-level and handshake component views of a modulo-10 counter

2.2 Introduction to the Balsa Language

This section gives a very brief overview of the Balsa language. More details on the language are found in the tutorial guide [12].

The Balsa language is a dedicated high-level hardware description language designed for the Balsa framework. It is a strongly typed language whose data types are build upon bit vectors [12]. It supports both sequential (";" operator) and concurrent ("||" operator) programming. It also has the usual features such as looping construct and conditional construct that are commonly find in other high-level HDL such as Verilog with some slight differences in the syntax. The

structure “procedure” forms the bulk of a Balsa description, which is more or less similar to the “module” structure in Verilog. Each procedure corresponds to a piece of hardware and each call to a procedure generates a new instantiation of the same piece of hardware. To avoid duplicating the same piece of hardware when multiple calls to the same procedure are involved, the procedure can be declared as a “shared” procedure so that the same piece of hardware is shared by the multiple calls. In this way, it helps to reduce the size of the circuits generated.

2.3 Balsa Vs Other Asynchronous Software

Apart from Balsa, there are a few other tools available for VLSI asynchronous circuit design, such as the Tangram framework [14] from the Philips Research Lab and the CaSCADE tool set [15] from Columbia University and University of Southern California.

The Tangram framework from the Philips Lab is very similar to the Balsa system. It also adopts a fully automated approach through compiling the source code into an intermediate netlist of handshake components first, and then replacing the handshake components with various combinations of standard digital cell units to obtain the final gate-level netlist which can be used by commercial P&R tools to generate the layout.

The CaSCADE asynchronous design tool set adopts a different design approach which is half automated half customized. It uses a language called Communicating Hardware Processes (CHP) which is based on C.A.R. Hoare's CSP [16] and E.W. Dijkstra's guarded commands [17]. Through compilation, the processes described in CHP are decomposed into a series of production-rule expansions. The so-called production-rule expansion then corresponds to some primitive custom designed cells. Overall, this approach tends to obtain better performance than the fully automated approach adopted by Balsa and Tangram when proper optimization is carried out for the custom designed primitive cells. These custom designed primitive cells tend to work more efficient than the handshake components which are made up of standard digital cells from the target technology. The disadvantage for the approach adopted by the CaSCADE lies in the inconvenience of moving from one target technology to another. The layout of each primitive cell needs to be redrawn and optimized again. On the other hand, Balsa's handshake components are made up of standard digital cells which are supplied by the vendor, therefore it is easy for an asynchronous design to transfer from one target technology to another technology. The user just needs to replace the original digital cell library with the new one and using the software to regenerate the handshake components based on the new digital cell library.

In short, the CaSCADE asynchronous design tool set tends to generate circuits with better performance due to the better circuit optimization obtained through

custom designed primitive cells for asynchronous logic. Whereas for Balsa and Tangram framework, the design process is fully automated and it is more convenient to change from one target technology to another one than the CaSCADE tool set. In this work, the Balsa framework is chosen to synthesize the asynchronous core for the 8051 microcontroller to take advantage of the relatively easier design flow and the convenience when a change of technology is desired in the future.

Chapter 3

Synchronous Intel 8051 Microcontroller

3.1 Introduction to Intel 8051

The standard synchronous 8051 microcontroller is originally developed by Intel and usually comes with a 40-pin package as shown in the figure below [19]. Intel's very early 8051 family is made using the NMOS technology. In later versions, Intel moves to the CMOS technology and the name changes to 80C51. Both 8051 and 80C51 are members of Intel's MCS-51 family of microcontroller

ICs. The generic Intel 8051 is a single-chip package that adopts the Harvard architecture which separates the storage and signal paths for instruction program and data. The Intel 8051 is one of the world's most popular microcontroller cores and it has a wide range of variants made by many manufacturers all over the world.

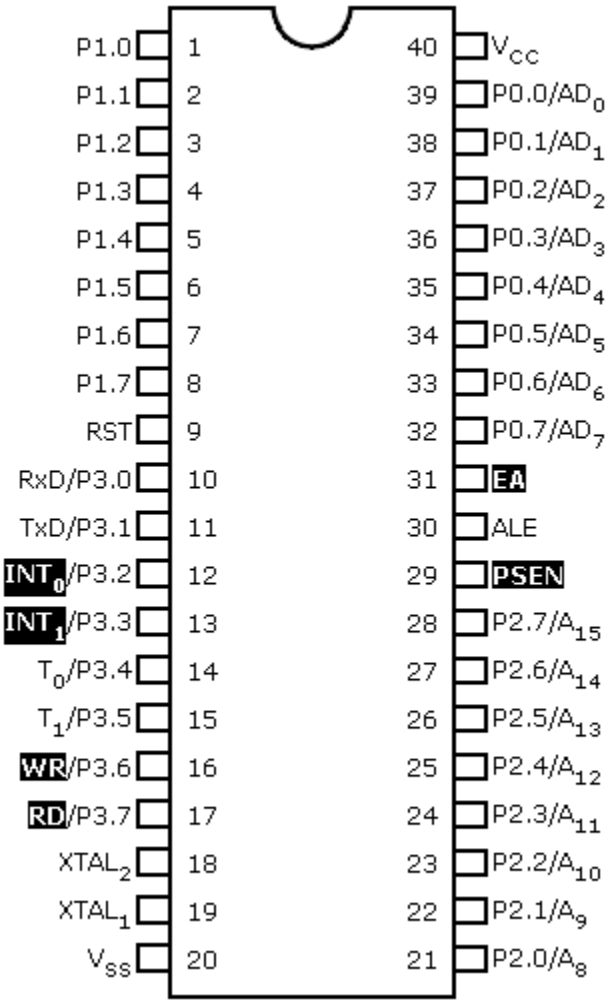


Figure 3.1: Pin configuration of a DIP packaged Intel 8051

3.2 Main Features of Intel 8051

The overall architecture of the Intel 8051 is shown in the figure below [20].

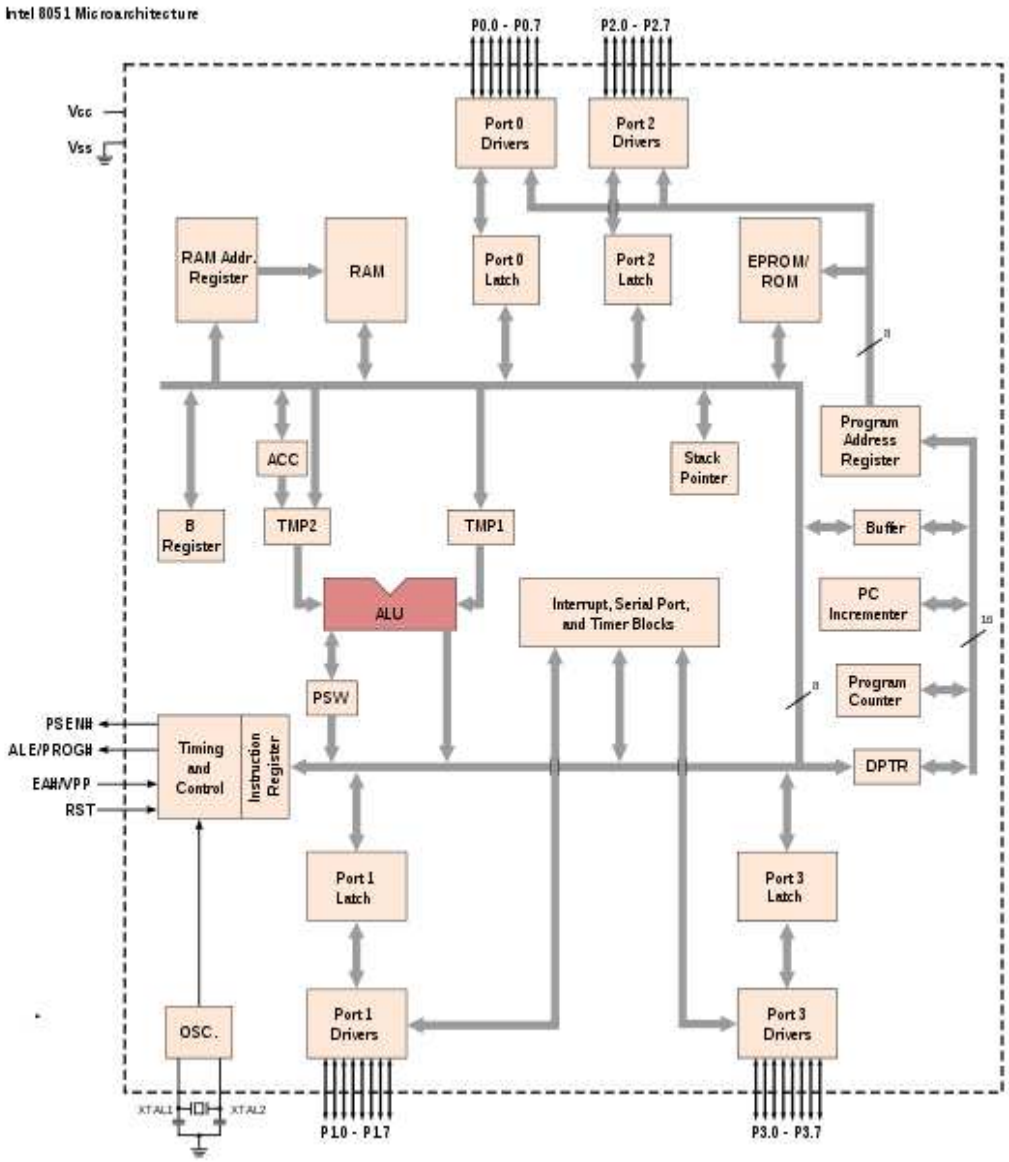


Figure 3.2: Overall architecture of a synchronous Intel 8051

The main features of a general 8051 consist of:

- 8-bit data bus

  The bus width of the Arithmetic and Logic Unit (ALU) and Special Function Registers (SFR) is 8-bit, therefore the 8051 is an 8-bit microcontroller.

- 16-bit address bus

  The address bus is 16-bit wide for both ROM and RAM, which supports up to 64 KB of memory locations.

- On-chip 128 byte of internal RAM

  The 8051 has 128 byte of SRAM built internally for temporary data storage as well as functioning as a stack for Program Counter (PC) register in the event of interrupt calls. Out of the 128 bytes, 16 bytes are bit-addressable via the bit-addressing instructions. When a larger capacity of SRAM is desired for data storage, external SRAM block up to the size of 64 KB can be interfaced with the microcontroller through the alternate functions of the I/O ports.

- On-chip 4 KB of internal ROM

  The 8051 has 4 KB of ROM built internally which is used as the program memory. The program code can also be stored in external ROM blocks up to the size of 64 KB and the instructions are then fetched via the alternate functions of the I/O ports.

- Four general purpose I/O ports each of 8 bit wide

Some of the ports are multiplexed with alternate functions as shown in the pin configuration diagram. In fact, only port 1 is a purely general purpose I/O port with no other alternate functions.

- On-chip programmable fully duplex USART serial port

  The 8051's serial port supports both synchronous and asynchronous serial communication that closely follows the existing serial communication protocols. It is software configurable and supports several modes of operations. The programmable baud rate is provided by one of the internal timers.

- Two on-chip 16-bit timers

  The 8051 has two 16-bit timers on-chip, named as timer0 and timer1. Both of the timers can be configured to operate in several different modes. The overflow of each timer can be software configured to function as a source of interrupt.

- Two-level priority interrupt handling

  There are five sources of possible interrupt for standard 8051 microcontroller: external interrupt 0, external interrupt 1, timer 0 overflow interrupt, timer 1 overflow interrupt and serial port interrupt. The interrupt handling block supports a two-level priority handling mechanism and the priority level of each source of interrupt is software configurable. During the execution of a low priority level interrupt service routine, a high priority level interrupt can cause the CPU to jump to the execution of the high priority level interrupt service routine before

actually finishing the execution of the low priority level interrupt service routine. The execution of a high priority level interrupt cannot be interrupted by any incoming interrupts.

3.3 Addressing Modes and Instruction Set of Intel 8051

Addressing modes refer to the various ways that the CPU can access data. The data can come from memory, registers or immediate data from the instruction byte. In Intel 8051, it supports five addressing modes and the complete instruction set of the 8051 can be classified into five groups as discussed later in this section.

3.3.1 Addressing Modes of Intel 8051

There are five types of addressing modes supported by the 8051: direct addressing, indirect addressing, register addressing, immediate addressing and indexed addressing [18]. Each of them will be elaborated in details in this section.

- Direct Addressing

In the direct addressing mode, the instructions are two bytes long with the first byte being the op-code and the following byte being the address. The

address refers to a location in the internal RAM or SFR space where the data stored there is to be fetched.

- Indirect Addressing

In the indirect addressing mode, the instructions are typically one byte long. The address of the data to be fetched is not directly specified in the instruction byte. Instead, the instruction byte specifies a register (usually R0 to R7 or the DPTR register when accessing external data memory) which contains the address of the data to be fetched. In this mode, both the internal RAM as well as the external data memory can be accessed.

- Register Addressing

In the register addressing mode, the instructions are all one byte long and the registers used for this mode are typically R0 to R7 from the internal RAM. The data to be fetched in this case is just the data stored in the register specified in the instruction byte.

- Immediate Addressing

In the immediate addressing mode, the instructions are two to three bytes long. The data to be fetched is specified in the instruction bytes as immediate constant. The immediate constant is usually one byte, unless the destination register is DPTR in which case the immediate constant consists of two bytes.

- Indexed Addressing

  The indexed addressing mode is mainly used for two purposes: reading data from program memory or implementing jump tables. A 16-bit register (either DPTR or PC) is used to hold the base address and the accumulator holds the index. The data to be fetched is pointed to by the sum of the base address and the index.

3.3.2 Instruction Set of Intel 8051

In Intel 8051, the instruction length varies from 1 to 3 bytes. There are 111 instruction types all together. Among which, 49 types are one-byte, 45 types are two-byte and 17 types are three-byte [18]. Taking account of the variations of each type of instructions into consideration, there are in total 255 separate instructions available for the Intel 8051 instruction set. Each 8-bit hex code represents a valid instruction except the hex code "A5", which is not used. In general, the 255 instructions can be classified into five groups: arithmetic, logic, data transfer, Boolean and branching.

