5-2017

Power Efficient High Temperature Asynchronous Microcontroller Design

Nathan William Kuhns
University of Arkansas, Fayetteville

Power Efficient High Temperature Asynchronous Microcontroller Design

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Doctor of Philosophy in Engineering

by

Nathan Kuhns
University of Arkansas
Bachelor of Science in Computer Engineering, 2011

May 2017
University of Arkansas

This dissertation is approved for recommendation to the Graduate Council.

Dr. Jia Di
Dissertation Director

Dr. Pat Parkerson
Committee Member

Dr. Jingxian Wu
Committee Member

Dr. Dale Thompson
Committee Member

Abstract

There is an increasing demand for dependable and efficient digital circuitry capable of operating in high temperature environments. Extreme temperatures have adverse effects on traditional silicon synchronous systems because of the changes in delay and setup and hold times caused by the variances in each device's threshold voltage. This dissertation focuses on the design of the major functionality of an asynchronous 8051 microcontroller in Raytheon's high temperature Silicon Carbide process, rated for operation over 300ºC. The microcontroller is designed in NULL Convention Logic, for which the traditional bus architecture used for data transfer would consume a large amount of power. To make the design more power efficient, the bus architecture has been replaced with a more complex yet efficient MUX-based data transfer scheme. This change in the design architecture also allows for improved internal data transfer rates leading to an increase in overall circuit performance. Simulation results show that the designed Silicon Carbide microcontroller framework successful executes 8051 ISA instructions. Results from the MUX-based architecture show an overall decrease in power consumption of over two orders of magnitude when compared with its bus architecture counterpart. Also, the increased internal data transfer rates resulted in an overall performance improvement of 22.8%.

Acknowledgements

To Dr. Jia Di, thank you for your guidance, patience, and wisdom during the years I have spent under your leadership. Thanks to you, I have had many opportunities to learn and grow as a professional in our field. To my fellow colleagues at the University of Arkansas, thank you for your comradery in our studies and research. To my family and friends, your unwavering support has helped me overcome many challenges along the way.

Table of Contents

List of Tables

List of Figures

I. Introduction

A. Problem Statement

There is an increasing demand for reliable high temperature digital circuitry in many fields today, such as automobiles, commercial drilling, aerospace applications, and power electronics. In these fields, engineers are faced with the challenge of maintaining a suitable environment for integrated circuits (ICs). This is typically done by either routing signals across long distances away from the heat source or by implementing costly climate control spaces. These solutions significantly increase size, weight, and power (SWaP) demands and/or decrease the overall performance and reliability of the system, which would be avoided if the ICs themselves could operate in extreme temperatures. Along with extreme temperatures, large temperature swings also cause timing issues in traditional synchronous digital systems due to changes in electron mobility as temperatures change. These changes affect threshold voltages of each device which could lead to violations in setup and hold times, ultimately resulting in circuit failure.

In recent years, Silicon Carbide (SiC) has become an appealing alternative to Silicon (Si) for use as a semiconductor device material. SiC shares the same molecular structure as Si making it a suitable replacement but has a wider band-gap, which innately gives SiC more resiliency in high temperature environments. SiC components have proven to function reliably at temperatures up to 500 ºC over the span of 1,000 hours [1]. SiC processes are still a relatively new technology and therefore are still under active development. When compared to their Si counterparts, SiC processes exhibit larger device sizes and supply voltages as well as minimal routing layers. There is also a significant decrease in overall performance, and an increase in device-to-device variation which shifts device threshold voltages even further. Despite these shortcomings, research continues to improve SiC circuit technology for viable use in commercial applications. Developed by Raytheon, the high temperature silicon carbide (HTSIC) process offers an assortment of features that make digital IC design in SiC possible. HTSIC is a 1.2µm CMOS process rated for stable operation over 350 ºC [2]. It exhibits a 12-15V nominal voltage, one metal and two poly layers for routing, N-type substrate, and utilizes a 4H polytype molecular structure. HTSIC is the first CMOS SiC process with a high-fidelity process design kit (PDK) [3], and is chosen for the implementation of this work.

Asynchronous circuits use handshaking signals instead of a global clock signal to control logic flow throughout a design, making them impervious to the timing problems caused by large temperature swings in their synchronous counterparts. NULL Convention Logic (NCL) is a quasi-delay insensitive digital logic paradigm introduced in 1996 [4], and has been successfully implemented in SiC in the past [5], [6]. NCL is a "correct-by-construction" logic architecture, meaning if all individual devices remain fully functional then the design as a whole will continue to function properly regardless of timing, making NCL extremely robust in high-temperature applications.

In the early 1970's, Gary Boone of Texas Instruments (TI) noticed several commonalities between many ongoing projects that could be performed by a single multi-purpose IC rather than several different specialized circuits. Boone's idea gave birth to the first ever microcontroller, called the TMS1802NC, and was released in industry in 1974 [7]. The TMS1802NC contained 5,000 transistors, 128 bits of data memory and 3,000 bits of program memory. It was the first IC with the flexibility to perform any number of unique tasks, and within the next decade nearly 100 million units were sold worldwide. Today, microcontroller designs are still in use and have undergone many improvements and alterations to encompass an even larger area of functionality. In 1980, Intel released the 8051 microcontroller which is regarded as one of the most popular designs and is still in popular use today. The straightforward modular design of the 8051 gave way to several variations of the design to be released, including the 8031 variant which uses only external program memory instead of an internal/external split memory structure. The flexibility of this microcontroller makes it a highly desirable design for a wide range of applications and is the reason it was chosen for this work.

B. Dissertation Statement

The goal of this dissertation is to design a NCL SiC microcontroller instruction block modeled after the Intel 8031, and will represent the first asynchronous microcontroller component design in SiC. The common bus structure used in synchronous 8031/8051 for data delivery will be replaced with a MUX-based data delivery structure to reduce power consumption at the cost of the area required to implement the bus replacement system and the control logic necessary for its operation. Also, several performance

improvements implemented were made possible by the removal of the bus due to the added ability to transfer multiple sets of data simultaneously.

C. Dissertation Organization

Chapter 2 provides background information for the purpose of gaining a better understanding of the integral components and theory behind this work: SiC, NCL and the 8051 microcontroller. Chapter 3 gives insight into the approach taken to design this circuit and the challenges faced during each phase of the project. Chapter 4 presents performance and power consumption transistor level simulations results and analysis. Chapter 5 summarizes the findings based on the simulation results and discusses future possibilities of this work.

II. Background

A. Silicon Carbide (SiC)

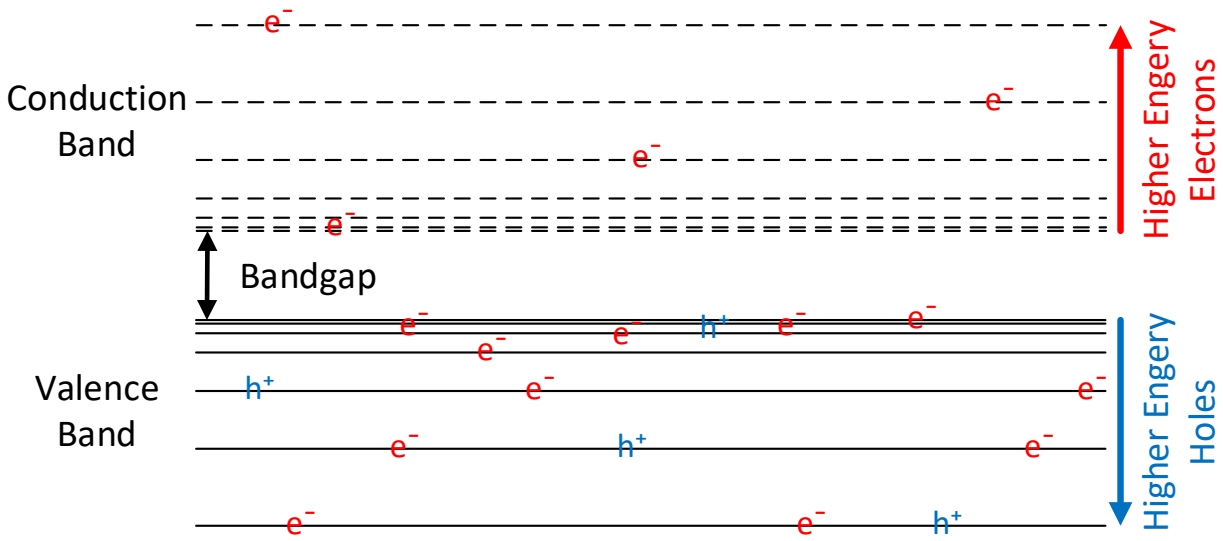

Figure 1. Bandgap Illustration [6]

SiC is a *wide-bandgap* semiconductor material that shares the same molecular structure as Si, known as a *crystalline lattice* structure [8]. This structure lends itself well for free electrons to freely move throughout the structure, making SiC a suitable replacement for Si in semiconductor devices. When an electron is at rest, covalently bonded with its parent atom, it is in what is known as the *valence band*. In an atomic structure, there are many atomic orbitals in which an electron may reside. Electrons tend to come to rest

in the orbital where the least amount of energy is required. However, several factors such as ambient energy in the environment, the atomic structure resided in, and other quantum forces may prevent electrons from attaining their lowest possible energy state. Higher energy electrons reside in the upper levels of the valence band and if energy is obtained from an outside source such as heat, an electric field or even light, then the electron can break free from its covalent bond. When this happens, the electron enters what is known as the *conduction band* and leaves behind what is known as a *hole*. A *hole* is simply a vacant location in an atomic orbital with a net positive charge, therefore attracting other electrons. As this process continues, the positive charged holes effectively move in the opposite direction of the free electrons. This creates two types of carriers, electrons and holes, and when these carriers move across the semiconductor they create an electric current in the opposite direction of the electron travel. The term *bandgap* refers to the energy required to move an electron from the valence band to the conduction band in a molecular structure. These concepts are illustrated in Figure 1 [6]; in this visual, high energy electrons rise to the upper levels of the conduction band and high energy holes sink to the lower levels of the valence band.

Semiconductor Material	Bandgap (eV)
Germanium	0.66
Silicon	1.12
Gallium Arsenide	1.42
4H-Silicon Carbide	3.26
Gallium Nitride	3.4
Silicon Nitride	5
Diamond	5.5
Silicon Dioxide	9

Table 1. Semiconductor Material Bandgap Values [8]

The energy an electron carries is measured in *electron volts*, or eV. Si is the most prominent semiconductor material because it exhibits a comparatively small bandgap of 1.12 eV, this being the energy required to free an electron from its covalent bond. This attribute means Si semiconductor devices are fast and require a small nominal voltage for operation. It also means that Si devices are more susceptible to undesired device breakdown when exposed to environments with excess ambient energy present, such as high temperatures and radiation. When this happens, the molecular structure of the

semiconductor is inundated with carriers and the electrical current can no longer be controlled leading to complete circuit failure. SiC exhibits a bandgap of 3.26 eV, nearly triple the bandgap of Si. The wide bandgap gives SiC an innate resistance to the ambient energy in an environment causing unwanted conduction. Table 4 displays common semiconductor materials and their associated bandgap values [8].

B. NULL Convention Logic (NCL)

Multi-Rail Encodings

Asynchronous systems typically encode their data in one of two ways: "bundled-data encoding" which utilizes one wire per bit and is accompanied by separate wires signifying request and acknowledge signals, and "multi-rail encoding" which utilizes more than one wire per bit which allows for a three-state logic encoding. NCL implements multi-rail encodings, most commonly dual-rail, to represent the presence of data as well as the absence of data. For example, a single bit in dual-rail encoding is represented by two wires, or *rails*, and has three possible values: DATA1, DATA0 and NULL. These rails are encoded in a one-hot scheme and are mutually exclusive, meaning if both are asserted simultaneously it is an error state. Table 1 displays the encodings for dual-rail signals in NCL. The purpose of the NULL state is to flush out the previous logic values and indicate DATA is not ready.

	NULL	DATA0	DATA1	INVALID
Rail0	0	1	0	1
Rail1	0	0	1	1

Table 2. Dual-Rail Encoding

In certain scenarios, it may be more efficient to implement quad-rail encoding. Quad-rail encoding uses four rails to represent a single bit but adds two more data values, DATA2 and DATA3. Quad-rail encoding also uses a mutually exclusive one-hot scheme and all encodings falling outside of this scheme are considered an error state, as seen in Table 2. In rare cases, encodings greater than quad-rail may be used and in such cases the same protocols used for quad-rail encoding are followed but expanded to suit the number of rails and possible data values.