- Arithmetic Operations

  The arithmetic operations of the 8051 includes: add (ADD), add-with-carry (ADDC), subtract-with-borrow (SUBB), increment (INC), decrement (DEC), decimal-adjust-accumulator (DA), multiply (MUL) and divide (DIV). For multiplication and division, the two operands are

fixed to be the contents of the accumulator and register B. The addressing mode supported in arithmetic operations are direct, indirect, register and immediate.

- Logic Operations

  The logic operations of the 8051 includes: and (ANL), or (ORL), exclusive-or (XRL), clear (CLR), complement (CPL), rotate-right (RR), rotate-right-with-carry (RRC), rotate-left (RL), rotate-left-with-carry (RLC) and swap (SWAP).

- Data Transfer Operations

  The data transfer operations include move operations (MOV, MOVC, and MOVX), stack push operation (PUSH), stack pop operation (POP) and the exchange operation (XCH). For the three types of move operations, MOV is used to transfer data to-and-from the internal RAM and Special Function Register (SFR) space. MOVC is dedicated to transfer data from the program memory to the accumulator. MOVX is used to transfer data to-and-from the external RAM.

- Boolean Operations

  The Boolean operations basically consist of all instructions that operate on a single bit. These instructions can only operate on the bit-addressable locations in the internal RAM and SFR space.

- Branching Operations

  This group of instructions includes the subroutine calls and returns, and various conditional and unconditional jumps. There are three basic types of jump operations: the short jump (SJMP), the long jump (LJMP) and the absolute jump (AJMP). The short jump uses a relative offset and the offset ranges from -128 to 127 bytes. The long jump uses a 16-bit immediate address which constitutes the last two bytes of the instruction, therefore it is able to reference to any location in the entire 64 KB program memory space. The absolute jump uses 11-bit address from the instruction to replace the lower 11-bit of the program counter (PC), therefore the location can be referenced is at most 2K bytes from the address immediately following the AJMP instruction.

3.4 Memory and Register Organization of Intel 8051

The 8051 has separate storages for program (ROM) and data (RAM). It supports up to 64K bytes of program memory with the first 4K bytes being internal on-chip ROM and the rest being external. It also supports up to 64K bytes of external RAM for data storage, not including the 128 bytes of internal on-chip RAM.

There are another 128 bytes of registers available internally which forms the so-called Special Function Register (SFR) Space.

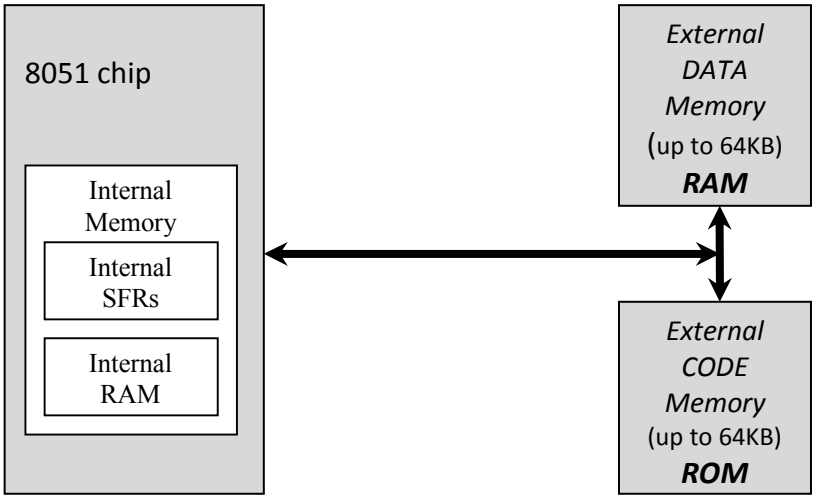


Figure 3.1: Memory organization of Intel 8051

- Program Memory

The 8051 program memory space ranges from 0000h to FFFFh. The first 4K bytes of program memory space can refer to either internal on-chip ROM or external ROM depending on the value of the external address pin $\overline{EA}$. If $\overline{EA}$ is tied to low, the entire 64K bytes of program memory space is mapped to the external ROM. If $\overline{EA}$ is tied to high, the first 4K bytes (0000 to 0FFF) refers to the on-chip ROM while the rest is mapped to the external ROM as shown in the figure below.

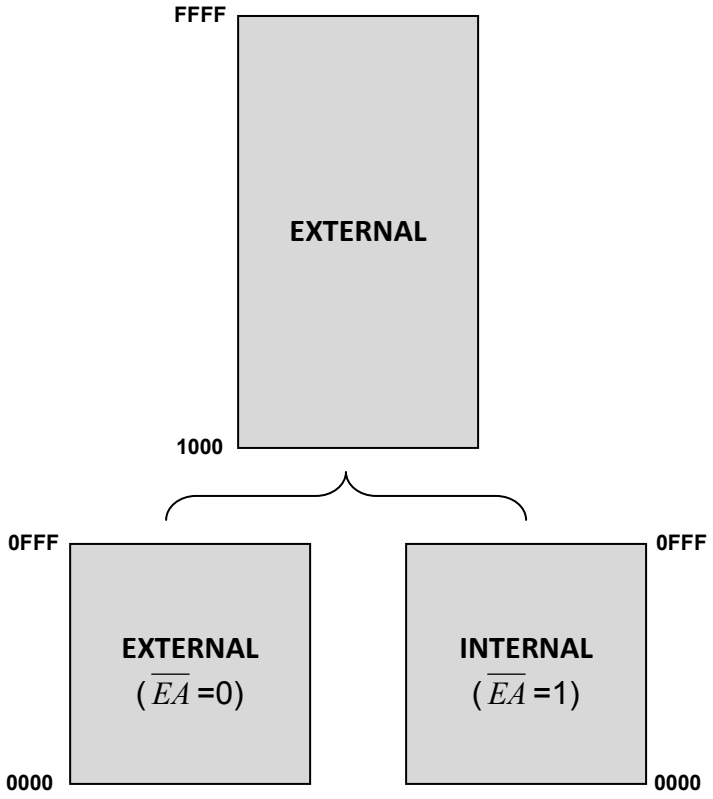


Figure 3.2: Program memory structure of Intel 8051

- Internal Data Memory and Register

The internal data memory consists of two blocks: the first 128 bytes of memory space is the internal RAM and the next 128 bytes is the SFR space as shown in the figure below.

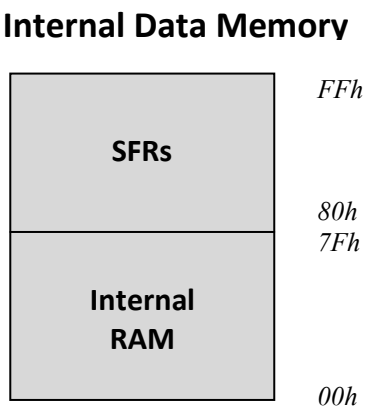


Figure 3.3: Internal data memory structure of Intel 8051

For the 128 bytes of internal RAM, the first 32 bytes forms the four banks of registers (R0 to R7), the next 16 bytes of RAM is bit-addressable and the remaining 80 bytes is the general purpose data memory as shown in the figure.

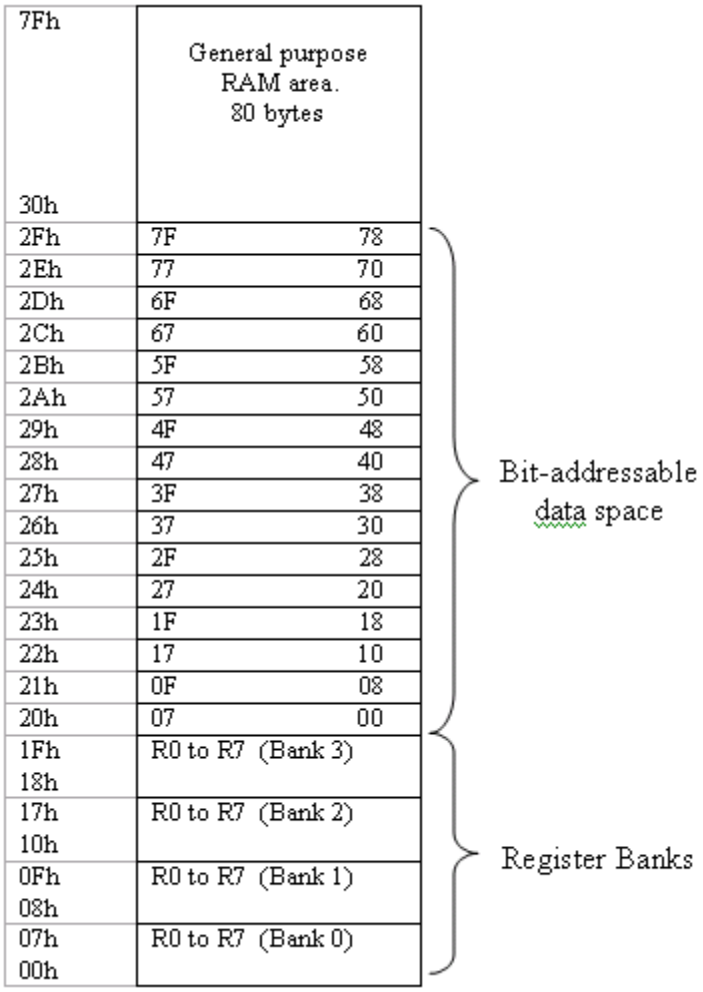


Figure 3.6: 128-byte internal ram

In the SFR space, not all the locations are utilized. In fact, only a small portion of the SFR space is associated with registers that has special functions. Some of the locations in the SFR space are bit-addressable as indicated by * in the figure below.

FFh	
F0h	B *
E0h	A (accumulator) *
D0h	PSW *
B8h	IP *
B0h	Port 3 (P3) *
A8h	IE *
A0h	Port 2 (P2) *
99h	SBUF
98h	SCON *
90h	Port 1 (P1) *
8Dh	TH1
8Ch	TH0
8Bh	TL1
8Ah	TL0
89h	TMOD
88h	TCON *
87h	PCON
83h	DPH
82h	DPL
81h	SP
80h	Port 0 (P0) *

Locations indicated by * is bit-addressable

Figure 3.7: 128-byte SFR space

- External Data Memory (RAM)

The 8051 supports up to 64K bytes of external data memory. The external memory is accessed through ports 0 and 2. Only the MOVX type of instructions can access the external data memory.

Chapter 4

Architecture of the Proposed Asynchronous 8051

In this chapter, the overall system architecture of the proposed asynchronous 8051 microcontroller will be discussed and each individual block in the system architecture will be elaborated in details.

4.1 Requirements of the Asynchronous 8051

Before the system architecture of the proposed asynchronous 8051 is discussed, I will first explain the design requirements of the asynchronous 8051 in this work.

The first requirement of the asynchronous 8051 to be designed is that it should be able to function well in a wide range of supply voltage to accommodate the possible different modes of operation (fast, medium and slow) required in the actual sensor interface block to handle different workloads respectively.

The second requirement is the presence of peripherals such as serial port and interrupt controller as in the case of standard Intel 8051. The serial port may be used to transfer the data out via the standard UART protocol and the interrupt controller is needed to handle the possible external interrupts coming from other parts in the sensor interface block such as the front-end amplifier and the ADC.

The third requirement is that the asynchronous 8051 should be able to interface well with external commercial SRAM which will be used as temporary storage of data received from the front-end ADC. Therefore the asynchronous 8051 to be designed must be able to communicate with the various commercial SRAM blocks in general.

To meet the first design requirement, the dual-rail 4-phase protocol is chosen to implement the asynchronous core of the 8051, making the asynchronous core quasi-delay-insensitive and robust towards variations in the supply voltages. To meet the second design requirement, the synchronous peripherals are included in the overall design with special interface blocks designed to enable the

communication between the synchronous peripherals and the asynchronous core. To meet the third design requirement, a novel interface block is designed to enable the asynchronous 8051 microcontroller to communicate with practically any commercial SRAM in general.

4.2 System Architecture of the Proposed Asynchronous 8051

The system architecture of the proposed asynchronous 8051 is shown in the figure below. There are five major blocks in the proposed asynchronous 8051: asynchronous core, synchronous peripherals, external memory interface, I/O ports and asynchronous internal RAM. Each individual block will be elaborated in details later on. All the signals within the asynchronous core are dual-rail in nature (a single bit of data is represented by two wires as explained in Chapter 1). Outside the asynchronous core, most of the signals are single-rail (a single bit of data is represented by one wire only) except some signals in the custom designed asynchronous RAM which are also dual-rail. Since dual-rail signals cannot communicate with single-rail signals directly, various wrapper blocks are designed around the asynchronous core to perform the data conversion between dual-rail data and single-rail data. Another function performed by these wrapper blocks is associated with the synchronization of data and handshake signals coming into and out of the asynchronous core.

In the current design, there is no internal ROM for program storage. The program code is stored in the external ROM (actually an external flash memory is used).