	NULL	DATA0	DATA1	DATA2	DATA3	INVALID
Rail0	0	1	0	0	0	OTHERS
Rail1	0	0	1	0	0	
Rail2	0	0	0	1	0	
Rail3	0	0	0	0	1	

Table 3. Quad-Rail Encoding

There are generally two methods used to denoting multi-rail signals. The first of which, more commonly used in the design phases, is by *name*(*i*).rail<*y*>, where *name* signifies the actual name of the signal, *i* signifies the bit position of the wire in a multi-bit signal, and *y* signifies the rail being referenced for the particular signal. For example, *output_low*(*5*).*rail1* indicates rail one of bit position 5 in a signal named "output_low." The second method used to denote a multi-rail signal, more commonly used to represent signals in displaying equations, is by $name_i^y$. To clarify, the signal from the previous example would be denoted as $output\_low_5^1$. Throughout this dissertation, the first notation will be used.

NCL Gates

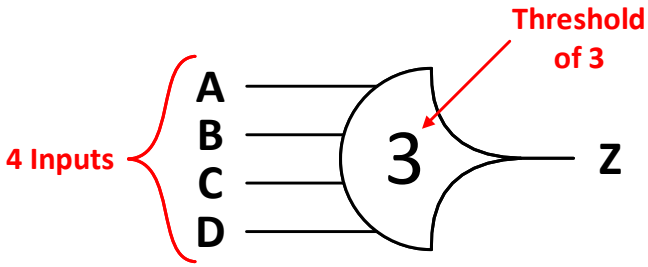

Figure 2. TH34 Gate Symbol

There are 27 fundamental gates, called *threshold* gates, in the NCL paradigm. As the name implies, the output of each gate will only be asserted once the inputs to the gate meet certain criteria, or *threshold*. Each of these gates has between two and four inputs with only one output, and performs a unique operation represented by a simple Boolean function. The naming convention clearly conveys the output equation for each gate and is as follows: *THmn*, where 1 ≤ *m* ≤ *n* in cases excluding weighted inputs, *m* represents the threshold value that must be met for the output to be asserted, and *n* represents the total

number of inputs to the gate. For example, there are four inputs to a TH34 gate and at least three of those inputs must be asserted for the output to be asserted. For the purposes of this dissertation, the inputs for NCL gates will be labeled *A* through *D* and the output will be labeled *Z*. Therefore, the output equation for a TH34 gate will be $Z = ABC + ABD + ACD + BCD$. The symbol for the TH34 gate can be seen in Figure 2.

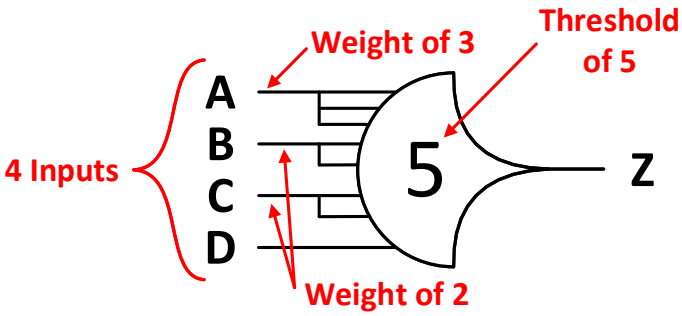

Figure 3. TH54w322 Gate Symbol

There are several addendums to this notation to indicate special functionality in a gate. The first of which is weighted inputs, signified by $THmnWx_1x_2...x_n$ where $1 < x \leq m$. The value of each consecutive variable corresponds to the weight of each input in order, e.g., x_1 represents the weighted value of input *A*, x_2 represents the weighted value of input *B*, and so on. Therefore, a TH54w322 represents a four-input gate with a threshold value of five, and input *A* has a weighted value of 3 while inputs *B* and *C* both have a weighted value of 2. The fourth input, *D*, has a weighted value of one and therefore does not have a value representing it in the list of weights. This means the output equation of the TH54w322 gate is $Z = AB + AC + BCD$, the symbol for which can be seen in Figure 3.

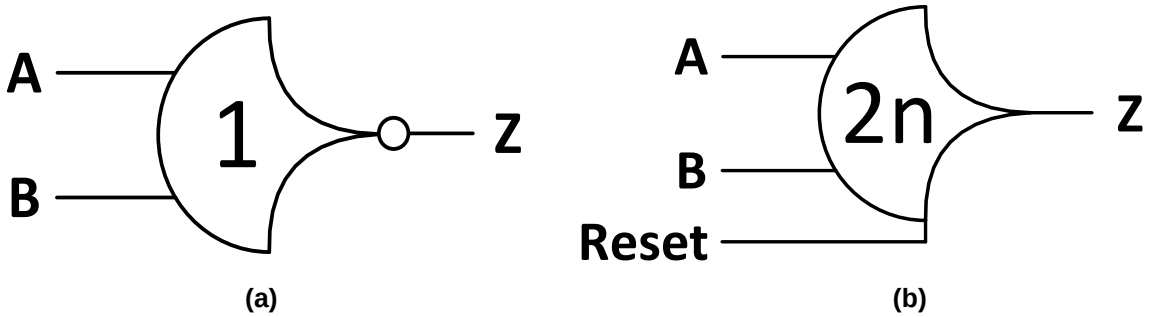

Figure 4. (a) TH12b Gate Symbol and (b) TH22n Gate Symbol

The next addendum to the *THmn* notation is required for inverting gates, which are represented by the form *TH1nb.* Generally, inverting NCL gates have a threshold of one and are analogous to their non-inverting counterpart with the only difference being the absence of an output inverter. The symbol for an inverting gate, the TH12b, is shown in Figure 4a. The last addendum is used for resettable gates, which are indicated by the concatenation of a single letter *d* or *n* to the end of the *THmn* notation. All resettable gates have a separate input designated "reset" and when asserted, forces the output of the gate to DATA if a *THmnd* or NULL if the gate is a *THmnn*. An example resetting gate symbol for the TH22n is seen in Figure 4b.

NCL Gate	Boolean Function
TH12	A + B
TH22	AB
TH13	A + B + C
TH23	AB + AC + BC
TH33	ABC
TH23w2	A + BC
TH33w2	AB + AC
TH14	A + B + C + D
TH24	AB + AC + AD + BC + BD + CD
TH34	ABC + ABD + ACD + BCD
TH44	ABCD
TH24w2	A + BC + BD + CD
TH34w2	AB + AC + AD + BCD
TH44w2	ABC + ABD + ACD
TH34w3	A + BCD
TH44w3	AB + AC + AD
TH24w22	A + B + CD
TH34w22	AB + AC + AD + BC + BD
TH44w22	AB + ACD + BCD
TH54w22	ABC + ABD
TH34w32	A + BC + BD
TH54w32	AB + ACD
TH44w322	AB + AC + AD + BC
TH54w322	AB + AC + BCD
THxor0	AB + CD
THand0	AB + BC + AD
TH24comp	AC + BC + AD + BD

Table 4. 27 Fundamental NCL Gate Asserting Functions

All NCL gates utilize hysteresis, meaning the logic in each gate is structured such that when the output is asserted, all inputs are required to be deasserted for the output to deassert unless the gate is resettable.

Hysteresis works in conjunction with the NULL state to achieve delay insensitivity in the NCL paradigm. The previously mentioned 27 fundamental NCL gates are listed in Table 3 with the output equation for the Boolean function they perform.

All Threshold gates are composed of four major sections that are integral to their functionality. These sections are *set, reset, hold0* and *hold1.* Each section is easily distinguishable when a transistor-level schematic of a gate is examined, as seen with the TH33 gate schematic pictured in Figure 5. The function of the *set* and *reset* sections is to change the output of the gate from logic0 to logic1 or from logic1 to logic0, respectively. The *hold0* and *hold1* sections work in series with a set of hysteresis transistors and are responsible for holding the current output logic value determined by the *set* and *reset* sections.

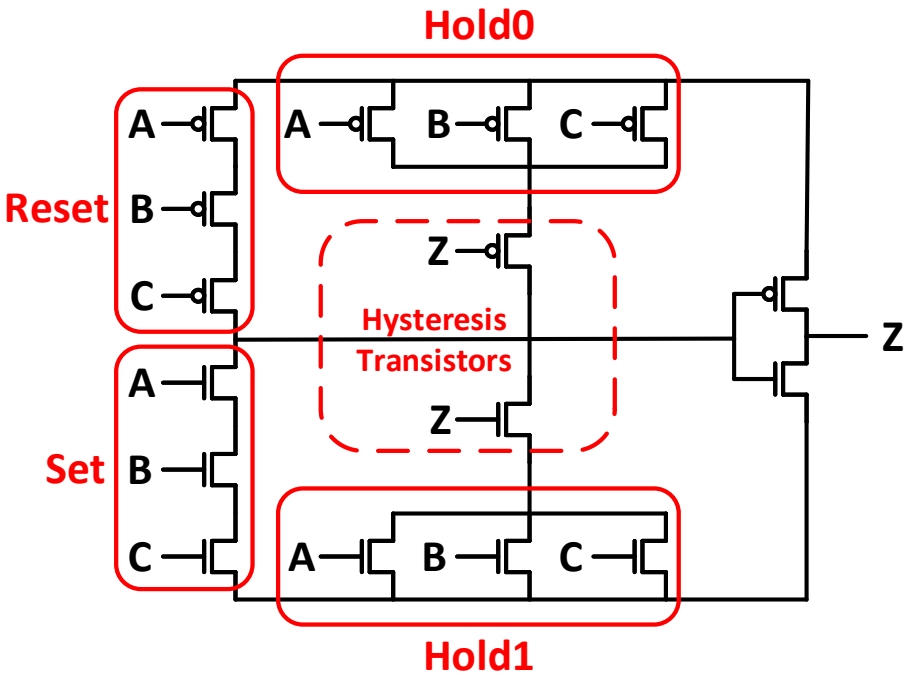

Figure 5. TH33 Gate Schematic

To achieve delay-insensitivity, NCL circuits also need to be fully observable and input-complete. For a design to be fully observable, all gate transitions must have an impact on an output of the circuit. The isochronic fork assumption allows for orphans that do not cause a gate transition to be ignored. Orphans are defined as wires that are asserted but are not used to determine an output of the circuit [4]. For a design to be input-complete, it must exhibit the following: all inputs of the circuit must transition from

NULL to DATA before any outputs transition from NULL to DATA, and all inputs of the circuit must transition from DATA to NULL before any outputs transition from DATA to NULL.

NCL Circuit Operation

In order to fully understand sequential NCL circuit operation, it is important to first understand the functionality of NCL registers. The schematic for a single-bit NCL register can be seen in Figure 6. These registers implement resettable and inverting NCL gates, and are analogous to a Boolean D-Flip Flop. In addition to storing the value of a dual-rail signal, these specialized registers can perform the handshaking functionality necessary for asynchronous sequential operation. This is done by the *Ko* signal, which signifies whether the register is currently storing DATA or NULL, and the *Ki* signal, which controls whether a DATA or NULL input can be stored in the register. When *Ki* is asserted, only DATA on the input will be able to pass through and be stored on the output, and when *Ki* is not asserted, only NULL will be able to be stored on the output. When DATA is stored in the register, *Ko* outputs a '0' which is known as a request for NULL (*rfn*) signal. Conversely, when a NULL is stored in the register, *Ko* outputs a '1' which is known as a request for DATA (*rfd*) signal. It is important to note that these handshaking signals are single bit, which means that completion logic is required for the *Ko* signal in multi-bit components. Completion logic is typically made up of *THnn* gates that ensure every bit of the component is outputting DATA or every bit is outputting NULL, and transitions the *Ko* signal appropriately. For example, in an 8-bit NCL register, 8 single-bit registers would be used simultaneously. In this setup, all *Ki* signals would be tied together into a single input signal and the eight *Ko* signals would be fed into two TH44 gates that feed into a single TH22 gate. The output of the TH22 gate would serve as the *Ko* signal for the entire 8-bit NCL register.

One of the most vital concepts in sequential NCL circuit operation is the interaction between the *Ki and Ko* signals that create the handshaking protocol. The simplest form of logic that exhibits this interaction in NCL is a single-stage pipeline, which consists of two NCL registers and a block of combinational logic as seen in Figure 8. The primary inputs for this circuit are connected to the input register, which are stored in the register and fed to the combinational logic depending on the value of the input register's *Ki* signal. The resulting signals from the combinational logic are connected to another register whose output represents

the circuit's primary output. The output register's *Ki* signal and the input register's *Ko* signal are both external signals, and the output register's *Ko* signal is connected directly to the input register's *Ki* signal.