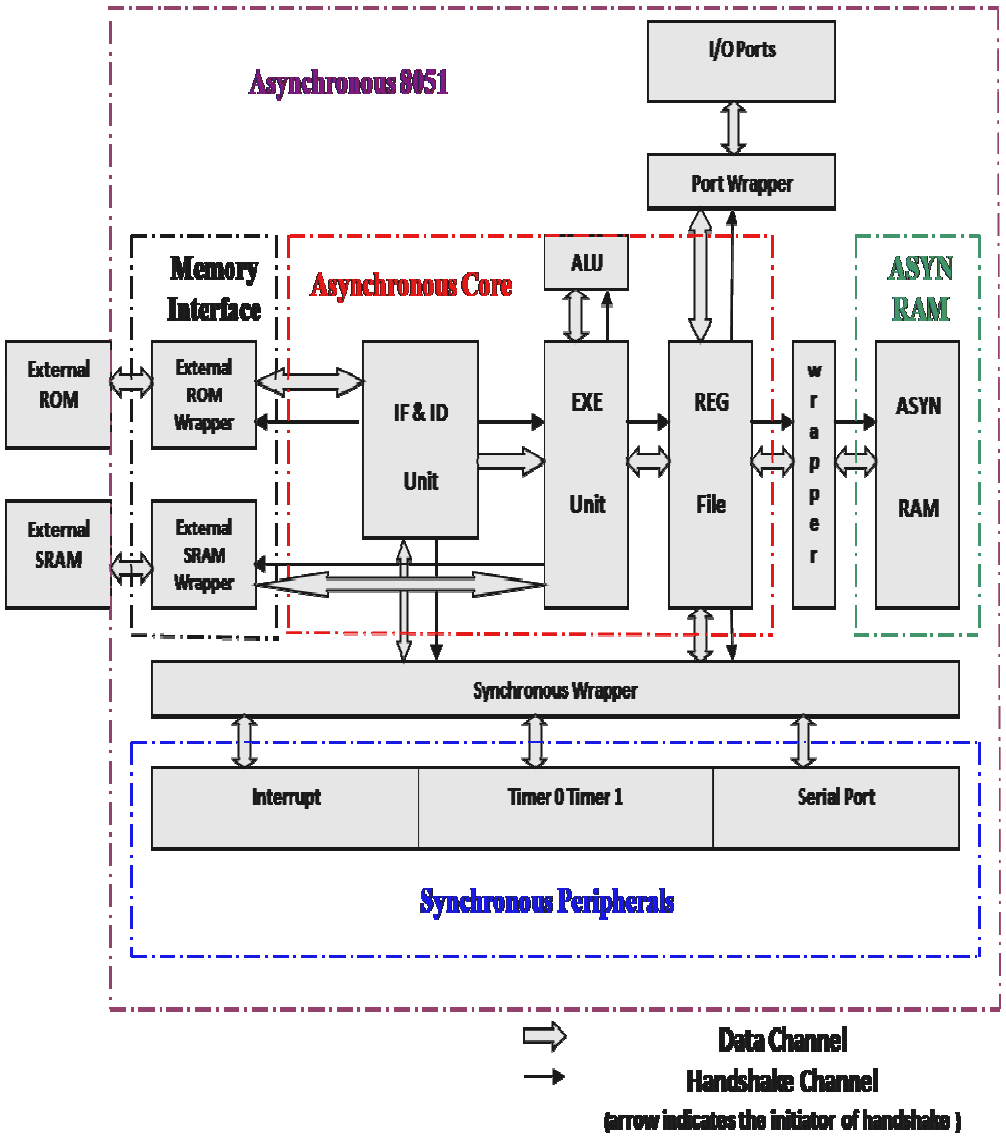


Figure 4.1: Architecture of the proposed asynchronous 8051 microcontroller

4.3 The Asynchronous Core

The asynchronous core as shown in Fig. 4.2 is synthesized using the dual-rail 4-phase protocol in the Balsa framework. As the dual-rail protocol is quasi-delay-insensitive, the asynchronous core is very robust towards variations in fabrication process and supply voltage. It is able to work in a wide range of supply voltages as illustrated in Chapter 6.

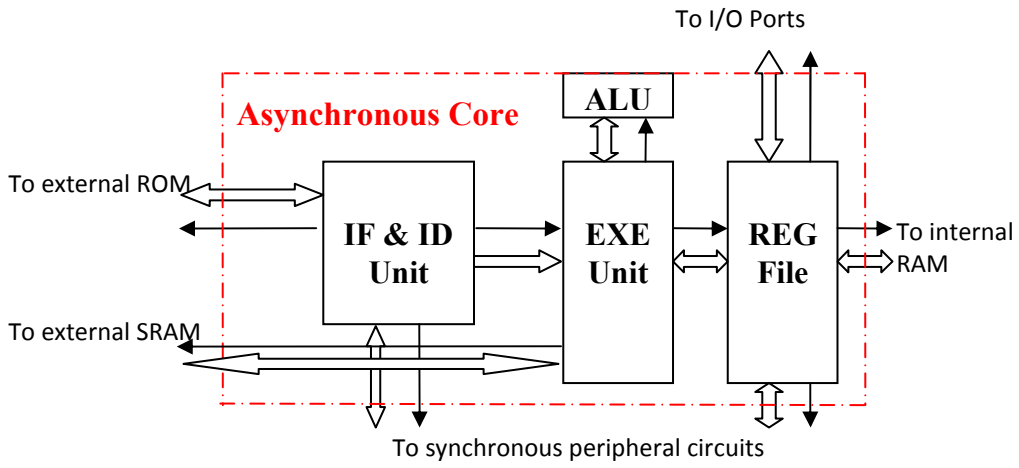


Figure 4.1: The four-phase dual-rail asynchronous core

The asynchronous core is first described using the Balsa language. After passing the behavioral simulation in the Balsa framework, it is synthesized into a gate-level Verilog netlist comprising of the standard digital cells of the AMS 0.35um technology.

There are four major sub-blocks inside the asynchronous core: the instructions fetch and decode unit (IF & ID), the execution unit (EXE), the ALU (ALU), and the register file (REG File) unit. The function of each unit will be discussed in details in the following section.

- **The Instructions Fetch and Decode Unit**

The instructions fetch and decode unit is responsible for two main tasks: the fetching and decoding of the instruction bytes and the checking of the current interrupt status coming from the interrupt controller of the synchronous peripherals. Before fetching the instruction byte from the external program memory, the current interrupt status is checked through checking two variables coming from the interrupt controller of the synchronous peripherals: "int_req" and "int_mask". The "int_req" variable is a 3-bit signal that indicates the type of incoming interrupt. The designation of this variable is shown in the table below.

int_req	Interrupt Source
000	No interrupt
001	External interrupt 0
010	Timer 0
011	External interrupt 1
100	Timer 1
101	Serial port
110	Not used
111	Not used

Table 4.1: Encoding of the different interrupt sources

The “int_mask” variable is a 2-bit signal which indicates the priority level of the currently serviced interrupt. The designation of this variable is shown in the table below. When the “int_mask” value is “00”, any incoming interrupt can be serviced immediately since there is no interrupt being serviced at that moment. When the “int_mask” value is “01”, a low priority level interrupt is being serviced. Therefore only high priority level incoming interrupts can be serviced immediately, any low priority level incoming interrupt can only be serviced when the current interrupt service routine finishes. When the “int_mask” value is “10” or “11”, all incoming interrupts have to wait for the current interrupt service to be finished since the currently serviced interrupt is of high priority level.

int_mask	Interrupt Level
00	No interrupt
01	Low priority level
10	High priority level
11	High priority level (the current high priority interrupt occurs during the execution of a low priority interrupt

Table 4.2: Encoding of the interrupt priority level

At the interrupt checking step, when no interrupt is detected, the CPU proceeds to the instruction fetch step without modifying the value stored in

the PC (Program Counter) register. If an incoming interrupt is detected, the original value held in the PC register is pushed onto the stack and the corresponding ISR (Interrupt Service Routine) address value of the incoming interrupt is loaded into the PC register to start the interrupt handling process.

At the instruction fetch and decode step, the opcode (operation code) which is the first instruction byte is fetched from the external program memory according to the value held in the PC register. The fetched opcode byte is decoded and depending on the outcome of the decoding process, the second or even the third byte may need to be fetched for the current instruction. The decoded opcode and operands of the current instruction are then passed forward to the execution step.

- **The Instruction Execution Unit**

The instruction execution unit is basically a huge "case" structure with 111 options which corresponds to the 111 instruction types as mentioned previously in Chapter 3. The default option corresponds to a "NOP" operation. The instruction execution unit first receives the decoded operation code from the Instruction Fetch and Decode unit. If it is a valid operation code, the corresponding commands to execute that operation will be carried out. Any invalid operation code will be treated as a "NOP" operation, which

simply increments the Program Counter register without changing the values stored in any other registers and memory locations.

Depending on the type of instruction executed, the execution unit may need to access the ALU unit, the Register File unit or the external SRAM during the execution process. If the execution process involves arithmetic or logical operations, the execution unit will need to send the ALU operation code and operands to the ALU unit and fetch the corresponding outcome from the ALU unit when it is ready. If the execution process involves data movement to and from the SFR space or the internal RAM, the execution unit then will need to exchange data with the Register File unit. If the instruction belongs to the MOVX type of instructions, access to the external SRAM block will be required.

- **The ALU Unit**

The ALU unit is a dedicated block for arithmetic and logical operations. In addition, it also takes care of shifting operation as well as some jump operations which modifies the PC values obtained in the early stage from the Instruction Fetch and Decode unit. The ALU unit is a case structure with 15 valid operations, namely ALU_OPC_ADD, ALU_OPC_SUB, ALU_OPC_DA, ALU_OPC_NOT, ALU_OPC_AND, ALU_OPC_XOR, ALU_OPC_OR, ALU_OPC_RL, ALU_OPC_RLC, ALU_OPC_RR, ALU_OPC_RRC, ALU_OPC_PCSADD, ALU_OPC_PCUADD,

ALU_OPC_MUL and ALU_OPC_DIV (The last two operations responsible for multiplication and division are not present in the early tape-out version, but they are included in a modified version. Since the Balsa framework does not have operators that support integer variable multiplication and division, two modules are designed specially to handle the multiplication and division operations). The ALU unit receives the ALU opcode and operands from the Execution Unit and sends the resultant value together with affected flag values back to the Execution Unit.

The ALU_OPC_ADD operation deals with all kinds of add operations (ADD) and add-with-carry operations (ADDC). The ALU_OPC_SUB operation deals with all kinds of subtract-with-borrow operations (SUBB). The ALU_OPC_DA operation takes care of the decimal-adjust operation (DA). The ALU_OPC_NOT, ALU_OPC_AND, ALU_OPC_XOR and ALU_OPC_OR corresponds to the logical “not”, “and”, “xor” and “or” operations. The ALU_OPC_RL, ALU_OPC_RLC, ALU_OPC_RR and ALU_OPC_RRC handles the rotate-left, rotate-left-with-carry, rotate-right and rotate-right-with-carry operations. The ALU_OPC_PCSADD operation calculates the new Program Counter value for various conditional jump operations through signed 2’s complement addition. The ALU_OPC_PCUADD operation derives the new Program Counter value for the MOVC type of operations through unsigned addition. The ALU_OPC_MUL is dedicated to handle the integer multiplication operation.

It is composed of a series of add and shift operations to derive the resultant product value. The ALU_OPC_DIV deals with the integer division operation. The module that handles the 8-bit integer division checks first if the scenario belongs to some special cases such as the divisor is one or powers of two where the quotient and remainder can be derived easily; if the scenario does not fall into any of the special cases, then a series of "shift and compare" operations are performed to find the quotient and the remainder and this process is mainly carried out by three sub-modules: "div_4 ( )", "div_16 ( )" and "div_32 ( )" (for this general case, the range of the quotient to be found is from 1 to 255/3 = 85 since both 255/1 and 255/2 belong to the special cases). Each of the three sub-modules is elaborated in details in the following section. The basic idea is to express the resultant quotient as the sum of several quotient terms which are to be found through a series of "shift and compare" operations. Each quotient term is of the form 2^n and the search procedure finds the quotient terms in a descending order, starting with the largest quotient term.

The module "div_4 ( )" handles the case where the sum of the remaining quotient terms is less than 4 and the algorithm for this module is described as below.

1. Compare the current dividend with the current divisor, if the dividend equals the divisor, set the second LSB of the quotient to 1 (the

quotient term is 2) and the remainder to 0, then exit; else if the dividend is less than the divisor, go to step 2; else if the dividend is greater than the divisor, go to step 3.

2. Shift the divisor to the left by one and compare it with the dividend. If the dividend is less than the new divisor, set the remainder to be the dividend (the quotient term is 0) and exit; else if the dividend equals to the divisor, set the LSB of the quotient to 1 (the quotient term is 1) and the remainder to 0, then exit; else if the dividend is greater than the divisor, set the LSB of the quotient to 1 (the quotient term is 1) and the remainder to the difference between the dividend and the divisor, then exit.
3. Set the second LSB of the quotient to 1 (the quotient term is 2) and subtract the divisor from the dividend. Let the difference be the new dividend and go to step 2 above.

The module “div_16 ( )” handles the case where the sum of the remaining quotient terms is less than 16 and the algorithm for this module is described as below.

1. Shift the current divisor to the right by two and compare it with the current dividend. If the dividend equals to the new divisor, set the third LSB of the quotient to 1 (then quotient term is 4) and the remainder to 0, then exit; else if the dividend is less than the divisor,

shift the divisor to the right by one and call div_4 ( ); else if the dividend is greater than the divisor, go to step 2.

2. Shift the divisor to the left by one and compare it with the dividend. If the dividend equals to the divisor, set the fourth LSB of the quotient to 1 (the quotient term is 8) and the remainder to 0, then exit; else if the dividend is less than the divisor, go to step 3; else if the dividend is greater than the divisor, go to step 4.
3. Shift the divisor to the right by one and subtract it from the dividend. Let the difference be the new dividend and set the third LSB of the quotient to 1 (the quotient term is 4). Shift the divisor to the right by one again and call div_4 ( ).
4. Subtract the divisor from the dividend and let the difference be the new dividend. Set the fourth LSB of the quotient to 1 (the quotient term is 8) and shift the divisor to the left by one. If the new dividend equals to the new divisor, set the third LSB of the quotient to 1 (the quotient term is 4) and the remainder to 0, then exit; else if the dividend is less than the divisor, shift the divisor to the left by one again and call div_4 ( ); else if the dividend is greater than the divisor, go to step 5.
5. Subtract the divisor from the dividend and let the difference be the new dividend. Set the third LSB of the quotient to 1 (the quotient term is 4) and shift the divisor to the left by one. Call div_4 ( ).

The module “div_32 ( )” handles the case where the sum of the remaining quotient terms is less than 32 and the algorithm for this module is described as below.

1. Compare the current dividend with the current divisor, if the dividend equals to the divisor, set the fifth LSB of the quotient to 1 (the quotient term is 16) and the remainder to 0, then exit; else if the dividend is less than the divisor, call div_16 ( ); else if the dividend is greater than the divisor, go to step 2.
2. Subtract the divisor from the dividend and let the difference be the new dividend. Set the fifth LSB of the quotient to 1 and call div_16 ( ).

Based on the above three sub-modules, the overall division algorithm is explained as below.

1. If the dividend is less than the divisor, the quotient is zero and the remainder is just the dividend; else if the dividend equals the divisor, the quotient is one and the remainder is zero; else go to step 2.
2. If the divisor belongs to the special cases: 1 or powers of two (2, 4, 8…128), the quotient and remainder can be easily obtained through shifting the dividend; else go to step 3.

3. Add eight 0's in front of both the dividend and divisor to make them both 16 bits long and the 8-bit quotient is initialized to be zero. Shift the new divisor to the left by four and then compare the shifted divisor with the dividend. If the dividend equals to the shifted divisor, set the fifth LSB of the quotient to 1 (the quotient term is 16) and the remainder is 0, then exit; else if the dividend is less than the divisor, call div_16 ( ); else if the dividend is greater than the divisor, go to step 4.
4. Shift the divisor to the left by two and compare it with the dividend. If the dividend equals to the divisor, set the second MSB of the quotient to 1 (the quotient term is 64) and the remainder is 0, then exit; else if the dividend is less than the divisor, go to step 5; else if the dividend is greater than the divisor, go to step 6.
5. Shift the divisor to the right by one and compare it with the dividend. If the dividend equals to the divisor, set the third MSB of the quotient to 1 (the quotient term is 32) and the remainder is 0, then exit; else if the dividend is less than the divisor, shift the divisor to the right by one and subtract it from the dividend, set the fifth LSB of the quotient to 1 (the quotient term is 16) and call div_16 ( ); else if the dividend is greater than the divisor, subtract the divisor from the dividend and let the difference be the new dividend, set the third MSB of the quotient to 1 (the quotient term is 32), shift the divisor to the right by one and call div_32 ( ).

6. Subtract the divisor from the dividend and let the difference be the new dividend. Set the second MSB of the quotient to 1 (the quotient term is 64) and shift the divisor to the right by two. Call div_32 ( ).

- **The Register File Unit**

The Register File Unit controls the access to the SFR space and internal RAM of the 8051. There are four sub-blocks in this unit: ReadRAM ( ), ReadBitRAM ( ), WriteRAM ( ) and WriteBitRAM ( ). The same address bus is shared by the four sub-blocks. At any time when the Register File Unit is accessed by the Execution Unit, only one of the four sub-blocks will be activated. The ReadRAM ( ) block is responsible for retrieving 8-bit data from the SFR space or the internal RAM. The ReadBitRAM ( ) block is responsible for retrieving one bit of data from bit-addressable memory locations. The WriteRAM ( ) block writes 8-bit data into the SFR space or the internal RAM. The WriteBitRAM ( ) block controls the writing of a single bit of data into memory locations that support the bit-addressing mode.

The operations of the four sub-blocks are very similar. For illustration purpose, the operation of the ReadRAM ( ) block is briefly described as follows. First of all, the MSB of the address bus is checked to determine the destination memory space, being the SFR space (MSB = 1) or the internal RAM (MSB = 0). If the destination memory space is the SFR space, it is

again separated into two scenarios since the SFR space resides in two different regions for this design. Most of the registers in the SFR space lie within the asynchronous core, except some registers that are related to interrupt handling, timer operation and serial port are found in the synchronous peripherals. If the target register is within the asynchronous core, the stored value can be retrieved directly. If the target register falls in the synchronous peripherals, the address bus together with some control signals are passed to the synchronous peripherals via the synchronous peripheral wrapper block (will be covered in later sections of this Chapter) to retrieve the data from the destination register that resides in the synchronous peripherals. If the destination memory space belongs to the internal RAM, the seven least significant bits of the 8-bit address bus are passed to the internal RAM as the address of the requested location and the corresponding value stored there can be retrieved thereafter.

4.4 The Interface to Synchronous Peripherals

Since all signals within the asynchronous core are encoded using the dual-rail protocol, in order to communicate with signals coming from the synchronous peripherals where data is single-rail in the sense that one bit of data is represented by one wire only, some kind of data conversion between dual-rail and single-rail data is required. In addition, since the asynchronous core uses handshake signals

to control data movement whereas the synchronous peripherals rely on the clock signal instead, some kind of data synchronization is also required when the data moves across the boundary. These two tasks (data conversion and data synchronization) are performed by the wrapper block called "Synchronous Wrapper" located in-between the asynchronous core and the synchronous peripherals.

For the data conversion part, the main task is to convert data from its dual-rail representation into its single-rail representation and vice vs. According to the definition of the dual-rail representation introduced early in Chapter 1, converting data from its dual-rail representation to its single-rail representation is rather straight forward: just ignore the false lines of the dual-rail data and take the true lines of the dual-rail data to be its single-rail representation. Converting data from its single-rail representation into its dual-rail representation requires some slight manipulations as illustrated by the figure below.

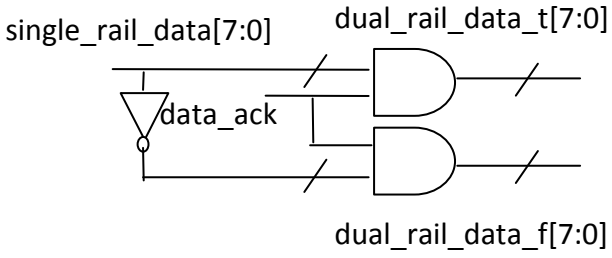


Figure 4.2: Data conversion from single-rail into dual-rail representation

Through an “AND” operation with an acknowledge signal “data_ack”, the 8-bit single-rail data “single_rail_data[7:0]” from the synchronous peripherals is converted to be the true lines “dual_rail_data_t[7:0]” of its dual-rail representation. Similarly, the inversion of the single-rail data is converted to be the false lines “dual_rail_data_f[7:0]” of its dual-rail representation. The 8-bit dual-rail data formed by “dual_rail_data_t[7:0]” and “dual_rail_data_f[7:0]” are now ready to be used by the asynchronous core.

In order to indicate that the data obtained through the above conversion mechanism (between single-rail and dual-rail) is the correct valid data, there is a need to perform some kind of data synchronization since data is moving from one domain to another different domain.

In the case of data moving from the asynchronous core to the synchronous peripherals (converting data from dual-rail to single-rail), the handshake signals and data bus coming out and into the wrapper block is shown by the figure below. For simplicity, other signals such as the address bus and read/write control lines are not shown in the figure below and they are treated in a similar way as the data bus.

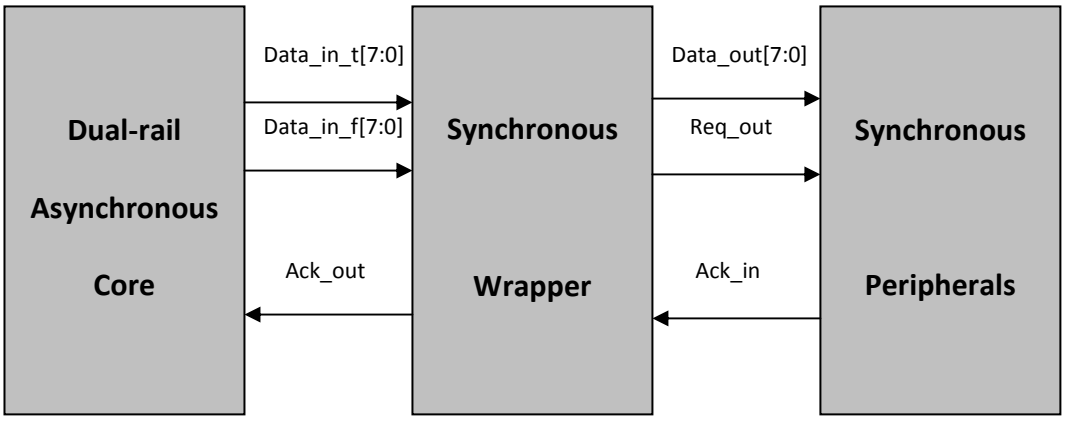


Figure 4.3: Data moving from asynchronous domain to synchronous domain

The asynchronous core domain initiates the communication through outputting a valid data on the dual-rail data bus. The dual-rail data is then converted into its single-rail representation by the synchronous wrapper and sent to the synchronous peripherals domain. Upon capturing a valid single-rail data, the synchronous peripherals domain sends a high acknowledge signal to the synchronous wrapper which then passes it to the asynchronous core domain. This completes the positive phase of the handshake cycle between the two different domains. In the return-to-zero phase, the asynchronous core puts an empty data on the data bus and synchronous peripherals domain lowers the acknowledge line upon detecting an invalid single-rail data coming from the synchronous wrapper block. The purpose of the extra signal "Req_out" in the above figure and the generation of the acknowledge signal "Ack_out" by the synchronous wrapper block is explained in the following paragraphs.

The validness of a single bit of data in its dual-rail representation can be checked through performing an “XOR” operation on the true and false lines. A logical “1” output indicates a valid data while a logical “0” indicates an empty data. Thus the validness of a data bus in its dual-rail representation can be checked through performing an “AND” operation on all the signals that indicating the validness of each single bit of the data bus in shown in the figure below. A logical “1” at the output “Data_ready” indicates a valid data on the bus while a logical “0” indicates an invalid data on the bus (empty or partially valid data). Therefore the arriving dual-rail data “Data_in_t[7:0]” and “Data_in_f[7:0]” from the asynchronous core is able to indicate its own validness through the dual-rail encoding.

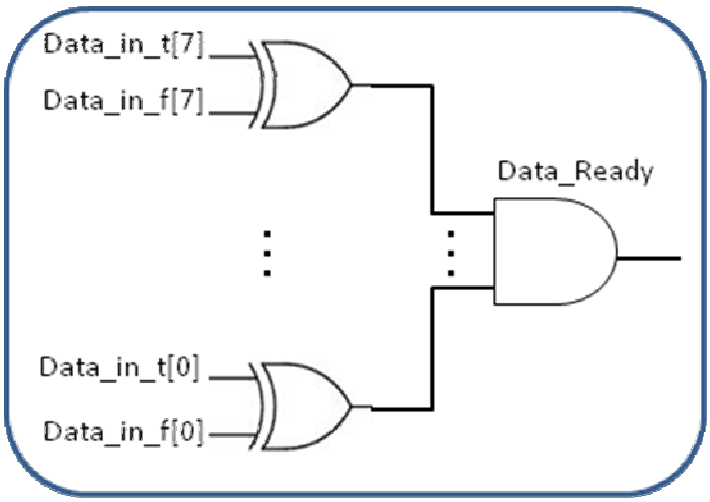


Figure 4.5: Generation of validity indicating signal of dual-rail data bus

However after conversion, the data in its single-rail representation “Data_out[7:0]” (basically wires connecting to the true lines of the dual-rail data) cannot indicate its own validness without the extra indicating signal “Req_out”.

Consider the case that when the incoming dual-rail data is in the empty state where both true lines and false lines are all zero, after conversion the single-rail data obtained is zero and it may be mistaken as a valid zero by the synchronous peripherals domain. Also, if the active clock edge of the capturing register in the synchronous peripherals domain occurs during the transition state where the incoming dual-rail data changes from the empty state to a valid state, a wrong single-rail data may be captured since not all the true lines of the dual-rail data have changed to the correct state. Therefore an extra indicating signal "Req_out" is required in this case to ensure that the single-rail data captured by the synchronous peripherals domain is valid. This indicating signal is actually derived from the "Data_ready" signal in Fig. 4.5, which is similar to that used for completion detection in the asynchronous dual-rail protocol by replacing the "AND" gate with a Muller C element. In this way, the single-rail data "Data_out[7:0]" (the true lines) is guaranteed to be a valid data when the indicating signal "Req_out" is at logic "1". Each time when the synchronous peripherals domain captures the single-rail data, it checks the indicating signal coming out from the wrapper block to determine the validness of the data captured.

The signal "Ack_in" coming out from the synchronous peripherals domain and the "Ack_out" signal sent out by the synchronous wrapper block are generated according to the timing diagram shown below. During the positive phase, the acknowledge signal "Ack_out" is asserted at the next rising edge of the clock

signal after capturing an asserted indicating signal “Req_out” in the previous clock rising edge. During the return-to-zero phase, in order to reduce the communication cycle, the acknowledge signal “Ack_out” is lowered immediately once a low “Req_out” signal is captured desipte that the “Ack_in” signal coming from the synchronous peripherals is only lowered later on.

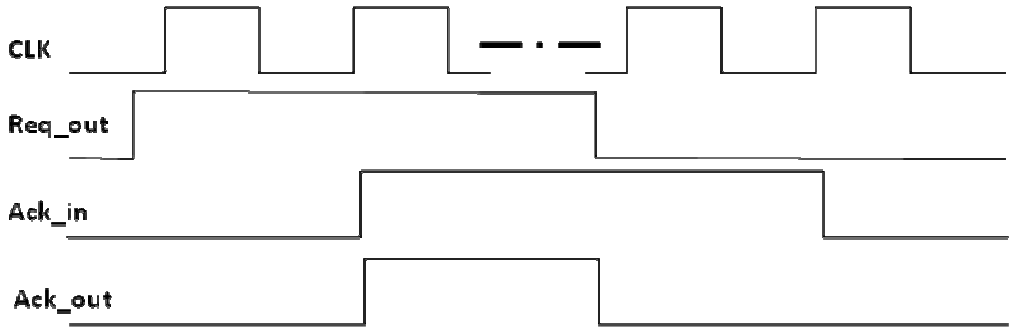


Figure 4.6: Timing diagram of the handshake signals

In the case of data moving from the synchronous peripherals domain to the asynchronous core (converting data from single-rail to dual-rail), the handshake signals and data coming out and into the wrapper block is shown by the figure below. Again for simplicity, other signals such as the address bus and read/write control lines are not shown in the figure below and they are treated in a similar way as the data bus.