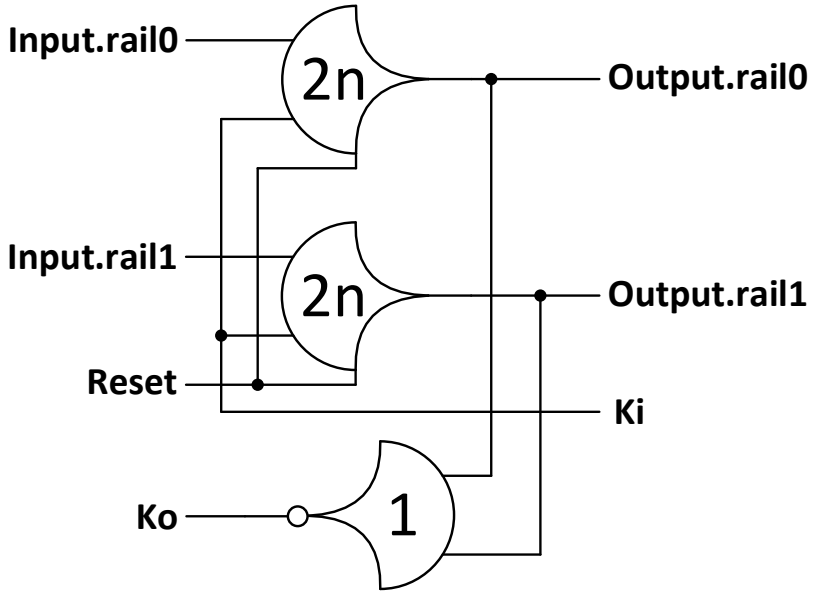

Figure 6. One-Bit NCL Register

At circuit startup, reset will be asserted which typically means both registers will be set to NULL. This means both register's *Ko* signals will have a value of '1' indicating a *rfd*. When a DATA wavefront is presented to the input register it will be stored immediately because the *Ki* signal, which is the *Ko* signal from the output register, is asserted. This action transitions the input register's *Ko* signal to a value of '0' indicating a *rfn*. The DATA wavefront then propagates through the combinational logic and if the external *Ki* signal is asserted then the results are stored in the output register, which causes its *Ko* signal to deassert. The system is now presenting the combinational logic's output and is ready for a NULL wavefront to propagate through and clear the previous DATA wavefront's values. When a NULL wavefront is presented, it is immediately stored on the input register which incurs a *rfd*. The NULL wavefront propagates through the combinational logic and is fed to the output register. When the external *Ki* signal is deasserted, the NULL wavefront is stored on the output register and the system has returned to its initial state. This process is what is known as a DATA-NULL cycle.

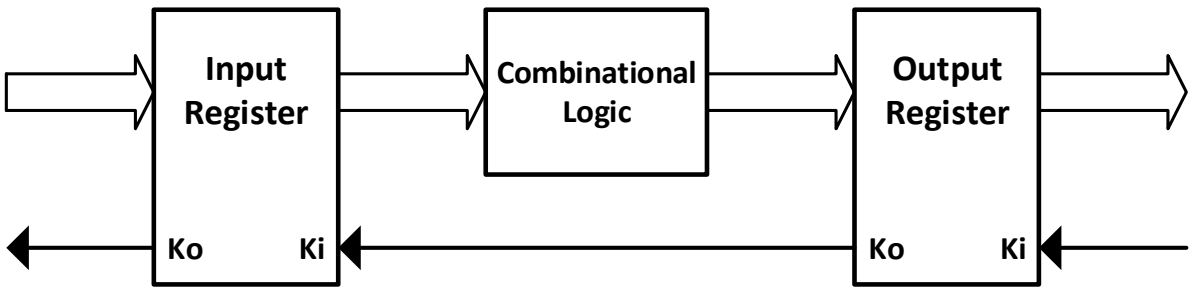

Figure 7. Single-Stage NCL Pipeline

The standard structure used for data flow and storage in NCL is the 3-ring register, an 8-bit version can be seen in Figure 7. This structure is necessary to preserve a set of DATA during the process of a NULL wavefront propagating through a design. In the reset state, the first two registers are initialized and store NULL values, while the third register stores and outputs a value of DATA0. With the *external Ki* at logic0, the state of each register is as follows: The combinational logic is presenting DATA to the first register but its *Ki* is logic0, preventing the DATA set from being stored; the second register is being presented with a NULL set from the first register and its *Ki* is logic0, allowing the NULL set to be stored; the third register is being presented with a NULL set from the second register and its *Ki* signals is logic1, which does not allow for the NULL set to be stored. The system is now in a steady-state, known as the NULL state of a 3-ring register because the external data output is NULL, and will not change as long as the value of the *external Ki* signals remains logic0. When the primary data output is NULL and the *external Ko* signal is logic1, the system is ready for reset to be deasserted the *external Ki* signal to be asserted.

When the *external Ki* signals is driven to logic1, a series of events will occur that will result in another steady-state known as the DATA state. After the *external Ki* signal is asserted, the output of the completion logic between the first and second register will switch to logic1 and will allow for the DATA set from the combinational logic to be stored on the first register. This means the primary data output and the input to the second register are both the value of the combinational logic DATA set. Now that the first register is storing a DATA set, its *Ko* is deasserted and allows the third register to store the NULL set being presented from the second register. This action also sends a NULL wavefront that propagates through the combinational logic. The *Ko* signal of the third register is then asserted, which deasserts its *Ko* signal feeding into the completion logic and allows the register to store the DATA set being presented

by the first register. The system has now reached the DATA state and will remain in this state until it is reset or the *external Ki* signal is deasserted.

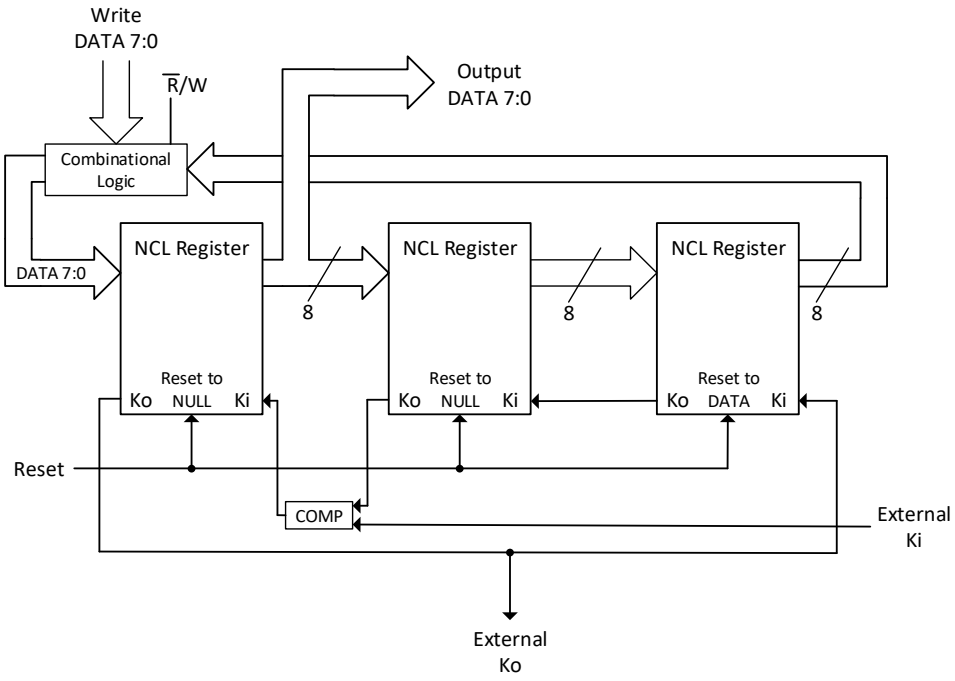

Figure 8. NCL 3-Ring Register

The time between two consecutive DATA wavefronts, or the DATA-to-DATA time, is referred to as T_{DD} and is analogous to a clock cycle in traditional synchronous systems. However, unlike a clock cycle, T_{DD} does not occur at a fixed frequency and can differ from cycle to cycle based on the data pattern and the completion time of the logic contained in the design. The circuit produced is one that operates at the highest speed possible for every cycle, resulting in what is known as “average-case performance.”. Synchronous designs require detailed timing analysis to determine the slowest path through the circuit which ultimately decides the maximum performance of the entire system. The result is a circuit that performs at what is known as “worst-case performance.” This only becomes more complex as the temperature of the circuit’s environment rises. NCL’s ability to automatically adjust performance in any environment makes it highly resistant to factors such as process variation, drastic temperature swings and supply voltage variations [9], [10]. NCL’s robustness and average-case performance, compared to synchronous systems worst-case performance, make it suitable for this work.

C. 8051 Microcontroller

Design Overview

The Intel 8051 has become one of the most widely used microcontrollers since its release in 1980. Its modular design and Harvard architecture make it a robust and efficient IC capable of a wide range of programmable functionalities. The features of the 8051 include [11]:

- 8-bit CPU
- Extensive Boolean processing (single-bit) capabilities
- 64K Program Memory address space
- 64K Data Memory address space
- 4K bytes of on-chip Program Memory
- 128 bytes of on-chip Data RAM
- 32 bidirectional and individually addressable I/O pins
- Two 16-bit timer/counters
- Full duplex UART
- 6-source/5-vector interrupt structure with two priority levels
- On-chip clock oscillator

The 8051 instruction set is optimized for 8-bit operations and provides a variety of addressing methods for accessing the internal data memory. In addition to the suite of 8-bit operations, extensive support for single bit operations is available which allow for direct bit manipulation. Note that there is a memory block designated for special function registers (SFRs) which shares the same memory space as the internal data memory but exists as a separate entity in the overall design. Also, the circuit designed in this work more closely resembles the 8031 variant of 8051, in which there is no program memory that resides internally, meaning all 64K program memory is external. The full 8051 architecture is shown in Figure 9 [11]. The focus of this dissertation is to produce a design with reduced power consumption due to the bus data transfer method and increase overall performance of the functionality contained in the instruction set. Due to this, a number of components and functions that are unessential to the operations performed by instructions were excluded from the design. These items include: the serial port, timer/counters,

interrupts, power control and the associated SFRs. The work in this dissertation builds off a previously designed NCL 8031 microcontroller successfully implemented in a 0.5 µm Si process [12]. This circuit will serve as a reference point for comparison in performance and power consumption simulation results.