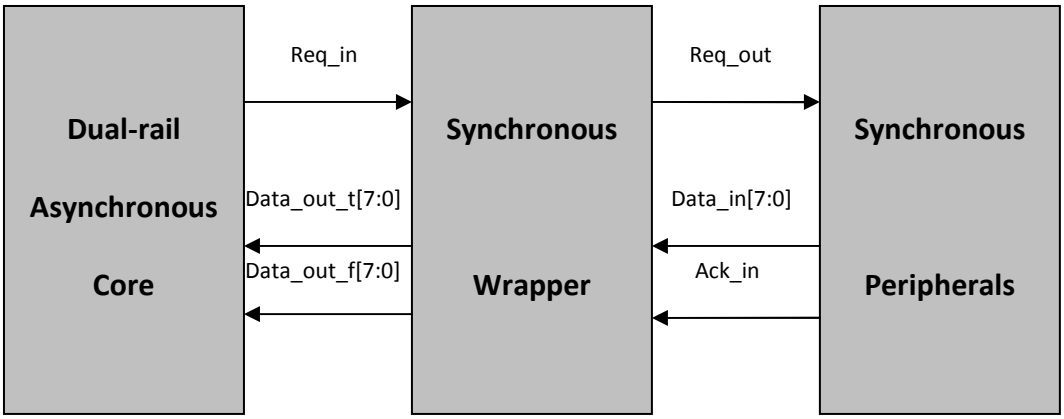


Figure 4.4: Data moving from synchronous peripherals to asynchronous core

The asynchronous core domain initiates the communication through asserting the request signal "Req_in". The wrapper block forwards this request signal to the synchronous peripherals domain via the "Req_out" signal. Upon detecting a high "Req_out" signal at the rising edge of the clock signal, the synchronous peripherals issues the requested single-rail data on the "Data_in[7:0]" bus together with a logic "1" on the acknowledge line "Ack_in". In this case, the acknowledge signal "Ack_in" serves as the validity indicating signal for the single-rail data "Data_in[7:0]" coming from the synchronous peripherals domain. Within the synchronous wrapper block, the input single-rail data "Data_in[7:0]" is converted into its dual-rail representation and sent out through the output data bus "Data_out_t[7:0]" and "Data_out_f[7:0]". Internally, the acknowledge signal used to perform the conversion (the "data_ack" signal in Fig. 4.3) is not simply the input "Ack_in" signal, but rather the output of a 2-input Muller C-element with "Ack_in" as one of the inputs and the request signal "Req_in" from the

asynchronous core domain as the other input. In this way, a valid dual-rail output data will only appear on the outputs when the input acknowledge signal “Ack_in” is at logic “1” and it will only return to the empty state when both “Ack_in” and “Req_in” signals are at logic “0”. The Muller C-element is required to prevent the valid dual-rail data from returning to the empty state before being latched by the asynchronous core domain. The timing diagram for the generation of the acknowledge signal “Ack_in” is same as the one shown in Fig. 4.6 for the previous case.

4.5 The Interface to External Commercial Memory

In the proposed design, the program code is stored in the external commercial ROM and an external commercial SRAM is used for temporary data storage, therefore, the asynchronous 8051 needs to interface with both external commercial ROM and SRAM. However, since the interface to external commercial ROM is similar to the interface to external SRAM, only the interface to external commercial SRAM will be elaborated in this section.

Just like the interface to synchronous peripherals covered in the previous section, the interface to external commercial SRAM also performs two main tasks: data conversion between dual-rail data (from the asynchronous core domain) and

single-rail data (from the external commercial SRAM domain) as well as data synchronization to ensure the validness of data traversing across the two different domains.

For illustration purpose, the case of a reading operation on the external SRAM block by the asynchronous core is discussed below to demonstrate the internal data conversion and synchronization performed by the wrapper block "Extenal SRAM Wrapper" shown in Fig. 4.1 previously in this Chapter.

At the start of a reading operation, the asynchronous core sends out several signals to the wrapper block: a data request signal, a read signal, an external SRAM access enable signal and an address bus as shown in Fig. 4.8 below. The external SRAM access enable signal "Xram_en_out" acts as a control signal to select one of the two external memory chips (external ROM or external SRAM) to be active since the two external memory chips share the same address bus and data bus. The data request signal "Data_req" is a handshake signal that initiates the reading process. Other data signals are all in the dual-rail representations and needs to be converted into the corresponding single-rail representations by the wrapper block before being passed to the external SRAM block. The conversion process from dual-rail to single-rail is done in a similar manner as covered previously in the synchronous peripheral interface design. In Fig. 4.8, the "Read_out" signal, which is connected to the "Read" input of the external SRAM chip, is not simply a wire connecting to the true line of the dual-rail signal

("Read_in_t"). It is the output of some combinational logics with an internal validity indicating signal as one of the inputs. Basically through the combinational logics, it is ensured that when the "Read_out" signal is at logic "1', the single-rail address bus "Addr_out[7:0]" contains the valid address values. The inversion of the "Read_out" signal is connected to the "Output Enable" input of the external SRAM chip.

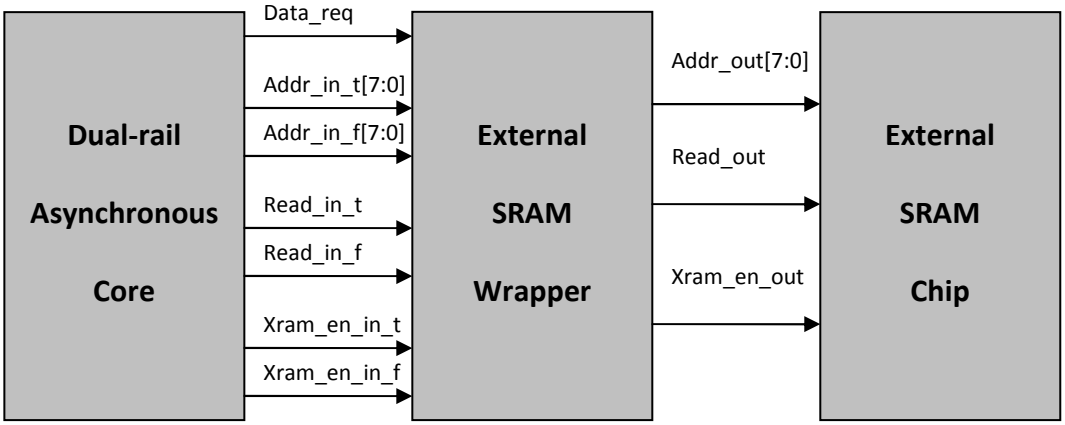


Figure 4.5: First phase of a read operation on the external SRAM

Upon receiving the reading address and the output enable signal, the external SRAM outputs the requested data on the data bus after a specified delay T. The output data "Data_in[7:0]" from the SRAM block is in the usual single-rail representation and it is converted to its dual-rail representation by the wrapper block before being sent out to the dual-rail asynchronous core as shown by the figure below.

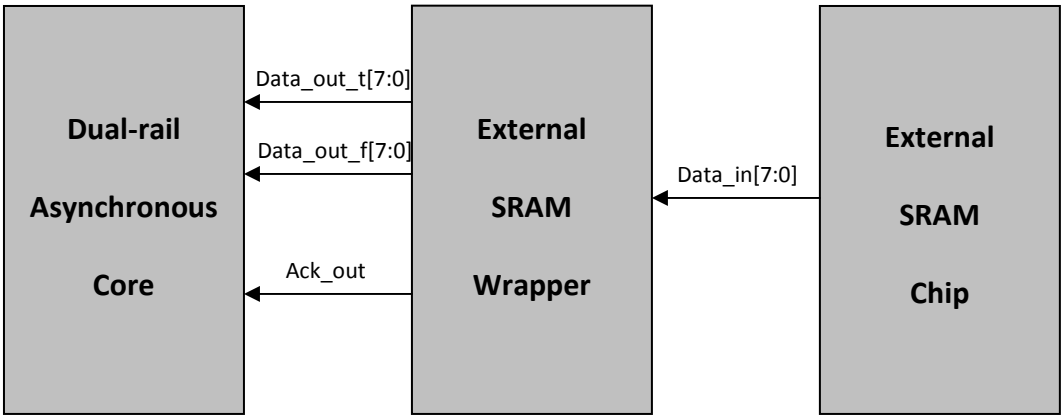


Figure 4.6: Second phase of a read operation on the external SRAM

As mentioned previously, to convert data from the single-rail representation to the dual-rail representation, a validity indicating signal is required to ensure that the single-rail data being converted is the desired valid data. Internally within the wrapper block, the validity indicating signal "sram_ack" is generated by a counter module called "SRAM_Wrapper Counter" as shown in Fig. 4.10 below.

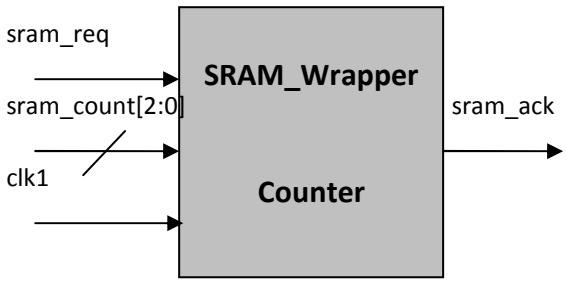


Figure 4.7: Symbol view of "SRAM_Wrapper Counter" module

The assertion of the input signal “sram_req” (which is an internal signal in the wrapper block) starts the 3-bit counter within this module and the counting starts from zero. This internal signal “sram_req” is derived based on the request signal “Data_req” from the asynchronous core and the validity checking of the relevant dual-rail data from the asynchronous core as well. Essentially, when “sram_req” is asserted, the single-rail data on the address bus “Addr_out[7:0]” shown in Fig. 4.8 is ensured to be the correct valid address value.

The 3-bit input signal “sram_count[2:0]” (in the early version, this 3-bit wide signal is adjusted through external pins; in the later version, it is adjusted through modifying the values stored in a dedicated internal register in the SFR space) is adjustable and sets the overflow value for the internal counter based on the actual access time of the commercial SRAM used and the frequency of the clock signal driving the counter. When the internal counter reaches the overflow value, it asserts the output acknowledge signal “sram_ack” which indicates the validity of the single-rail data “Data_in[7:0]” shown in Fig. 4.9 coming from the external SRAM block.

The last input signal “clk1” defines the clock period of the internal counter. However, it is not used directly to drive the internal counter. The actual driving clock of the internal counter is “clk_sram”, which is a gated version of the “clk1” signal with the “sram_req” signal. In this way, it helps to reduce the power

consumption of the counter module as the 3-bit counter will only be running when a read or write operation on the external commercial SRAM is desired.

The timing diagram for the generation of the acknowledge signal “sram_ack” is shown in the timing diagram below. In the positive phase, the acknowledge signal is asserted after the internal counter reaches the preset overflow value. In the return-to-zero phase, the acknowledge signal is lowered immediately after the request signal “sram_req” is lowered.

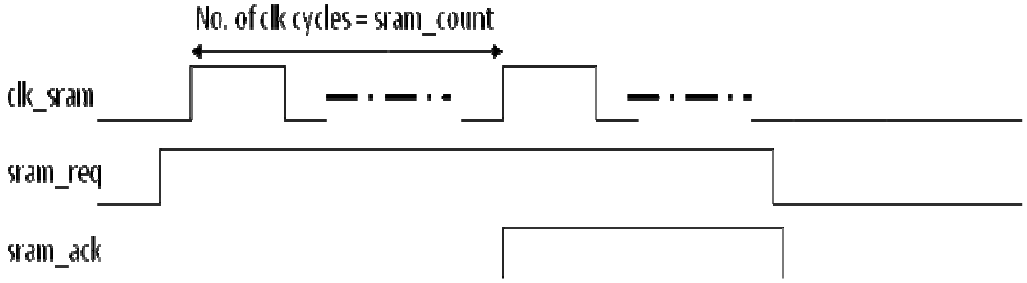


Figure 4.11: Timing diagram of the acknowledge signal “sram-ack”

In short, a reading operation on the external SRAM by the asynchronous core goes in the following order: the asynchronous core sends out a read request signal and the relevant control signals and address bus to the wrapper block; the wrapper block processes these signals and converts the dual-rail signals into their single-rail representations and sends them to the external SRAM block, meanwhile, the internal signal “sram_req” is asserted which starts the internal counter; upon

receiving the valid single-rail control signals and address bus “Addr_out[7:0]” from the wrapper block, the external SRAM block outputs the requested data onto the data bus “Data_in[7:0]” after a specified delay time (the access time defined in the actual datasheet of the external commercial SRAM chip); when the internal counter reaches the preset overflow value which is defined by the 3-bit signal “sram_count[2:0]”, the acknowledge signal “sram_ack” is asserted (the time taken from the starting of the counter to the assertion of the acknowledge signal is typically slightly longer than the access time of the external SRAM chip) which indicates the validity of the single-rail data “Data_in[7:0]” coming from the external SRAM; once the acknowledge signal “sram_ack” is asserted, the conversion of the single-rail data from the external SRAM to its dual-rail representation is carried out within the wrapper block; the converted dual-rail data is sent to the asynchronous core together with an asserted acknowledge signal “Ack_out” as shown in Fig. 4.9; upon receiving the valid dual-rail data and an asserted acknowledge signal from the wrapper block, the asynchronous core latches the dual-rail data and then lowers the data request signal “Data_req” and clears the dual-rail control signals and address bus, starting the return-to-zero phase of the reading procedure; upon detecting a low data request signal and empty dual-rail data signals coming from the asynchronous core, the wrapper block lowers the control signals sent to the external SRAM to indicate the end of a read operation on the SRAM part and lowers the internal acknowledge signal “sram_ack” immediately without starting the counter, the lowering of the “sram_ack” signal leads to a low acknowledge signal “Ack_out” and an empty

state in the dual-rail data bus going to the asynchronous core, ending the entire handshaking process of a reading operation on the external SRAM.

When the external clock source “clk1” is adjustable, the wrapper block interface design is able to work with practically any commercial SRAM block with different access time. If the external clock source “clk1” is fixed at a certain frequency, this interface design is still able to work with a wide range of commercial SRAM blocks since the preset overflow value of the internal counter is still adjustable. For example, when the external clock source is fixed at 50MHz (clock period is 20ns), to interface with an external SRAM whose access time is 65ns, the overflow value can be set to 4 to ensure the time taken to assert the acknowledge signal is longer than the access time of the external SRAM. If another external SRAM with an access time of 130ns is used, the overflow value can be adjusted to 7.

4.6 The Interface to Internal Customized Asynchronous SRAM

As mentioned early on, a custom-designed 128x8 asynchronous SRAM block is used to serve as the 128 byte internal RAM of the proposed asynchronous 8051 microcontroller. The customized asynchronous SRAM is designed by another Master student (Cheng Xiang) and it adopts the four-phase handshake protocol

with dual-rail encoding for the input and output data buses. The symbol diagram of the customized asynchronous SRAM is shown in the figure below.

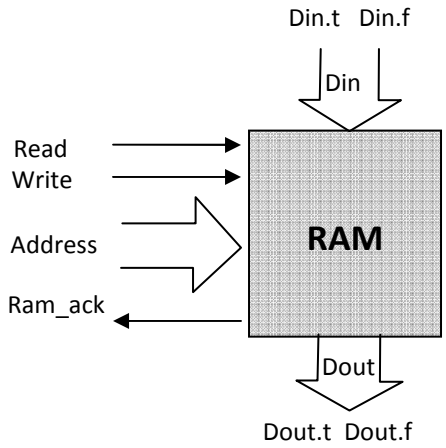


Figure 4.8: Symbol view of the customized asynchronous SRAM block

The input data bus "Din" and output data bus "Dout" are both dual-rail encoded and thus can interface with the dual-rail asynchronous core directly. The acknowledge signal "Ram_ack" is a handshake signal and can also be connected to the handshake signals of the asynchronous core directly. However, for the read signal "Read", the write signal "Write" and the address bus "Address", all of them are single-rail and thus needs to pass through a wrapper block "Ram_Wrapper" to interface with the dual-rail asynchronous core.

The operation of the customized asynchronous SRAM block is briefly elaborated in this section. For a reading operation, the sequence of actions goes in the

following order: when a logic “1” is detected on the line “Read”, the RAM block outputs the requested data onto the output data bus “Dout” according to the value of the address bus and subsequently sets the output acknowledge signal “Ram_ack” high; in the return-to-zero phase, upon detecting a logic “0” on the line “Read”, the RAM block changes the output data bus to the empty state and lowers the acknowledge line accordingly, ending the complete four-phase handshake cycle. For a writing operation on the RAM block, the sequence of actions goes in the following order: upon detecting a logic “1” on the line “Write”, the RAM block writes the data on the input data bus “Din” into the memory location specified by the address bus and sets the acknowledge line high accordingly; in the return-to-zero phase, upon detecting a logic “0” on the line “Write”, the RAM block lowers the acknowledge line and ends the handshake cycle for a write operation.

From the above elaboration, it is easy to figure out that for a reading operation the read signal should only goes to logic “1” state after the single-rail address bus contains the correct valid address. Otherwise the data stored in a wrong location may be read and sent to the asynchronous core. The same goes for the write signal in the case of a writing operation. The timing requirements for these signals are enforced within the wrapper block while the data conversion is performed. Basically, within the wrapper block, it checks for the validness of the dual-rail address bus from the asynchronous core before sending out the read or write control signals to the RAM block.

4.7 The Synchronous Peripherals

There are mainly three components in the synchronous peripherals block shown in Fig. 4.1 previously: interrupt handler, timers and serial port. All of the three sub-blocks closely follows the structure of a standard Intel synchronous 8051 design with very minimum changes to accommodate some control signals coming from the synchronous wrapper block. Since the functionality of these blocks are the same as that find in standard Intel 8051, only brief coverage on these blocks will be presented in this section.

4.7.1 The Interrupt Handler

The interrupt handler tackles interrupts coming from five different sources either within the synchronous peripherals block or from the external I/O pins. The Interrupt-Service-Routine (ISR) addresses for the five sources of possible interrupts are tabulated in the table below.

Symbol	Address	Interrupt Source
EXTI0	03H	External Interrupt 0

TIMER0	0BH	Timer 0 interrupt
EXTI1	13H	External Interrupt 1
TIMER1	1BH	Timer 1 interrupt
SINT	23H	Serial Port Interrupt

Table 4.3: Addresses of different Interrupt-Service-Routines

The interrupt handler block sends the current interrupt status to the asynchronous core, where the interrupt status is checked at the beginning of each instruction fetching stage. If an interrupt is detected, the Program Counter is branched to the corresponding ISR address shown in the table above.

The interrupt handler supports two priority levels (high and low) for the five sources of interrupts. The priority level of each interrupt source can be configured through adjusting the corresponding bit of the Interrupt Priority (IP) register located within the synchronous peripherals block. A high-level incoming interrupt can interrupt the execution of a low-level interrupt. In addition, each interrupt source can also be enabled or disabled through changing the corresponding bit of the Interrupt Enable (IE) register located within the synchronous peripherals block as well.

4.7.2 The Timers

There are two 16-bit timers built within the synchronous peripherals block, namely Timer 0 and Timer 1. Both of them can work as either a timer or a counter

depending on the selected driving clock source. When functioned as a timer (interval counting), the registers which store the current count value are incremented once per eight clock periods of the global clock driving the synchronous peripherals block. When functioned as a counter (event counting), the registers are incremented on the falling edge of the external pins T0 (for timer 0) or T1 (for timer 1). There are four different operational modes available for both timers. For modes 0, 1 and 2, Timer 0 and Timer 1 function in the same way; for mode 3, the two timers function in different ways. The selection of the driving clock source and operational mode of the two timers is controlled by the values of the TMOD register located in the SFR space.

In mode 0, the timer works as a 13-bit counter with the lower 8-bit count stored in the corresponding TL register (TL0 or TL1) and the higher 5-bit count is stored in the lower 5 bits of the corresponding TH register (TH0 or TH1). The counting process starts when the corresponding TR bit (TR0 or TR1) is set. When the timer overflows, the corresponding TF bit (TF0 or TF1) is set. Mode 1 is very similar to mode 0. Instead of working as a 13-bit counter, it works as a full 16-bit counter in mode 1 with the lower 8-bit count stored in the TL register and the higher 8-bit count stored in the TH register. In mode 2, the TL register alone functions as an 8-bit counter and the TH register holds an 8-bit preset value. Each time the TL register overflows, the preset value in the TH register is reloaded into the TL register. In mode 3, Timer 1 is not activated and the corresponding registers TL1 and TH1 hold their previous values. For Timer 0 in mode 3, the corresponding

registers TL0 and TH0 acts as two separated 8-bit counters. TL0 is associated with the usual Timer 0 control bits (TR0 and TF0) and TH0 is associated with the usual Timer 1 control bits (TR1 and TF1).

4.7.3 The Serial Port

The serial port in this design is fully duplex: it allows simultaneous data transmission and reception. Essentially the function performed by the serial port is converting output data from parallel to serial and converting input data from serial to parallel via the two external I/O pins TXD and RXD.

There are four different operation modes supported by the serial port through changing the two control bits SM0 and SM1 in the SCON register. In mode 0 (SM0 = 0 and SM1 = 0), the serial port functions as an 8-bit shift register with the baud rate fixed at $1/12^{th}$ the clock frequency driving the synchronous peripherals block. The RXD pin functions as the data line with serial data entering into and exiting the chip. The shifting clock signal is sent out via the TXD pin. Both transmission and reception starts with the least-significant bit first. In mode 1 (SM0 = 1 and SM1 = 0), the serial port functions as an 8-bit UART with variable baud rate which is supplied by Timer 1. In this mode, serial data are transmitted out via the TXD pin and received via the RXD pin. Each data frame contains 10 bits with the first bit being “0” which acts as the start bit, followed by 8 data bits and finally the last bit being “1” which acts as the stop bit. Transmission is

initiated by a write operation to the SBUF register and reception is initiated by a 1-to-0 transition on the RXD pin. In mode 2 (SM0 = 0 and SM1 = 1), the serial port functions as a 9-bit UART with fixed baud rate. Each data frame in this mode contains 11 bits: a start bit, 8 data bits, a programmable ninth data bit and a stop bit. The programmable ninth data bit can be an extra data bit or a parity bit used for simple error detection. The baud rate in this case is fixed at $1/32^{nd}$ or $1/64^{th}$ of the driving clock frequency of the synchronous peripherals block. Finally in mode 3 (SM0 = 1 and SM1 = 1), the serial port functions as a 9-bit UART with variable baud rate. This mode is very similar to the previous mode 2 except that this time the baud rate is variable. The variable baud rate is supplied by Timer 1 just like the case in mode 1.

Chapter 5

Different Structures of the Asynchronous Core

In this chapter, four different structures of the asynchronous core developed along the way in this work will be discussed and compared. The first design is a non-pipelined design with isolated Register File block where the Register File block is described in a separate module within the Balsa framework. The first design was sent out for tape-out in January 2009 using the AMS 0.35µm CMOS technology. The second design is completed after the previous tape-out and it is still a non-pipelined design but with the Register File block integrated into the main module within the Balsa framework. Through this integration process and some changes

made to better utilize the integrated Register File block at the code level within the Balsa framework, improvements in both speed and power are obtained during simulation as compared to the first design. The third design is basically a two-stage pipelined version of the second design which aims to increase the operational speed of the asynchronous core. The fourth design is modified based on the third design through taking out the interrupt checking block from the Instruction Fetch stage to form a three-stage pipelined structure. In addition, this design also incorporates the Multiplication and Division operations into the ALU block.

5.1 Design 1: Non-pipelined With Isolated Register File Block

In this design, the Register File block is located outside of the main module of the asynchronous core. At the top-level in the Balsa framework, there are two modules: a module describing the Register File block and a main module describing the rest of the asynchronous core. The two modules are synthesized separately into two Verilog gate-level netlists in the Balsa framework using the dual-rail four-phase protocol. After synthesis, all the top-level inputs and outputs are transformed into active channels (active channels are the ones that initiate a handshake sequence as mentioned in Chapter 1) by default. Since an active channel cannot communicate with another active channel, the two synthesized

netlists cannot be connected directly. Passivators (a passivator is a Handshake Component in the Balsa framework that has passive channels for all its inputs and outputs) are manually inserted in-between the two synthesized netlists for the Register File block and the main block as shown in the figure below. Active channels are indicated by a black dot while passive channels are indicated by a white dot.

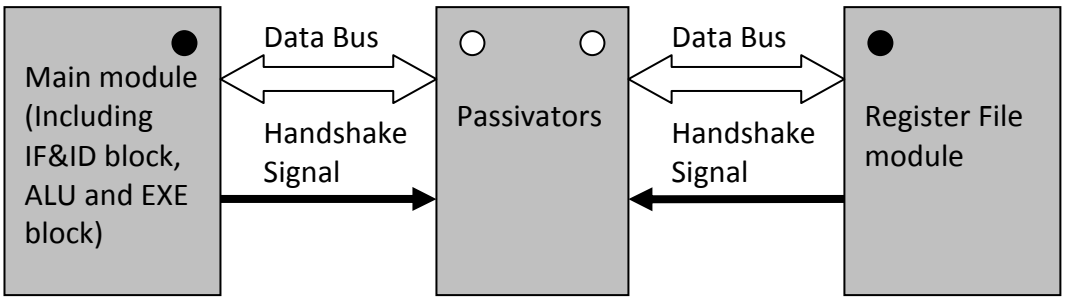


Figure 5.1: Communication between the Main block and the Register File block

In this implementation, since each handshake communication between the main module and the Register File module must go through the passivators in-between, thus it incurs a large communication delay and consumes unnecessary power while transferring data across the two blocks.

5.2 Design 2: Non-pipelined With Integrated Register File Block

This design is actually a optimized version of the first design with isolated Register File block. The Balsa description of the asynchronous 8051 core is modified to combine the two top-level modules of the first design into a single module. Essentially, the Register File block is moved into the main module of the asynchronous 8051 core in the Balsa framework. In doing so, the passivators required for the handshake communications in the previous design can be discarded away since there is only one gate-level netlist generated after synthesis in the Balsa framework.

Another advantage offered by combining the two modules is associated with the easier access to some frequently used registers that belongs to the SFR space. Registers such as the ACC and the PSW are read each time when an instruction is fetched. The time reduction in accessing these registers directly leads to the time reduction in the instruction fetch stage. In the previous design structure, whenever the instruction fetch and decode block needs to access the ACC and PSW registers, it needs to send the address of the corresponding registers along with some control signals to the Register File block via the passivators in-between. The Register File block decodes the address received and sends back the requested data stored in those locations to the main module via the passivators again. Therefore the whole process incurs large communication and decoding delay. On the other hand, for this design structure with integrated Register File block, the Register File lies within the main module. Therefore whenever the instruction

fetch and decode block needs to read data stored in the ACC and the PSW registers, it can be done through direct assignment without incurring long communication and decoding delay. In addition, since it cuts down on the logics involved in the long communication path for the previous design, reduction in power consumption can also be expected for this integrated Register File block approach.

5.3 Design 3: Two-Stage Pipelined Design

This third design structure is modified based on the second design with integrated Register File block by pipelining the Instruction Fetch and Decode stage with the Instruction Execution stage to achieve a faster operational speed at the expense of an increase in power consumption. Both of the two stages are slightly modified at the code level to better balancing the operational time required for the two stages. However, since the instruction set of the Intel 8051 is highly irregular, it is almost impossible to balance the two stages for all instructions. There are cases that the time taken by the instruction fetch and decode stage is significantly longer than that taken by the instruction execution stage, leading to the time taken for the two-stage pipeline design is almost the same as the that of the second design where the two stages executes sequentially. A typical example for this scenario is the "LJMP" (long jump) instruction. In the instruction fetch and decode stage, threes byte of instruction data are fetched sequentially from the Program ROM which

takes a rather long time. Whereas in the instruction execution stage, only a simple assignment to the Program Counter register is required which takes very little time as compared to the first stage. Also, there are cases that the instruction execution stage is actually the dominating stage and leading to only slight improvement in operational speed too as compared to the second design. A typical example for this scenario is the "ADD A, @Ri" instruction. In the instruction fetch and decode stage, there is only one instruction byte needs to be fetched from the Program ROM. However in the instruction execution stage, two sequential data fetching from the Register File block and one addition operation are required in total, consuming significantly longer time as compared to the first stage. Thus the time taken for the parallel execution of the two stages is actually comparable to the time taken for the sequential execution of the two stages for such kind of instructions.

In general, the time reduction harvested from the two-stage pipelined design is only significant for cases that have comparable execution time for both stages. Even in such cases, the time reduction harvested is still smaller than 50% due to some extra logics added to enable the parallel execution.