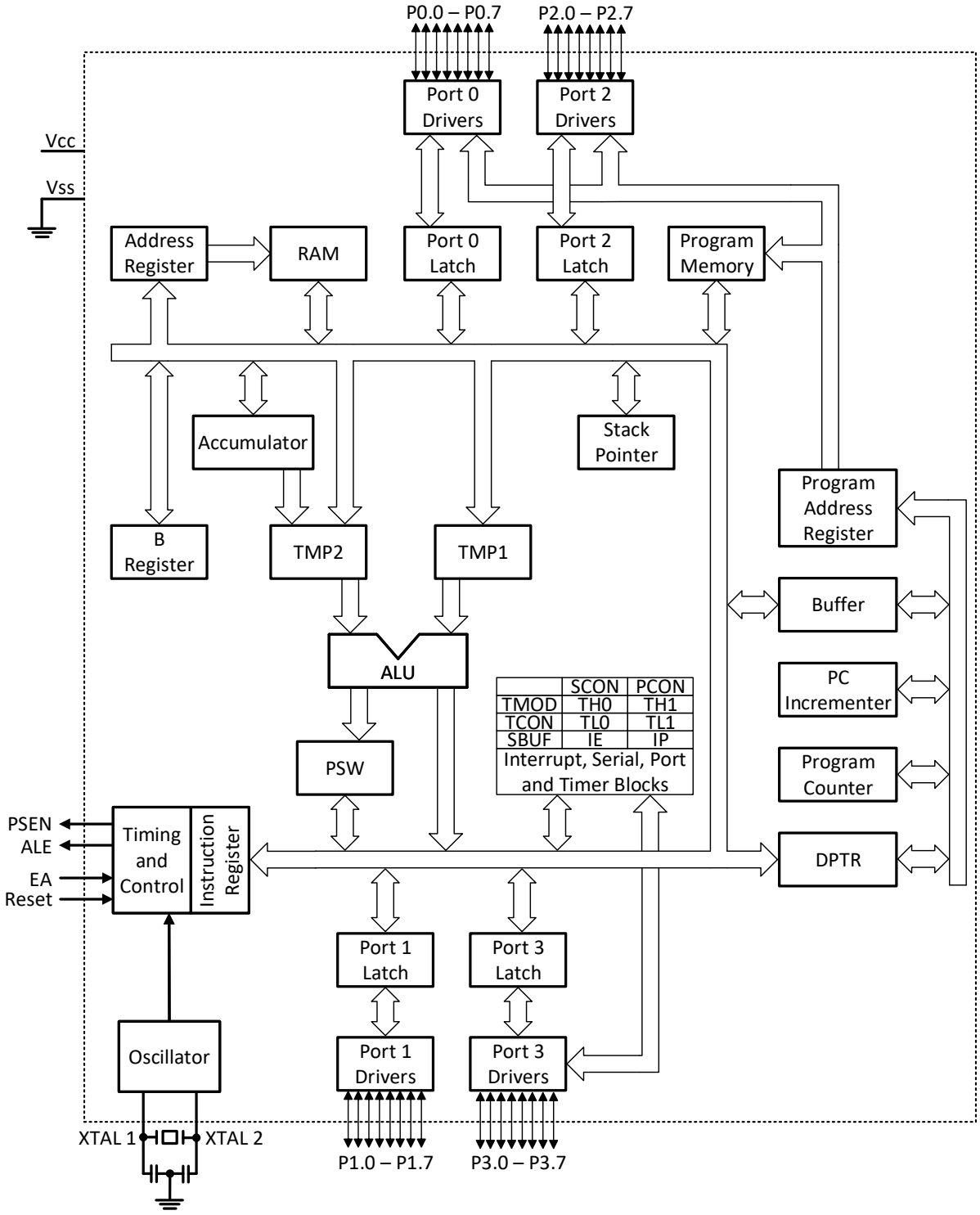

Figure 9. Intel 8051 Microcontroller Architecture

Addressing Methods

The 8051 utilizes 5 addressing methods known as: immediate, direct, register direct, register indirect and indexed addressing. In immediate addressing, the operand is represented by the 8-bit data value drawn directly from the program memory immediately after the instruction. Direct addressing uses the 8-bit data value read from program memory immediately after the instruction to represent the address of the operand. Only the internal memory address space, or the internal data memory and SFR space, can be accessed by direct addressing. Register direct addressing uses three bits of the instruction to specify a register in the internal data memory that contains the operand. This method is preferred for applications requiring a large amount of program memory because it only requires a one byte instruction to perform. Register indirect addressing specifies a specific register in the internal data memory that contains the address of the operand. Indexed addressing uses the data pointer (DPTR) or program counter (PC) to point to the base of a look up table in program memory and then increments the selected address by the value in the accumulator to reach the intended table entry containing the operand.

Component Descriptions

The following list contains the individual components of the 8051 microcontroller relevant to this work and their functionality:

- **Program Status Word –** The program status word (PSW) stores the value of several important status bits that reflect the current state of the CPU which are: the carry flag (CY), the auxiliary carry flag (AC), two bits signifying the current register bank selected, the overflow flag (OV), a parity flag, and two user-definable bits. In addition to arithmetic operations, the carry flag is also used to store values in several Boolean operations. The auxiliary carry flag is used primarily in BCD operations. The 8051 utilizes four register banks each containing eight 8-bit registers (R0-R7); the register bank currently in use is signified by the value of the two register select bits. The parity bit reflects the number of bits asserted in the Accumulator, P = 1 if there are an odd number of 1s in the accumulator and P = 0 if there are an even amount.
- **Accumulator –** The accumulator (ACC) is involved in over half of the instructions of the 8051, making it the most used SFR in the design. It is 8-bits wide and used as a general-purpose

register to accumulate and store the results of arithmetic, Boolean, data transfer and branch operations.

- **B Register –** The B register is similar to the ACC but is only used in two instructions: MUL AB and DIV AB. Many programmers also use the B register as a “scratch pad” or generic storage location.
- **Stack Pointer –** The stack pointer (SP) is an 8-bit register used to point to an internal data memory location of the stack. When an instruction pushes a value onto the stack, the SP is incremented and a value is stored at that location. When an instruction pops a value off the stack, the value at the location indicated by the SP is returned and then the SP is decremented.
- **Port Registers –** Each port is directly connected to their own set of registers which can be used as general purpose registers. If external memory is used then only port 1 and port 3 registers may be used for general purposes due to the actions port 0 and port 2 perform in accessing external memory.
- **Data Pointer –** The data pointer (DPTR) is the only user accessible 16-bit register in the 8051. It can also be used as two separate 8-bit register, if needed. The primary function of the DPTR is to store the address that will be used when accessing external data.
- **Program Counter –** The program counter (PC) is a 16-bit register that is not directly accessible by the user. Its primary function is to store the address of instructions read from the program memory. Upon startup, the PC is initialized to zero and is incremented by one after every read.

III. Approach

A. NCL Microcontroller Design Challenges

In order to implement the same component interaction scheme in an asynchronous NCL 8031 microcontroller that is used in a synchronous counterpart, involving a bus data transfer system, a couple adaptations are necessary. The first adaptation is that using the bus must be done under the assumption of isochronic forks if the design is to achieve quasi-delay insensitivity. This means that if a component is presented with a DATA or NULL set, then it must be assumed that all other components were also presented with that set. The second adaptation is to ensure that the bus can effectively present DATA-NULL sets throughout the microcontroller. This can be accomplished either by giving individual

components the capability to independently uphold each cycle or by making the control logic responsible for presenting a NULL cycle to all components between DATA cycles. The first method requires a 3-ring register or similar structure to be included in each component so that DATA and NULL sets are processed congruently. This method would be costly in terms of area, power and performance. The alternative requires pull-down resistors attached to each signal of the bus to ensure that NULL is presented whenever the bus is not being driven as DATA. This method is less costly in terms of area and performance but still consumes a large amount of power to implement due to the leakage power drawn through the pull down resistors during DATA sets.

The removal of the bus structure has significant effects on the adaptations required for the NCL 8031 design. First, there is no longer a need to tri-state component outputs that drive the bus, resulting in a reduction of area that was previously used for those circuits and the control signal logic to operate them. Also, this being the primary motivation behind this work, neither of the methods for maintaining congruent DATA-NULL wavefronts mentioned previously are required. The method to replace the bus architecture is straightforward: instead of implementing one data transfer path for all component's I/O, each data transfer path between components will now be implemented separately. With so many independent data transfer paths, it is necessary to utilize MUX and DEMUX circuit components at the output and input of each individual component. While this is costly in terms of area, it comes with the added benefit of improved overall power and performance which will be discussed later in this chapter. Also, upon system power up, the absence of the bus means all logic gates now require initialization by means of control logic. This control logic is also utilized in-between DATA sets to present a NULL wavefront that clears the previous state's data.

Figure 10 displays the general conceptual theory for the system level architecture used in this design. The general flow of processing instructions is identical to a typical synchronous microcontroller, despite the significant changes in data transfer operations. First, the timing and control (TAC) component stores the current instruction in the instruction register (IR). The TAC decodes the current instruction and generates control signals that are dependent on the operation being performed. These control signals are grouped together and transmitted to their respective "slave" components which then incite the appropriate action according to the instruction received. Control signal groups are composed of several different types

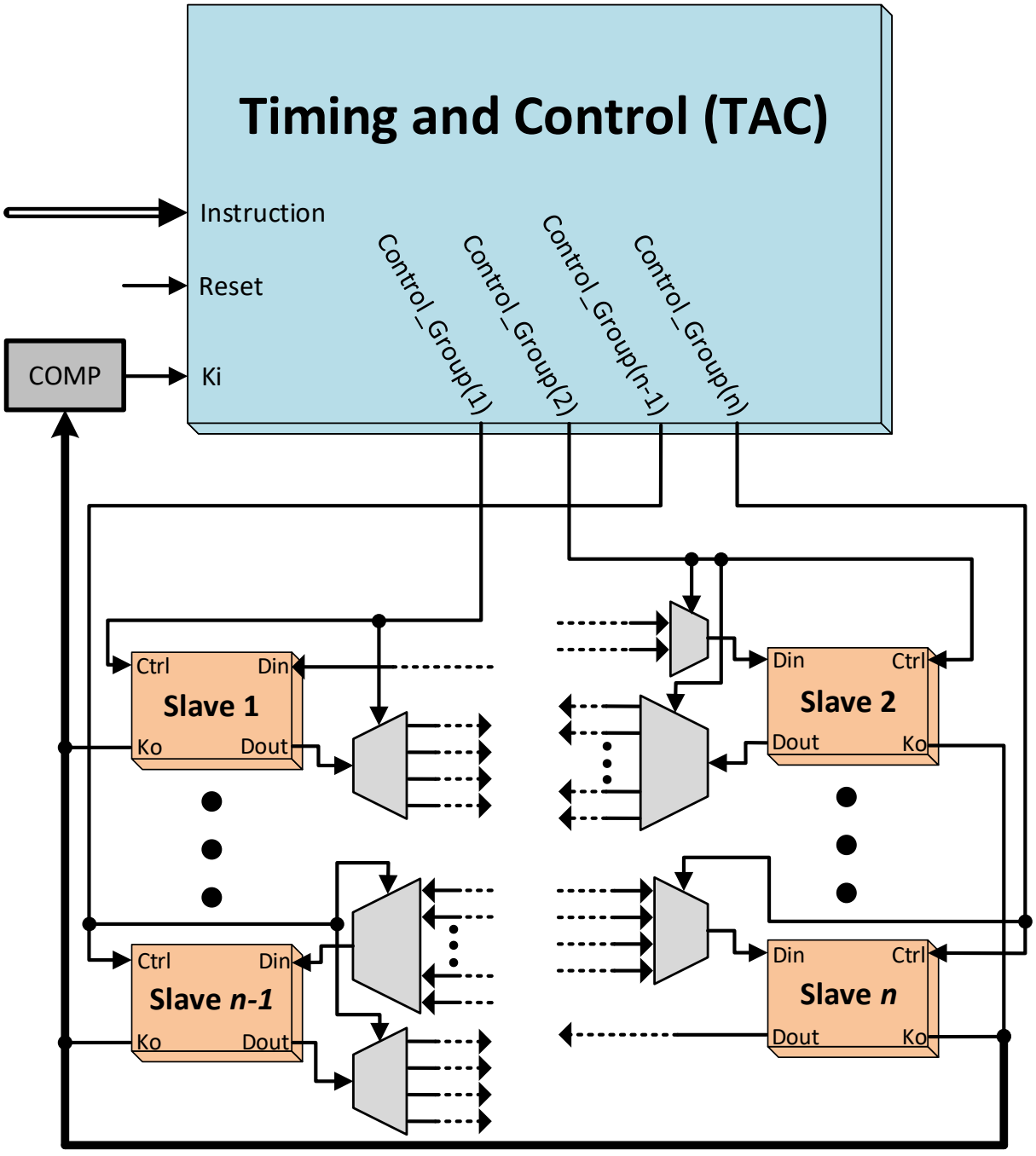

Figure 10. Timing and Control (TAC) and Bus Replacement Architectural Theory

of signals which may include: storage component read and/or write signals, operation selection, address encoding select signals, status flag set/reset, and MUX/DEMUX select signals for the data transfer system. Refer to Appendix A for a list and description of all control signals used. When DATA is being processed by a slave component, its respective *Ko* signal will deassert. Conversely, when a slave

component completes its operation then its *Ko* signal will be asserted. All *Ko* signals are fed into a completion logic block, the output is connected to the TAC's *Ki* input and is used to trigger state transitions in an asynchronous finite state machine (AFSM).

B. TAC Architecture and Functionality

As seen in Figure 11, the TAC is composed of the following: the IR, instruction decoder (MEAG_Opcode), a AFSM, state latches, and a control signal generation block (CS_Logic_Top) making up an asynchronous finite state machine (AFSM). The structure and functionality of each individual block is as follows:

- **Instruction Register –** The basic structure of the IR consists of a NCL 3-ring register and a 2-input MUX. The feedback loop of the 3-ring register is connected to one input of the MUX and the external instruction input is connected to the second. The select signal is decided by whether the AFSM is currently in state(0) and is used to choose between the current instruction stored in the 3-ring register, if it has not completed, or a new incoming instruction. The output of the MUX feeds into the input of the 3-ring register.
- **MEAG_Opcode –** This component decodes the instruction received from the IR and outputs a 111-bit mutually exclusive assertion group (MEAG) that represents the unique operation that is selected. Though the 8051 ISA is composed of 256 instructions, many of these instructions utilize one of the eight registers in the currently selected memory bank contained in the SRAM. For example, the "*ADD A, Rn*" instruction adds the value of the accumulator with the value contained in a specific register in the SRAM and then stores the result in the accumulator. In the 8-bit value for this instruction, the three least significant bits represent the register (0-7) to be used in the instruction. This means the "*ADD A, Rn*" instruction represents one unique operation but occupies 8 places in the instruction set. Given this information, the 256 individual instructions can be reduced to 111 unique operations.
- **State_Calc –** This is a block of control logic that calculates the next state of the system based on the current operation and the previous state of the system. State information is output in the form of a 13-bit MEAG signal.

- **State_Latches –** This is a unique NCL memory structure designed specifically for the state information received from the AFSM. Conceptually, this component is structured and behaves the same as a NCL 3-ring register; it differs from a 3-ring register by the data it stores, a 13-bit single-rail value as opposed to a dual-rail value, and it outputs the values from the previous data cycle as well as the current value. The State_Latches are responsible for storing the previous state of the AFSM for use in state calculations, sending the state(0) bit to the IR for use in selecting its input, and for sending the current state information to the control signal generation logic block.
- **CS_Logic_Top** – This is a large collection of combinational logic responsible for the generation of the control signals used throughout the microcontroller. For every state in each instruction a unique set of control signals are calculated in this component and sent directly to their respective destinations.

During the reset state, the IR will be initialized to its NULL state and State_Latches will be initialized to output state(0) as the current state. When reset is deasserted and the state(0) bit is asserted, two important actions take place. First, the IR MUX is set to accept the external instruction signal and waits until a DATA set is presented. Second, CS_Logic_Top will generate the control signals necessary to perform and external program memory read operation. Once the instruction is read from external ROM, it is passed through port0 and the MUX-based data transfer architecture and presented to the TAC. When the external read and data transfer operations are completed, the TAC *Ki* signal will be asserted allowing for the instruction to be stored in the IR and passed to MEAG_Opcode for decoding. Each of the 111 unique operations in the 8051 instruction set are represented by a single bit of the resulting opcode signal. Once the instruction is decoded, the opcode is sent to the AFSM and the control signal generation block for later use. The AFSM then uses the opcode in conjunction with the previous state value from the State_Latches block to calculate the next state value. Once this is completed, the state latches store the next state value, output it to the control signal generation logic as the current state value, and store the previous state value to be used by the AFSM during the next cycle. The CS_Logic_Top component is now presented with both the opcode and the current state value. Using these two sets of data, the control signals needed in each state are generated and output to their corresponding components throughout the microcontroller and the cycle begins again.

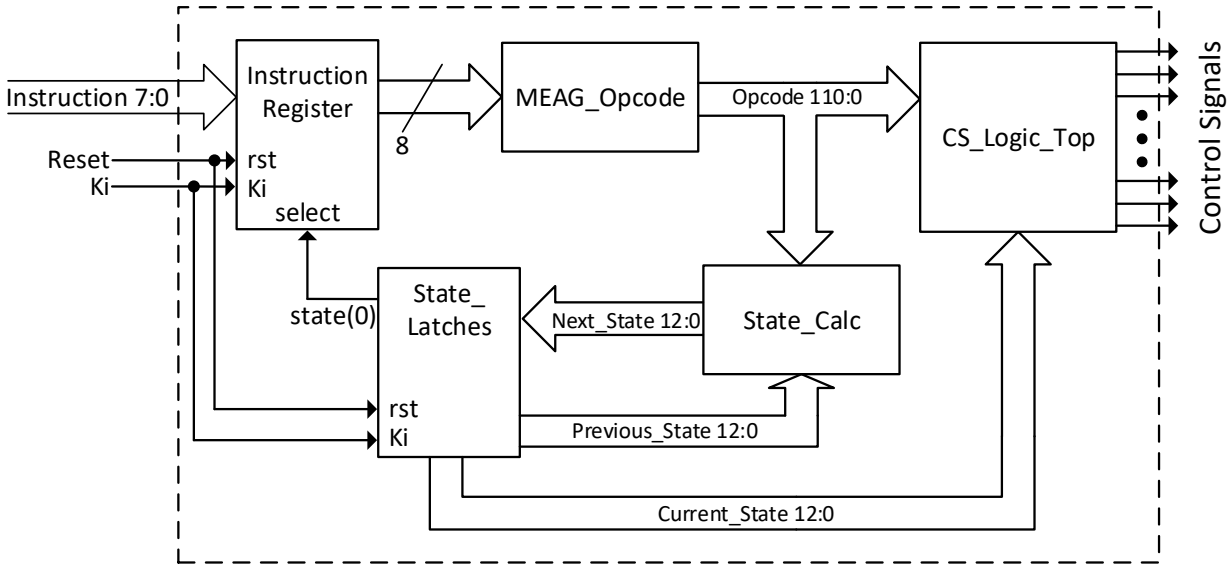

Figure 11. TAC Architecture

C. Performance Improvements

Each instruction entails a unique state list, up to a maximum of twelve states, and in each state a unique set of control signals are asserted. The full instruction set and state list for this design can be viewed in Appendix B. Using the conventional bus architecture in this design means that only one component can transfer data to another location in each state. Therefore, overall performance is limited by use of the bus, which often leaves data transfer operations waiting in the pipeline for others to complete. Using the MUX-based bus replacement architecture improves overall performance by making it possible to perform multiple data transfers in a single state, thereby reducing the total amount of states required to perform instructions. The only stipulation is that data sources and destinations must all be mutually exclusive of one another. Implementing the NCL 8031 using the bus architecture required a total of 877 states to perform all instructions [12]. Implementing the MUX-based data transfer architecture only requires a total of 677 states to perform all instructions, resulting in a 22.8% increase in overall performance.

D. Power Efficiency Improvements

In order to implement a traditional bus structure in a NCL circuit, there are modifications necessary compared to that in a synchronous circuit. First, as mentioned previously, pull-down resistors are added

to each individual wire of the bus. This is to pull the bus low when not being driven to ensure that a NULL wavefront is presented to each component connected to the bus between DATA wavefronts. Second, the outputs of each component connected to the bus need to be strongly buffered so they are capable of driving the bus high. This is due to the extra current drawn through the pull-down resistors added to the bus wires. The large buffers and the pull-down resistors require a significant amount of power to implement, especially in SiC processes due to their comparatively large nominal supply voltages. Replacing the bus with the MUX-based data transfer system eliminates the need for the pull-down resistors and large driving buffers, thereby improving the overall power efficiency of the microcontroller, as shown in Section IV.B.

IV. Results

A. Transistor-Level Simulation Results

After full-length functional HDL simulations of the SiC 8031 component were completed, the design was imported into the Cadence Virtuoso design environment to be implemented in the Raytheon HTSIC process. Once imported, full-length transistor level simulations were performed for functional verification and performance evaluation using room temperate average-case models. In these tests, the SiC 8031 design functioned properly, proving the full functionality of the TAC component and the MUX-based data transfer system. The results are shown in a subset of simulation waveforms seen in Figures 12 to 16. The T_{DD} for these simulations ranged from 2.29 to 3.44 µs, resulting in a maximum operating frequency range from 0.29 to 0.45 MHz.

Figure 12 displays the waveform results of the *ADD A, #data* instruction transistor-level simulation. In this instruction, an 8-bit data value received from external program memory is used as a direct input to the ALU to be added with the current value of the accumulator register and the result is then stored back in the accumulator. In the second state of this instruction, there are two internal data transfers performed simultaneously, from Port 0 to the ALU's primary input and from the accumulator to the ALU's secondary input, which previously was not possible using the bus architecture. This instruction's simulation results demonstrate proper functionality of the following: simultaneous internal data transfer operations, correct

incrementing of the program counter, valid use of the immediate addressing method, correct functionality of ALU arithmetic operations, and appropriate functionality of all involved SFRs.

Figure 13 displays the waveform results of the *ANL A, Rn* instruction transistor-level simulation. In this instruction, the 3 LSBs of the instruction designate which register in the SRAM whose value will be transferred to the ALU for the *AND* operation. This instruction's simulation results demonstrate proper functionality of the following: SRAM address encoding, SRAM data retrieval, and ALU Boolean operations.

Figure 14 displays the waveform results of the *MOV A, Rn* instruction transistor-level simulation. This instruction is simple, the value of the register designated by the instruction is stored in the accumulator. This instruction's simulation results demonstrate correct functionality of the MOV group of instructions.

Figure 15 displays the waveform results of the *JNC rel* instruction transistor-level instruction. This instruction reads an 8-bit value from external program memory and stores it in a temporary register. Then, based on the value of the carry bit in the PSW, the AFSM will either end the current instruction and begin the next or add the value in the temporary register to the current value in the PC and then store it in the PC. If the carry bit is asserted the current instruction will end, and if the carry bit is deasserted the operation will be performed before the next instruction begins. This instruction's simulation results demonstrate correct functionality of the following: bitwise read operations, PC arithmetic operations, and instruction branching.

Figure 16 displays the waveform results of the *SETB C* instruction transistor-level simulation. This instruction is used to assert the bit in the PSW SFR that represents the carry value used in ALU operations. This is performed by transferring the entire 8-bit PSW value to the ALU, performing the appropriate bitwise operation, and then transferring the result back to the PSW register. This instruction's simulation results demonstrate correct functionality of the control logic and ALU operations utilized in the BIT instructions. For the readers understanding, in the following waveforms all *Rail0* wires in dual-rail signals are blue and all *Rail1* wires are green.