5.4 Design 4: Three-Stage Pipelined Design With MUL and DIV

In design 4, there are mainly two changes made within the Balsa framework as compared to design 3. The first change is the addition of the Multiplication and Division operations for the ALU. The Multiplication and Division blocks are elaborated previously in Chapter 4. Another change is taking the interrupt checking block (mentioned earlier in Chapter 4) out of the Instruction Fetch and Decode block to form a standalone block which executes in parallel with the other two blocks. The interrupt checking block is mainly responsible for checking the current interrupt status coming from the synchronous peripherals block. Since the interrupt status coming from the synchronous peripherals is only updated on the rising edge of the clock signal, the asynchronous core may have to wait for as long as one clock period for the valid incoming interrupt status data. This waiting time may significantly slow down the operational speed of the asynchronous core when the synchronous peripherals' clock period is comparable to the circuit running time of the asynchronous core. Consequently, the running speed of the asynchronous core may be more inclined to be limited by the Instruction Fetch and Decode stage. By taking out the interrupt checking block, it helps to improve the operational speed of instructions that are originally limited by the Instruction Fetch and Decode stage.

Outside of the Balsa framework, some optimizations are also performed on the synthesized gate-level netlist within the Design Compiler to fix the heavy loading effect and long transition time of some interconnect nets via the insertion of

balanced buffer trees. The operational speed of the asynchronous 8051 core significantly increases after the insertion of balanced buffer trees.

After performing these optimizations within and outside of the Balsa framework, design 4 is capable of running at a significantly faster speed as compared to design 3 even after the inclusion of the Multiplication and Division blocks as shown by the Nanosim post-layout simulation results in the next chapter.

Chapter 6

Simulation Results of Different Asynchronous 8051 Core Designs

In this chapter, the simulation results of the four different designs of the asynchronous 8051 core mentioned in Chapter 5 will be presented and discussed. All the simulation data presented in this chapter are obtained via performing post-layout transistor-level simulation on the corresponding spice netlists of the asynchronous 8051 core in Nanosim. The spice netlists are created from the extracted versions of the circuit layouts which are obtained after performing P&R

in SOC Encounter. There are basically two main transistor-level blocks inside the spice netlist: the asynchronous 8051 microcontroller designed and an external program ROM which stores the testing program.

6.1 Common Simulation Settings

Before presenting the detailed simulation data, some common simulation settings adopted are elaborated in this section. For each design, simulations are performed at five different voltage levels: 3.3V (nominal supply voltage for AMS 0.35μm CMOS technology), 2.0V, 1.5V, 1.1V and 1.0V, in order to cover the potential supply voltage that ranges from 1.0V to 3.3V for sensor interface applications. Meanwhile, it also demonstrates the robustness of the dual-rail four-phase procotol towards variations in supply voltage. For simplicity, the frequency of the driving clock of the synchronous peripherals is fixed at 5MHz regardless of the supply voltage used. The frequency of the driving clock of the blocks that interface with external memory varies with the supply voltage used as shown by the table below while the overflow values of the internal counters located within the interface blocks are fixed at 2. The variation in the clock frequency is required to accommodate the variation in the access time of the program ROM under different supply voltages.

Table 6.1: Common simulation settings

Voltage	Frequency
3.3V	100MHz
2.0V	80MHz
1.5V	50MHz
1.1V	20MHz
1.0V	20MHz

Five performance indicating parameters are tabulated for each design at a particular supply voltage. The first parameter represents the time to complete one instruction and it is denoted by *"T(instruction)"* in the tables. The second parameter is called Million Instruction Per Second (*MIPS*) which represents the number of instruction that can be completed within one second. The third parameter is the total current consumption of the asynchronous 8051 microcontroller denoted by *"I_total"* which includes the current consumptions of the asynchronous core and the synchronous peripherals. The fourth parameter represents the current consumption of the asynchronous core alone and it is denoted by *"I_asyncore"* in the tables. The last parameter presents the energy consumption of the asynchronous core to complete one instruction and it is denoted by "*Energy/Instrn*" in the tables.

For typical synchronous 8051 design, it is relatively easier to measure the MIPS value since most of the instructions (except multiplication and division) finishes in one machine cycle. The length of a machine cycle may vary with the design

structure adopted (for a typical Intel 8051 microcontroller, one machine cycle consists of twelve clock cycles of the driving oscillator). For example, for a typical Intel 8051 microcontroller driven by a 12MHz oscillator, the time taken to complete any instruction except MUL and DIV is fixed at 1us, which means the chip is running at 1 MIPS. However, for asynchronous 8051 design, the time taken to complete one instruction is dependent on the instruction itself. It is possible that each instruction runs at different speeds due to the absence of the notion of a machine cycle. Therefore it is very hard to derive an accurate number for the MIPS value since a different piece of program code can generate a different MIPS value. In this work, for simplicity as well as simulation time considerations, the MIPS value is calculated as the average of seven different instructions that falls into the three major instruction types: arithmetic instructions (ADD_1 and ADD_4), data transfer instructions (MOV_1 and MOV_4) and logical instructions (ORL_1, ORL_4 and SWAP). Instructions ADD_4, MOV_4 and ORL_4 have two instruction bytes while the rest have one instruction byte. Each of the seven instructions is repeatedly executed for several times to obtain a more accurate average value for the parameter *"T(instruction)"*.

6.2 Simulation Results of Design 1

The Nanosim simulation data of the first design is summarized in the table below. As mentioned in Chapter 5, for design 1, the Register File block is a standalone module within the Balsa framework and its communication with the main module of the asynchronous core has to go through some passivators. Consequently its performance is the worst among the four.

Table 6.2: Post-layout transistor-level Nanosim simulation results of Design 1

Design 1								
3.3V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	0.80	0.93	0.80	0.80	0.80	0.80	0.60	0.79
MIPS	1.25	1.08	1.25	1.25	1.25	1.25	1.67	1.28
I_total/uA	2500	2100	2150	1990	2240	2100	2370	2207
I_asyncore/mA	2119	1719	1769	1609	1859	1719	1989	1826
Energy(pJ)/Instrn	5594	5276	4670	4248	4908	4538	3938	4739
2.0V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	1.40	1.40	1.20	1.20	1.20	1.20	1.00	1.23
MIPS	0.71	0.71	0.83	0.83	0.83	0.83	1.00	0.82
I_total/uA	837	799	791	764	870	809	847	817
I_asyncore/uA	650	612	604	577	683	622	660	630
Energy(pJ)/Instrn	1820	1714	1450	1385	1639	1493	1320	1547
1.5V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	2.00	2.20	1.80	1.80	1.80	1.80	1.40	1.83
MIPS	0.50	0.45	0.56	0.56	0.56	0.56	0.71	0.56
I_total/uA	443	415	438	413	425	423	847	486
I_asyncore/uA	329	301	324	299	311	309	733	372
Energy(pJ)/Instrn	987	993	875	807	840	834	1539	982
1.1V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	4.92	5.20	4.00	3.90	4.20	4.40	2.80	4.20
MIPS	0.20	0.19	0.25	0.26	0.24	0.23	0.36	0.25
I_total/uA	170	161	179	177	175	171	189	175
I_asyncore/uA	102	93	111	109	107	103	121	107
Energy(pJ)/Instrn	552	532	488	468	494	499	373	487

1.0V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	7.40	7.80	6.00	5.80	6.40	6.40	4.65	6.35
MIPS	0.14	0.13	0.17	0.17	0.16	0.16	0.22	0.16
I_total/uA	123	118	125	119	122	119	135	123
I_asyncore/uA	63	58	65	59	62	59	75	63
Energy(pJ)/Instrn	466	452	390	342	397	378	349	396

Observing the "T(instruction)" data at 3.3V, it may seem surprising that for instructions ADD_1, MOV_1, MOV_4, ORL_1 and ORL_4 all of them have the same value of 0.8us. The cause of this phenomenon lies in the interrupt checking step, where the asynchronous core has to wait for the interrupt status data from the synchronous peripherals that arrive only at the rising edge of the clock signal. The figure below illustrates the case for a two-byte long instruction. The "Syn_CLK" signal is the driving clock signal for the synchronous peripherals and it runs at 5MHz. The "Read" signal is sent out by the asynchronous core to the external Program ROM to fetch the instruction bytes. When repeatedly executing the same instruction, the "Read" signal is periodical and its period corresponds to the instruction execution time "T(instruction)". As shown in the figure, the total time can be divided into two parts: time taken for instruction fetch & execution step and time taken for interrupt checking step (the blue interval is actually the time taken by the interrupt checking step of the following instruction, but it is the same as that of the current instruction). As long as the finishing time of the instruction fetch & execution step passes the third rising edge of the "Syn_CLK" signal after the first "Read" pulse, the interrupt checking step has to wait for the interrupt status data that arrives earliest at the next rising edge of the "Syn_CLK"

signal. Therefore even though different instructions may have varying length for the instruction fetch & execution interval, they may still end up have the same "T(instruction)" value due to different waiting time in the interrupt checking step.

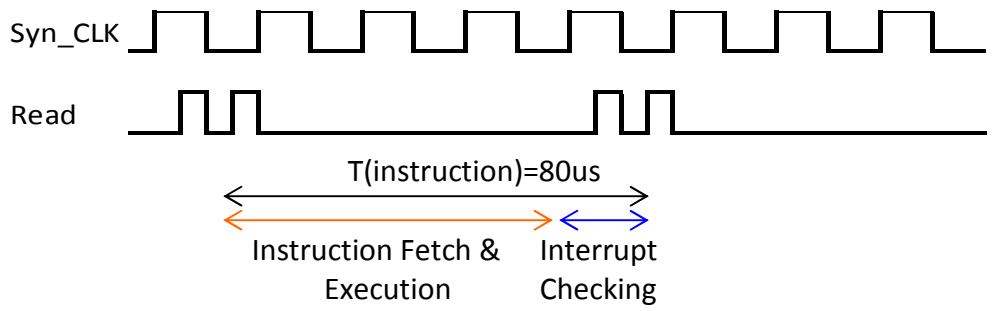


Figure 6.1: One instruction cycle of a two-byte instruction of Design 1 at 3.3V

In short, for this design, the asynchronous core is running at quite slow speed (1.28 MIPS at 3.3V and 0.16 MIPS at 1.0V) and the energy consumed per instruction at 1.0V is about 396pJ/Instrn.

This design went for tape out in January 2009 using the AMS 0.35µm CMOS technology. The die photo of the chip manufactured is shown in Fig. 6.2 below. The measurement results are in good agreement with the Nanosim simulation results. The MIPS values captured during testing are practically the same as those obtained in Nanosim post-layout simulation. For the power consumption, the measurement results are about 10% smaller than that obtained during simulation in average. Due to the time limitation at that time, this version (Design 1) is not optimized and its performance is not as good as those later versions.

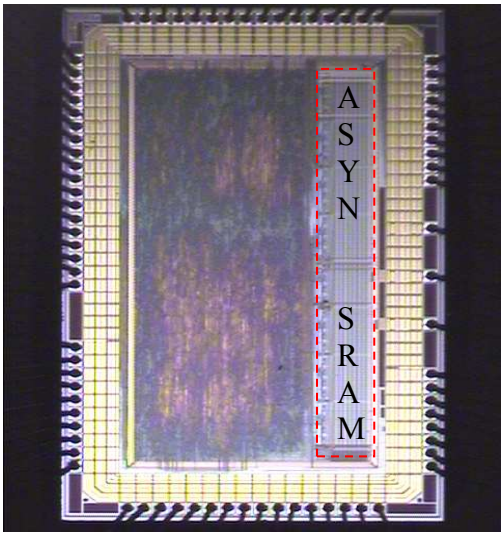


Figure 6.2. Die photo of the asynchronous 8051 microcontroller (Design 1)

6.3 Simulation Results of Design 2

The Nanosim simulation data of the second design is summarized in the table below. In design 2, the Register File block is integrated into the main module.

Table 6.3: Post-layout transistor-level Nanosim simulation results of Design 2

Design 2								
3.3V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	0.60	0.60	0.40	0.40	0.60	0.50	0.40	0.50
MIPS	1.67	1.67	2.50	2.50	1.67	2.00	2.50	2.07
I_total/uA	1500	1510	1660	1620	1360	1570	1370	1513
I_asyncore/mA	1212	1222	1372	1332	1072	1282	1082	1225
Energy(pJ)/Instrn	2400	2420	1811	1758	2123	2115	1428	2008

2.0V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	1.00	1.00	0.80	0.80	0.80	0.80	0.60	0.83
MIPS	1.00	1.00	1.25	1.25	1.25	1.25	1.67	1.24
I_total/uA	536	560	518	530	611	590	565	559
I_asyncore/uA	388	412	370	382	463	442	417	411
Energy(pJ)/Instrn	776	824	592	611	741	707	500	679
1.5V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	1.50	1.60	1.20	1.20	1.40	1.30	1.00	1.31
MIPS	0.67	0.63	0.83	0.83	0.71	0.77	1.00	0.78
I_total/uA	280	273	279	268	275	278	251	272
I_asyncore/uA	185	178	184	173	180	183	156	177
Energy(pJ)/Instrn	416	427	331	311	378	357	234	351
1.1V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	3.60	3.40	2.95	2.60	3.20	3.00	2.20	2.99
MIPS	0.28	0.29	0.34	0.38	0.31	0.33	0.45	0.34
I_total/uA	123	127	114	117	118	118	114	119
I_asyncore/uA	62	66	53	56	57	57	53	58
Energy(pJ)/Instrn	246	247	172	160	201	188	128	192
1.0V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	5.40	5.20	4.40	4.00	4.80	4.40	3.20	4.49
MIPS	0.19	0.19	0.23	0.25	0.21	0.23	0.31	0.23
I_total/uA	93	93	86	90	90	93	85	90
I_asyncore/uA	39	39	32	36	36	39	31	36
Energy(pJ)/Instrn	211	203	141	144	173	172	99	163

From the simulation data above, it is rather obvious that design 2 out-performs design 1 both in terms of speed and power consumption. The energy consumed per instruction at 1.0V supply is about 163pJ for this design.

6.4 Simulation Results of Design 3

The Nanosim simulation data of design 3 is summarized in the table below. For this design, the IF & ID unit runs in parallel with the EXE unit to form a two-stage pipeline structure. Due to the difficulties involved in balancing the time spent for the two stages, the improvement in operational speed is not very significant.