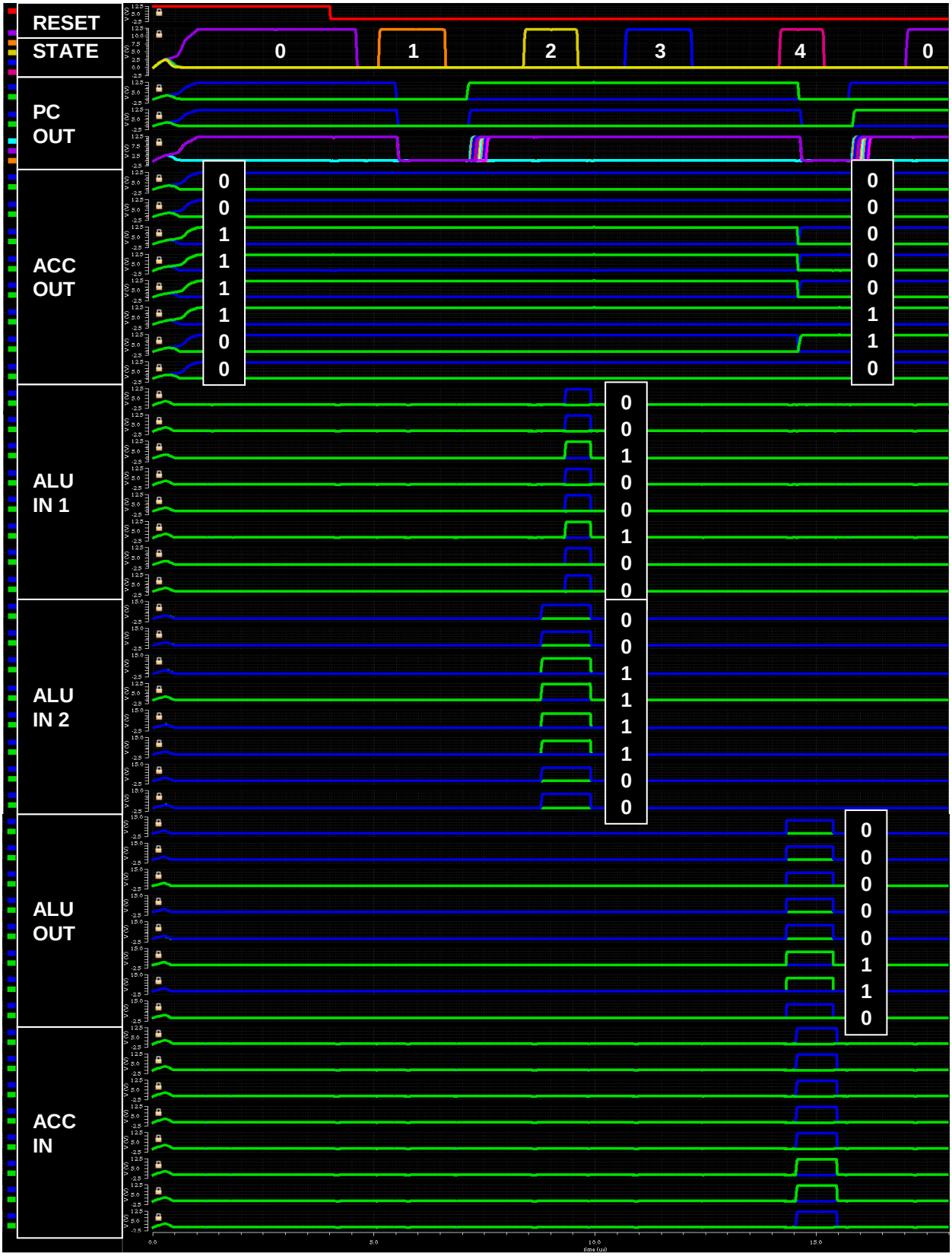

Figure 12. *ADD A, #Data* Instruction Simulation Output Waveform

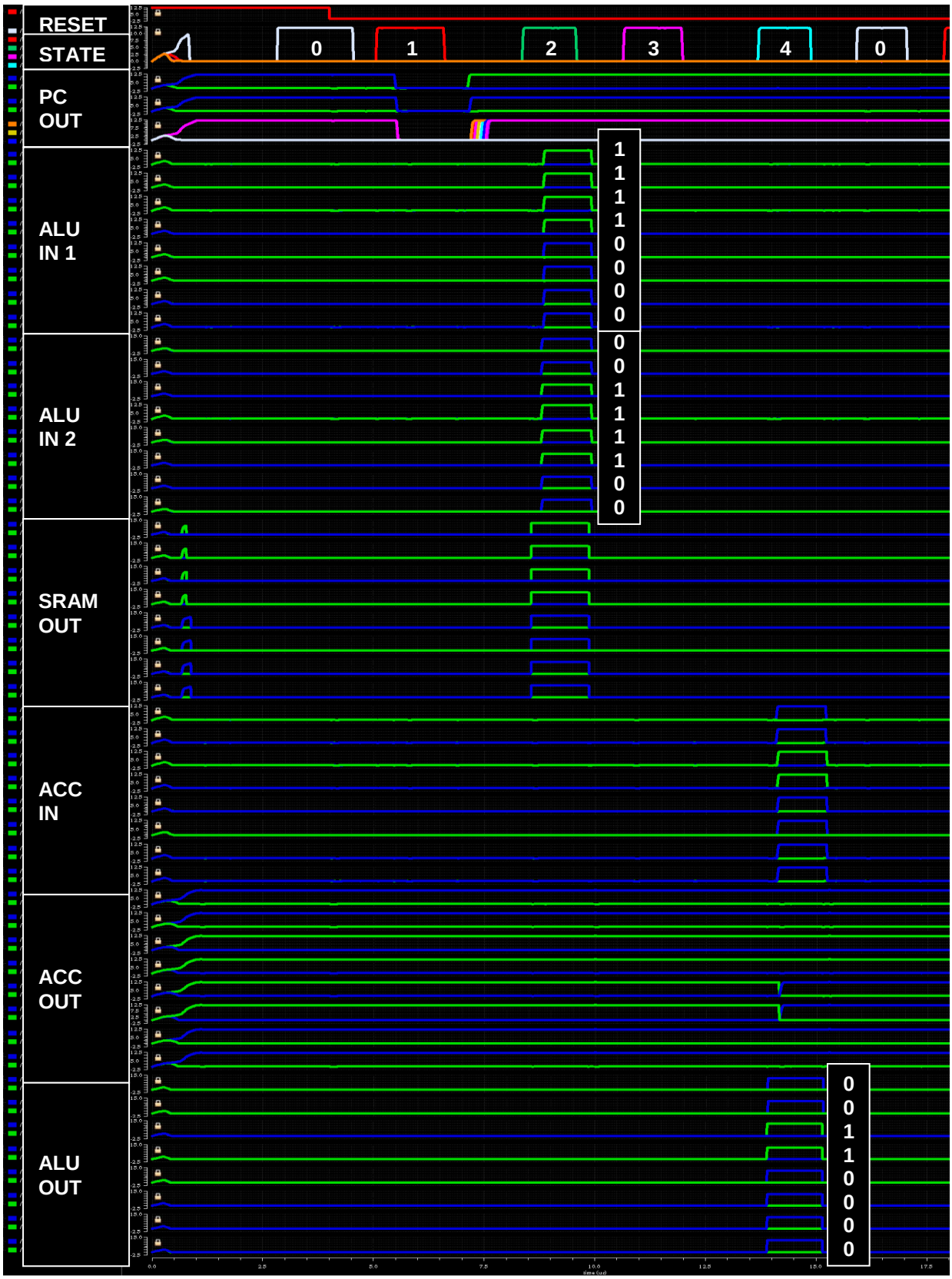

Figure 13. *ANL A, Rn* Instruction Simulation Output Waveform

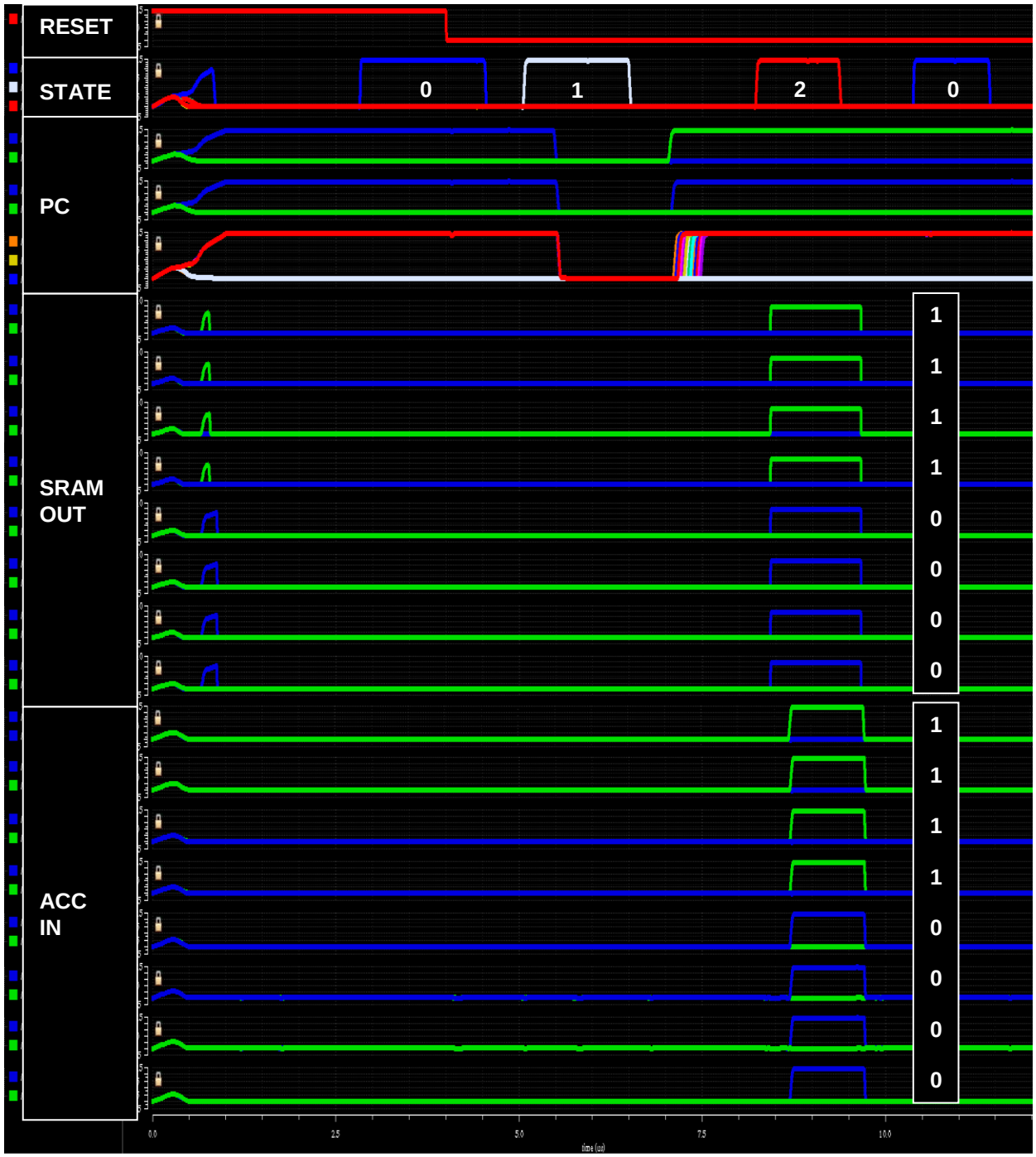

Figure 14. *MOV A, Rn* Instruction Simulation Output Waveform

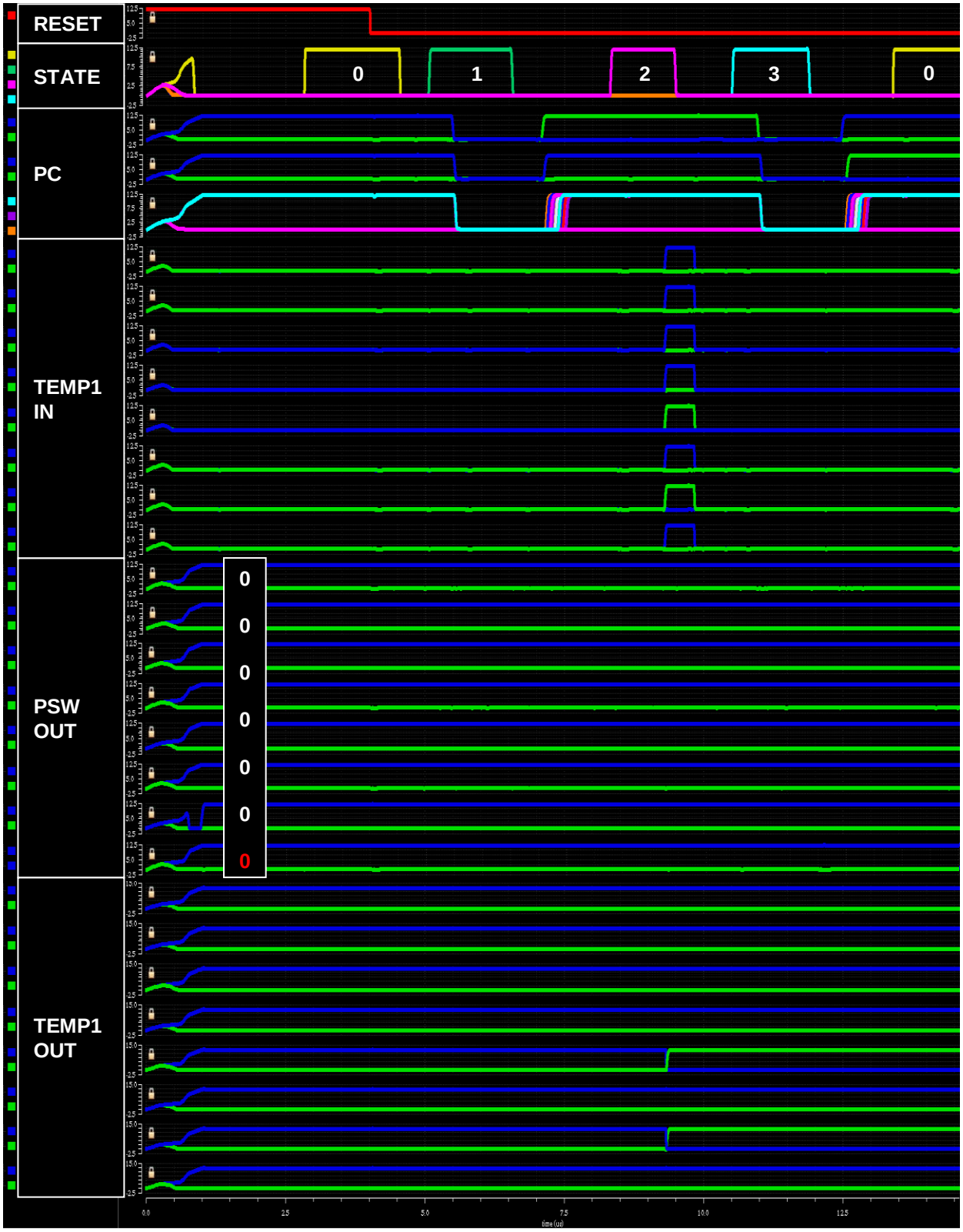

Figure 15. ***JNC rel*** **Instruction Simulation Output Waveform**

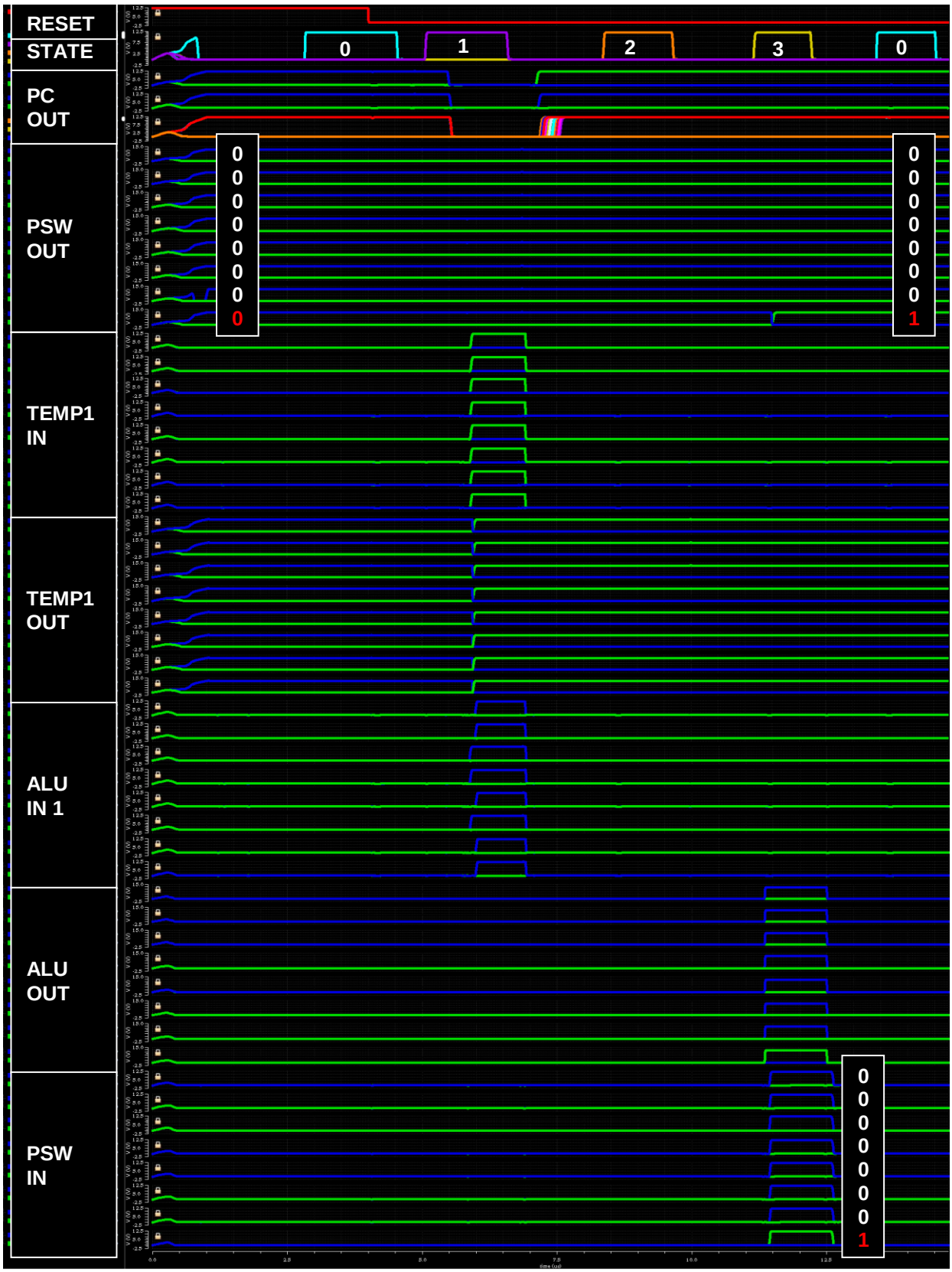

Figure 16. *SETB C* Instruction Simulation Output Waveform

B. Power Comparison Simulation Results

To gauge the effectiveness of the overall power performance improvements, separate test setups were created that perform the same operation but utilize the different data transfer methods. Power measurements are collected from simulations conducted for both setups in order to obtain an accurate side-by-side comparison of the power consumed during a data transfer process. The first test setup is composed of copies of the microcontroller's accumulator and temp1 registers that have been modified to operate in conjunction with a bus data transfer system. The outputs of the registers have been tri-stated and buffered in order to properly drive a DATA set to the bus wires. Also, included in the test setup are the control signals necessary to operate the registers, external inputs connected to the bus used for initial data transmission, and pull-down resistors attached to each individual wire of the data bus. The pull-down resistors (2 kΩ) and register output buffers are sized appropriately to produce acceptable rise and fall times for DATA-NULL transitions. The schematic used in the bus architecture power simulations is shown in Figure 16.

The second test setup, seen in Figure 17, models the behavior of the MUX-based data transfer architecture and uses the same accumulator and temp1 register components utilized in the SiC 8031 IC. To operate in conjunction with the MUX and DEMUX circuits, each component's I/O are changed from single wire bi-directional signals to two separate signals; one signal is used as the component's input and the other is used as its output. In Figure 17, each register's inputs are connected to the left side of the schematic component, leading to the output of a 2-to-1 MUX component, and their outputs are connected to the component's right side, leading to the input of a 1-to-2 DEMUX component. Each register's input MUX gate contains two inputs, one from external I/O and the other from the output of the opposite register's output DEMUX gate. Likewise, each component's output MUX gate contains two outputs, one to external I/O and the other to the opposite register's input MUX. This setup results in a cyclical data transfer structure between the registers with external I/O for both registers.