Table 6.4: Post-layout transistor-level Nanosim simulation results of Design 3

Design 3								
3.3V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	0.40	0.40	0.40	0.40	0.40	0.40	0.40	0.40
MIPS	2.50	2.50	2.50	2.50	2.50	2.50	2.50	2.50
I_total/uA	2420	2560	1950	2020	2250	2290	1960	2207
I_asyncore/mA	2109	2249	1639	1709	1939	1979	1649	1896
Energy(pJ)/Instrn	2784	2969	2163	2256	2559	2612	2177	2503
2.0V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	0.60	0.80	0.60	0.80	0.60	0.67	0.60	0.67
MIPS	1.67	1.25	1.67	1.25	1.67	1.49	1.67	1.52
I_total/uA	894	745	737	599	843	793	664	754
I_asyncore/uA	736	587	579	441	685	635	506	596
Energy(pJ)/Instrn	883	939	695	706	822	851	607	786
1.5V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	0.80	1.20	0.80	1.20	0.80	1.20	0.80	0.97
MIPS	1.25	0.83	1.25	0.83	1.25	0.83	1.25	1.07
I_total/uA	470	367	410	300	460	326	355	384
I_asyncore/uA	371	268	311	201	361	227	256	285
Energy(pJ)/Instrn	445	482	373	362	433	409	307	402

1.1V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	2.10	2.60	1.80	2.60	1.85	2.60	1.80	2.19
MIPS	0.48	0.38	0.56	0.38	0.54	0.38	0.56	0.47
I_total/uA	170	145	155	124	171	136	137	148
I_asyncore/uA	107	82	92	61	108	73	74	85
Energy(pJ)/Instrn	247	235	182	174	220	209	147	202
1.0V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	3.27	3.80	2.80	4.00	2.85	3.80	2.80	3.33
MIPS	0.31	0.26	0.36	0.25	0.35	0.26	0.36	0.31
I_total/uA	120	109	115	93	119	101	100	108
I_asyncore/uA	65	54	60	38	64	46	45	53
Energy(pJ)/Instrn	213	205	168	152	182	175	126	174

Design 3 runs at 2.5 MIPS at 3.3V supply and 0.31 MIPS at 1.0V supply. The energy consumed per instruction is about 174pJ at 1.0V supply.

6.5 Simulation Results of Design 4

The Nanosim simulation data of the third design is summarized in Table 6.5. In design 4, the Interrupt Checking unit is taken out of the IF&ID unit to form an independent stage. In addition, balanced buffer trees are inserted into the synthesized gate-level netlist using Design Compiler. The insertion of balanced buffer trees greatly helps to reduce the delay in some heavily loaded nets and improves the operational speed of the asynchronous core.

Table 6.5: Post-layout transistor-level Nanosim simulation results of Design 4

Design 4								
3.3V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	0.24	0.26	0.19	0.26	0.22	0.25	0.16	0.23
MIPS	4.17	3.85	5.41	3.85	4.55	4.00	6.25	4.58
I_total/uA	3610	3540	3370	2530	3470	3080	3290	3270
I_asyncore/mA	3269	3199	3029	2189	3129	2739	2949	2929
Energy(pJ)/Instrn	2589	2745	1849	1878	2272	2260	1557	2164
2.0V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	0.43	0.44	0.33	0.44	0.39	0.41	0.28	0.39
MIPS	2.33	2.27	3.03	2.27	2.56	2.44	3.57	2.64
I_total/uA	1135	1123	1114	823	1140	1017	1095	1064
I_asyncore/uA	960	948	939	648	965	842	920	889
Energy(pJ)/Instrn	826	834	620	570	753	690	515	687
1.5V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	0.75	0.75	0.56	0.75	0.68	0.70	0.49	0.67
MIPS	1.33	1.33	1.79	1.33	1.47	1.43	2.04	1.53
I_total/uA	500	507	482	370	500	460	492	473
I_asyncore/uA	391	398	373	261	391	351	383	364
Energy(pJ)/Instrn	440	448	313	294	399	369	282	363
1.1V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	1.90	1.81	1.45	1.86	1.73	1.70	1.19	1.66
MIPS	0.53	0.55	0.69	0.54	0.58	0.59	0.84	0.62
I_total/uA	183	193	175	148	177	176	182	176
I_asyncore/uA	118	128	110	83	112	111	117	111
Energy(pJ)/Instrn	247	255	175	170	213	208	153	204
1.0V	ADD_1	ADD_4	MOV_1	MOV_4	ORL_1	ORL_4	SWAP	Average
T(instruction)/us	2.91	2.61	2.22	2.67	2.66	2.45	1.72	2.46
MIPS	0.34	0.38	0.45	0.37	0.38	0.41	0.58	0.42
I_total/uA	124	135	121	110	124	124	126	123
I_asyncore/uA	68	79	65	54	68	68	70	67
Energy(pJ)/Instrn	198	206	144	144	181	167	120	166

Design 4 runs at 4.58 MIPS at 3.3V supply and 0.42 MIPS at 1.0V supply. The energy consumed per instruction is about 166pJ at 1.0V supply.

6.6 Comparison Between the Simulation Results of the Four Designs

The Nanosim post-layout simulation results of the previously mentioned four different designs of the asynchronous 8051 core is summarized in the table below for comparison.

Table 6.6: Comparison between the four asynchronous core structures

Comparison Between the Four Designs of the Asynchronous 8051 Core												
	Design 1			Design 2			Design 3			Design 4		
	MIPS	uW	pJ/ instrn	MIPS	uW	pJ/ instrn	MIPS	uW	pJ/ instrn	MIPS	uW	pJ/ instrn
3.3V	1.3	6026	4739	2.1	4042	2008	2.5	6257	2503	4.6	9666	2164
2.0V	0.82	1260	1547	1.2	822	679	1.5	1192	786	2.6	1778	687
1.5V	0.56	558	982	0.78	266	351	1.1	428	402	1.5	546	363
1.1V	0.25	118	487	0.34	64	192	0.47	94	202	0.62	121	203
1.0V	0.16	63	396	0.22	36	163	0.31	53	174	0.42	67	166

Comparing design 2 with design 1, design 2 out-performs design 1 both in terms of power and speed although they are both non-pipelined designs. The reason for

this phenomenon lies in the integration of the Register File block into the main module within the Balsa framework as explained previously in Chapter 5.

Comparing design 3 with design 2, design 3 runs faster than design 2 at the expense of higher power consumption due to its two-stage pipeline structure. The increase in operational speed (MIPS value) is about 41% at 1.0V supply. The energy consumed per instruction for design 3 is slightly higher than that of design 2 at the same supply voltage.

Comparing design 4 with design 2, design 4 runs much faster than design 2 at the expense of higher power consumption due to its three-stage pipeline structure and the insertion of balanced buffer trees. The increase in operational speed (MIPS value) is about 91% at 1.0V supply. The energy consumed per instruction is almost the same as that of design 2 at the same supply voltage.

In short, design 1 with an isolated Register File block is not a satisfactory design. Its performance is the worst among the four. Design 2 has the lowest power consumption and energy per instruction value although the operational speed is slower as compared to design 3 and 4. Design 4 has the fastest operational speed and a comparable energy per instruction value as compared to design 2.

6.7 Comparison with Other Existing Designs

The performance comparison between design 2, 3 and 4 with two other existing designs are summarized in the table below at 1.1V supply and 0.35µm technology

Table 6.7: Comparison with other existing designs at 1.1V 0.35µm

Comparison with Existing Designs @ 1.1V 0.35µm				
Design	**MIPS**	**mW**	**pJ/Instrn**	**$Et^2(Js^2)$**
Sync80C51*[4]	1.3	1.48	1100	6.51×10^{-22}
A8051#[6]	0.6	0.07	130	4.04×10^{-22}
Design2	0.34	0.064	192	1.66×10^{-21}
Design3	0.47	0.094	202	9.14×10^{-22}
Design4	0.62	0.121	203	5.28×10^{-22}

*The performance of this design is scaled from 3.3V.
#This is a 2-stage pipelined design using the four-phase bundled data protocol in Balsa

The first design "Sync8051" is an optimized synchronous version of the 8051 microcontroller designed by the Philips Lab. Its performance is scaled down from 3.3V supply. The second design "A8051" [6] is a two-stage pipelined asynchronous 8051 designed using the four-phase bundled-data approach in the Balsa framework. Comparing to our four-phase dual-rail designs (design 2, 3 and 4), the bundled-data design "A8051" is expected to run faster and consume less

power for reasons that have been mentioned in Chapter 1. In short, the optimized synchronous version "Sync8051" has the fastest operational speed at the expense of fairly large power consumption. Its energy consumed per instruction is also the largest. The "A8051" has the lowest energy consumption per instruction due to the four-phase bundled data approach used. However it is only able to work in a very narrow range of supply voltage around 1.1V since its proper functioning relies on the delay matching in the request signal. Our designs although have a slightly higher energy per instruction value as compared to "A8051", they are capable of functioning correctly in a much wider range of supply voltage due to the quasi-delay-insensitive nature of the four-phase dual-rail protocol used.

The table below shows the comparison between our designs with two other existing designs at 1.1V supply and 0.18µm technology.

Table 6.8: Comparison with other existing designs at 1.1V 0.18µm

Comparison with Existing Designs @ 1.1V 0.18µm				
Design	**MIPS**	**mW**	**pJ/Instrn**	**$Et^2(Js^2)$**
Lutonium[2]	100	20.7	207	2.06×10^{-26}
A8051*#[6]	5	0.114	23	9.11×10^{-25}
Design2*	2.8	0.104	37	4.72×10^{-24}
Design3*	3.9	0.153	39	2.56×10^{-24}
Design4*	5.2	0.197	38	1.40×10^{-24}

*These designs are scaled down from 0.35µm technology.
#This is a 2-stage pipelined design using the four-phase bundled data protocol in Balsa

The first design “Lutonium” [2] is a highly pipelined asynchronous 8051 microcontroller from Caltch and it is designed according to the four-phase dual-rail protocol using the CaSCADE tool set. “Lutonium” is capable of running at very high speed at the expense of high power consumption. Comparing our designs with the “Lutonium”, our designs consume much less energy per instruction though running at a much slower speed.

Chapter 7

Conclusion

In this thesis, the design of a low-power voltage-scalable asynchronous 8051 microcontroller with interface to external commercial memory is presented. The asynchronous core of the proposed design is synthesized in the Balsa framework using the dual-rail four-phase approach in order to achieve good robustness towards variations in supply voltage and fabrication process.

The main difference of the proposed asynchronous 8051 as compared to the synchronous Intel 8051 lies in the presence of some interface wrapper blocks which are located around the asynchronous core. The main purpose of these wrapper blocks is to control the data movement into and out of the dual-rail

asynchronous core. The design of the asynchronous core is elaborated and the constituting blocks are discussed in details in Chapter 4. The novel interface block design to external commercial memory can work with any commercial memory in general after proper configuration.

Four different versions of the asynchronous core developed in the course of this work are presented. Design 1 is the first version and its performance is less satisfactory as compared to the later three designs. Design 2 has the lowest energy consumption among the four and it consumes about 163pJ per instruction while running at 0.22MIPS at 1.0V supply. Design 3 is a two stage-pipelined version of Design 2 and it trades power consumption for speed improvement. The latest version, Design 4, is a three-stage pipelined design and it achieves the best overall performance, consuming about 166pJ per instruction while running at 0.42MIPS at 1.0V.

Currently, the Balsa framework is unable to optimize the drive strength of the gate cells while synthesizing the gate-level netlist. Applying more optimization techniques on the gate-level netlist generated by the Balsa framework to improvement the performance may be one area of future work.

Bibliography

[1]. J.Sparsø and S.Furber (eds), *Principles of asynchronous circuit design – A system perspecticve*, Kluwer Academic Publishers, 2001 (ISBN 0-7923-7613-7).

[2]. A. J. Martin*, et al.*, "The design of an asynchronous MIPS R3000 microprocessor," in *Conference on Advanced Research in VLSI*, pp. 164-181, 1997.

[3]. J. V. Woods, P. Day, S. B. Furber,J. D. Garside,N. C. Paver, S. Temple, "AMULET1: an asynchronous ARM microprocessor," IEEE Trans. Computers, Vol. 46, April 1997, pp. 385-398.

[4]. H.V. Gageldonk, K.V. Berkel, A. Peeters, "An Asynchronous low-power 80C51 microcontroller," in Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems, 30 March-2 April 1998, pp. 96-107

[5]. A. J. Martin, *et al.*, "The Lutonium: a sub-nanojoule asynchronous 8051 microcontroller," in *International Symposium on Asynchronous Circuits and Systems*, pp. 14-23, 2003.

[6]. Kok-Leong Chang and Bah-Hwee Gwee, "A low-energy low-voltage asynchronous 8051 microcontroller core" in *International Symposium on Circuits and Systems*, pp. 3181-3184, 2006.

[7]. Alain J. Martin, Mika Nystr¨om, and Paul I. P′enzes. "Et^2: A Metric for Time and Energy Efficiency of Computation." In *Power-Aware Computing,* R. Melhem and R. Graybill, eds. Boston, Mass.: Kluwer Academic Publishers, 2002.

[8]. D.A. Edwards and W.B. Toms, "The Status of Asynchronous Design in Industry", Information Society Technologies (IST) Programme, Concerted Action Thematic Network Contract. IST-1999-29119.

[9]. Handshake Solutions: www.handshakesolutions.com

[10]. Theseus Logic: www.theseus.com

[11]. Chao Xue, Xiang Cheng, Yang Guo, Yong Lian, "The Design of a Sub-Nanojoule Asynchronous 8051 with Interface to External Commercial Memory", ASICON, p427-430, 2009.

[12]. D. Adwards, A. Bardsley, L. Janin and W. Toms, Balsa: *A Tutorial Guide*, version 3.4.2, Jan 2005.

[13]. A. Bardsley, *Implementing Balsa Handshake Circuits*, PhD thesis, Department of Computer Science, University of Manchester, 2000.

[14]. Kessels, J. Peeters, A. Philips Res. Lab., "The Tangram framework: asynchronous circuits for low-power", Proceedings of the ASP-DAC, p 255-260, 2001.

[15]. CaSCADE tool set: http://www.cs.columbia.edu/~nowick/asynctools/

[16]. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[17]. E.W.Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs", Communications of the ACM, volume 18 issue 8, p 453-457, 1975.

[18]. I. Scott MacKenzie and Raphael C.-W.Phan, *The 8051 Microcontroller*, Upper Saddle River, NJ : Pearson Prentice Hall, c2007.

[19]. Intel 8051 pin configurations: http://www.cpu-world.com/info/Pinouts/8051.html

[20]. Intel 8051 structure : http://en.wikipedia.org/wiki/8051