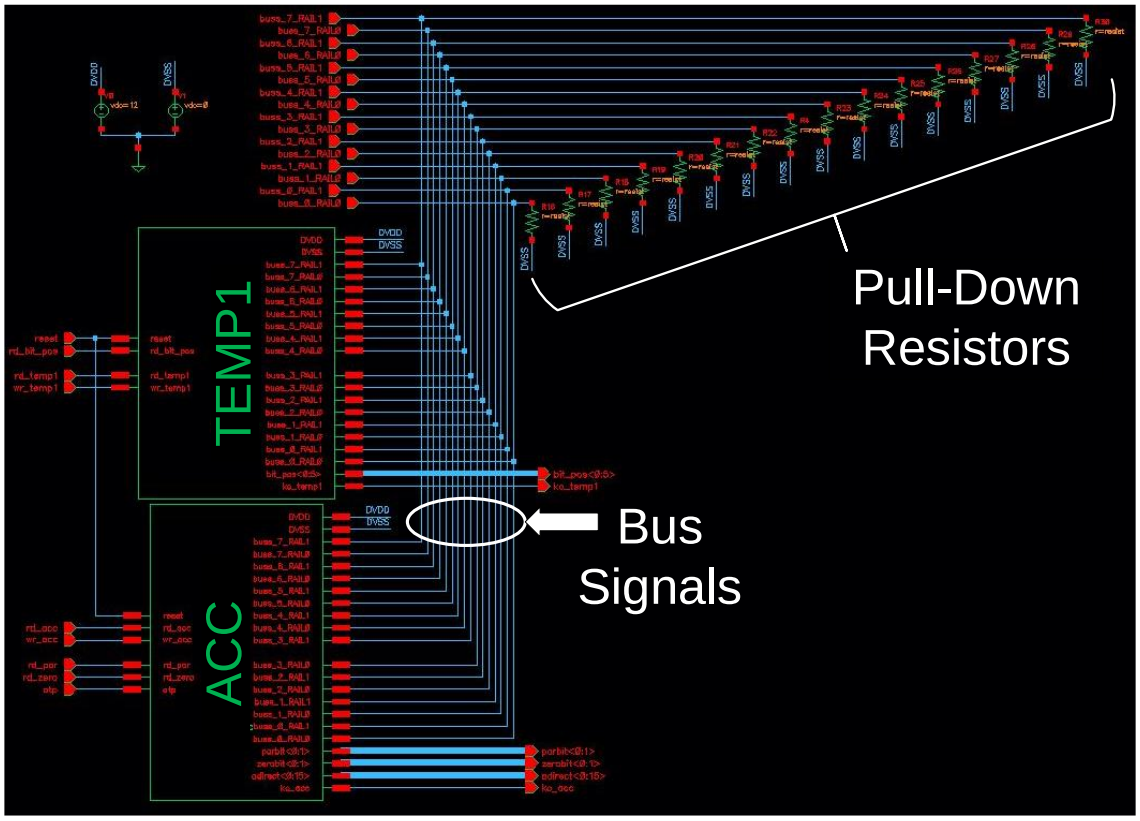

Figure 16. Bus Architecture Cadence Power Simulation Setup

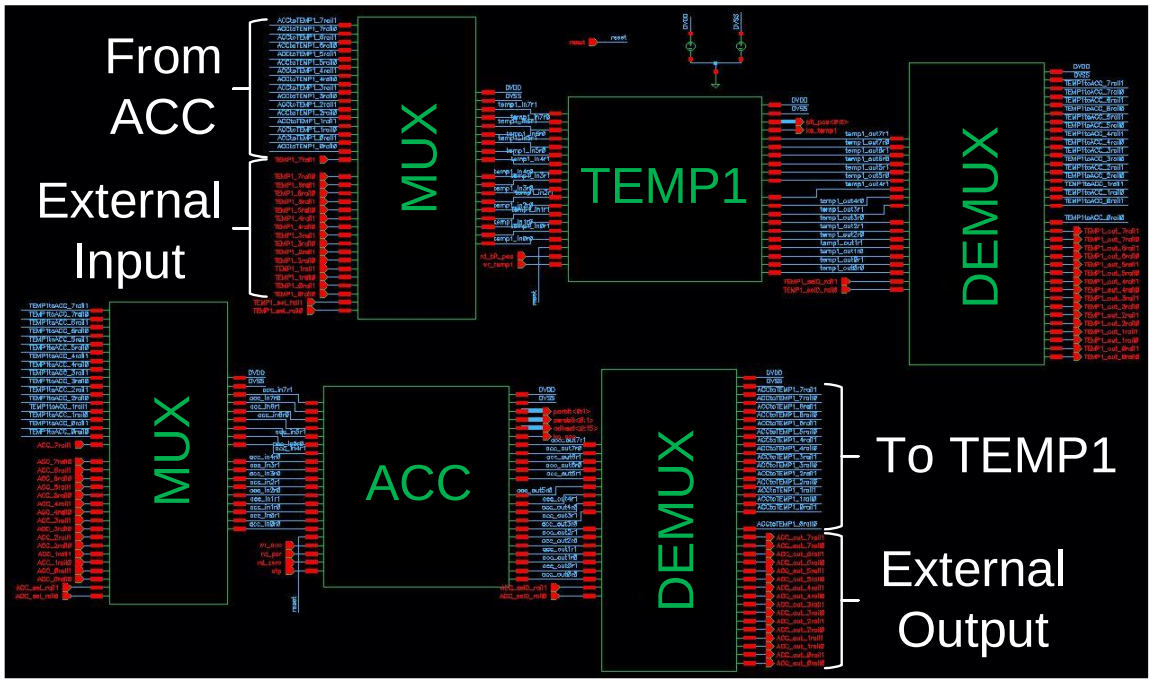

Figure 17. MUX-Based Architecture Cadence Power Simulation Setup

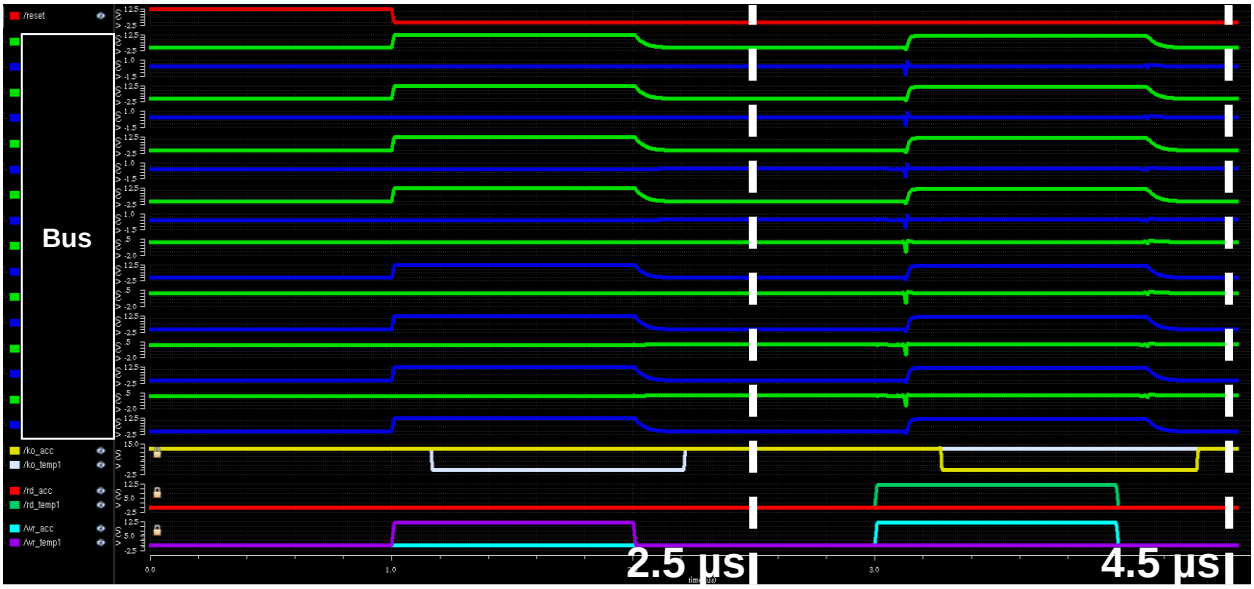

Figure 19. Bus Architecture Power Simulation Waveform

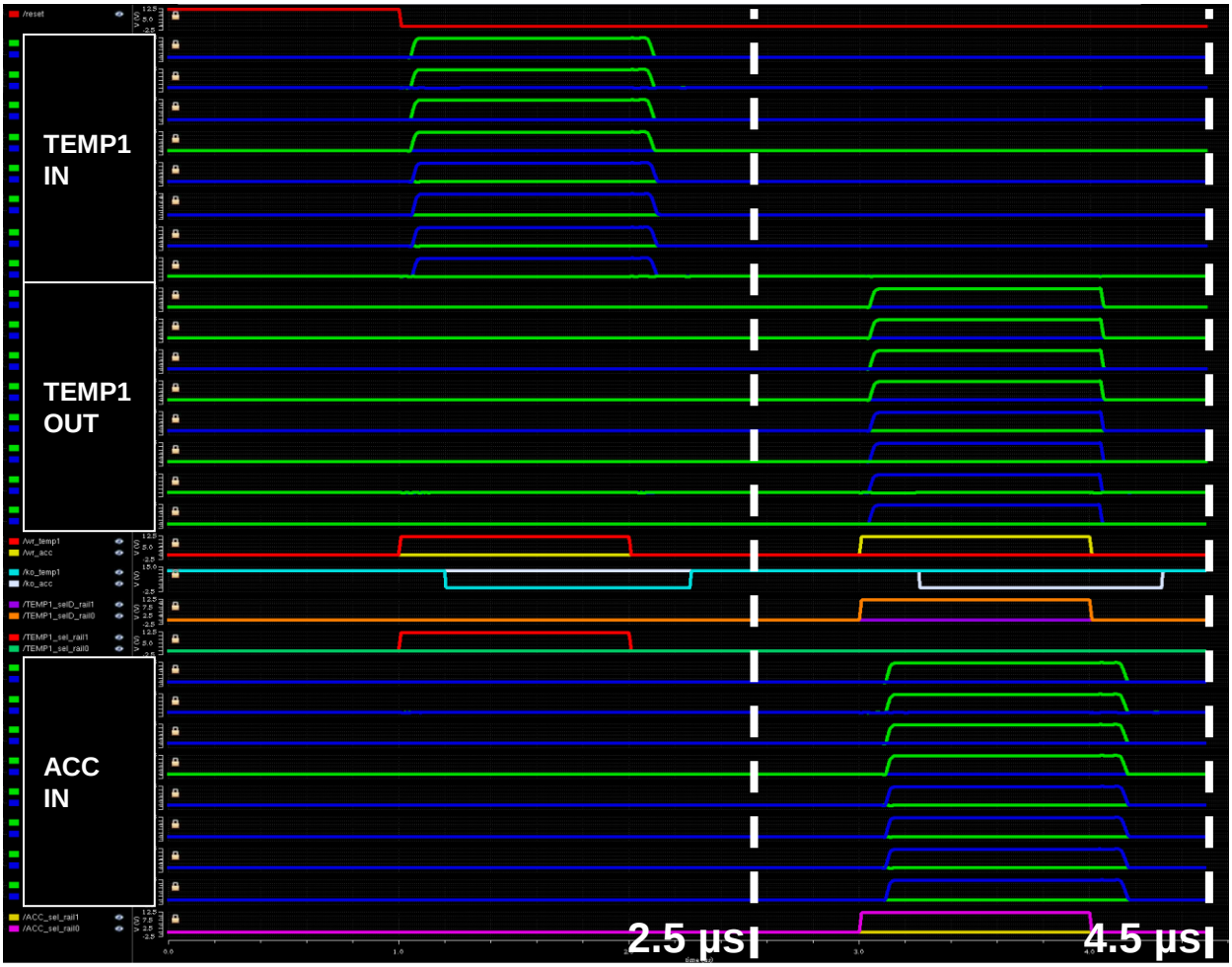

Figure 20. MUX-Based Architecture Power Simulation Waveform

Data Transfer Method	Power (mW)
Bus Architecture	369.080
MUX-Based Architecture	1.497

Table 5. Power Efficiency Comparison Simulation Results

To acquire an accurate comparison between the two data transfer architectures, the power measurements were only taken between 2.5µs and 4.5µs in this test. These start and end times were chosen because it represents a full DATA-NULL cycle containing only internal component activity. Also, all signals begin and end deasserted, as seen in the waveforms, meaning all switching activity for the data transfer is contained within the period for both architectures. The resulting power consumption for both architectures across this data transfer period is shown in Table 5. As expected, the MUX-based architecture greatly out-performed the bus architecture with over two orders of magnitude less power consumed during the data transfer. The large amount of power consumed by the bus architecture can be attributed primarily to the large supply voltage for the still-developing RaySiC process.

V. Conclusion

This dissertation presents the first set of asynchronous 8031 microcontroller components designed in a high temperature SiC process. While the delay insensitivity provided by the asynchronous NULL Convention Logic enables the designed circuits to function properly at high temperatures and largely tolerate the device variations in this still-developing SiC process by Raytheon, the traditional bus-based data transfer architecture has been replaced with a MUX-based counterpart for improving power efficiency and overall performance of the design, at the cost of circuit size/area.

This IC implements the major functionality of the Intel 8051/8031 microcontroller design capable of executing the full 8-bit instruction set. This instruction set is composed of 256 individual instructions capable of doing byte and bit-wise manipulation and calculations. Simulations were performed using the first CMOS SiC high-fidelity process design kit developed for the Cadence Virtuoso environment. Results show proper instruction set functionality with a 22.8% increase of overall performance, and an overall power efficiency increase of over two orders of magnitude.

Today, there is a rising demand for ICs capable of stable performance in environments with large temperature swings, especially in the currently prevailing smart car and aerospace technology fields. The NCL 8051/8031 microcontroller IC makes for a prime candidate in these fields due to its dependable operation in high-temperature environments attributed to its delay insensitivity and SiC material characteristics, and its robust functional capabilities attributed to the modular 8051 microcontroller family design. These fields also require power efficient designs due to power availability and packaging reliability concerns. While this design represents a promising first step, further work is needed to improve this ICs viability for commercial use. The design would benefit from the addition of the previously mentioned excluded common microcontroller functionalities. Also, with further design iterations and major architectural changes revolving around the use of the MUX-based data transfer method, it is possible the overall performance can be further improved.

VI. References

[1] D.J. Spry, P.G. Neudeck, L.Chen, D.Lukco, C.W. Chang, G.M. Beheim, M.J. Krasowski, and N.F. Prokop, "Processing and Characterization of Thousand-Hour 500 °C Durable 4H-SiC JFET Integrated Circuits", Proceeings of the 2016 IMAPsInternational High Temperature Electronics Conference (HiTEC 2016), Albuquerque, New Mexico USA, May 10-12, 2016

[2] R. F. Thompson, D. T. Clark, A. E. Murphy, E. P. Ramsay, D. A. Smith, R. A. R. Young, J. D. Cormack, J. McGonigal, J. Fletcher, C. Zhu, s. Finney, L. C. Martin, A. B. Horsfall, “High Temperature Silicon Carbide CMOS Integrated Circuits”, High Temperature Electronics Network (HiTEN) 2011.

[3] A. M. Francis, A. Rahman, J. Holmes, P. Shepherd, S. Ahmed, M. Barlow, S. Bhuyan, L. Caley, T. Moudy, H. A. Mantooth, J. Di, “Design of Analog and Mixed-Signal Integrated SiC CMOS Circuits with a High Fidelity Process Design Kit,” 2014 Government Microcircuit Applications & Critical Technology Conference (GOMACTech), March 2014

[4] K. M. Fant and S. A. Brandt, “NULL Convention Logic: A Complete and Consistent Logic for Asynchronous Digital Circuit Synthesis,” International Conference on Application Specific Systems, Architectures, and Processors, 1996.

[5] N. Kuhns, L. Caley, A. Rahman, S. Ahmed, J. Di, H. A. Mantooth, A. M. Francis, J. Holmes, “Complex High-Temperature CMOS Silicon Carbide Digital Circuit Designs,” IEEE Transactions on Device and Materials Reliability, vol. 16, no. 2, pp. 105-111, June 2016

[6] Caley, Landon John, "High Temperature CMOS Silicon Carbide Asynchronous Circuit Design" (2015). Theses and Dissertations. http://scholarworks.uark.edu/etd/30

[7] Microcontroller – Invention History and Story Behind the Scenes. (October 23, 2013) Available: http://www.circuitstoday.com/microcontroller-invention-history

[8] S. Dimitrijev, “The Energy-Band Model,” in Principles of Semiconductor Devices, 2nd edition, New York, Oxford University Press, 2012, ch. 2, sec. 2.2.3, pp. 55-60.

[9] B. Hollosi, M. Barlow, G. Fu, C. Lee, J. Di, S. C. Smith, H. A. Mantooth, and M. Schupbach, "Delay-Insensitive Asynchronous ALU for Cryogenic Temperature Environments,” IEEE Midwest Symposium on Circuits and Systems, August 2008.

[10] S. M. Nowick, M. Singh, “Asynchronous design – part 1: Overview and recent advances,” IEEE Design and Test, vol. 32, no. 3, pp. 5-18, June 1015

[11] Intel – MCS 51 Microcontroller Family User’s Manual. (February, 1994) Available: www.industrologic.com/MCS51FamilyUsersGuide.pdf

[12] Hollosi, Brent, “8051-compliant Asynchronous Microcontroller Core Design, Fabrication, and Testing for Extreme Environment” (2008). Theses and Dissertations. http://gradworks.umi.com/14/56/1456097.html

VII. Appendices

A. Appendix A: Control Signal Definitions

absolute - Modifies the PC according to the computation specified by the ACALL and AJMP instructions.

add - Addition of tmp1 and tmp2.

addc - Addition of tmp1, tmp2, CY bit.

and - Logical AND of tmp1 and tmp2.

anlb - Logically ANDs a bit in tmp2 whose position is specified by value read from temp1 and bit 7 of tmp1 which contains the CY bit.

anlbc - Logically ANDs the complement of a bit in tmp2 whose position is specified by value read from temp1 and bit 7 of tmp1 which contains the CY bit.

clrb - Clears bit of tmp1 whose position is specified by value read from temp1.

cpl - Complements tmp1.

cplb - Complements bit of tmp1 whose position is specified by value read from temp1.

c_zero - Supplies ALU with a low CY bit value.

da - Decimal adjust on tmp1.

dec - Decrements tmp1 by 1.

div - Divides tmp1 by tmp2.

fetch - Initiates the fetchx ASM. The fetchx ASM which synchronously generates the control signals responsible for fetching an 8-bit data value from a 16-bit external memory address specified by the contents of the PC.

fetchx - Identical to **fetch** except the contents of the DPTR are used to specify the 16-bit external memory address.

fetchxlow - Initiates the process of fetching an 8-bit data value from an 8-bit external memory address specified by the contents of either R0 or R1.

flag0 - Sets the CY, OV, and AC bits in the PSW based on the last ALU operation.

flag1 - Sets the CY and OV bits in the PSW based on the last ALU operation.

flag2 - Sets the CY bit in the PSW based on the last ALU operation.

flag3 - Sets the Parity bit in the PSW based on the current contents of the Accumulator.

gba (generate_byte_address) - Indicates that the byte address should be generated from the bit address being input to the SRAM block. All bit addresses between 00H-7FH are output as the byte address containing that bit. Bit addresses between 80H-FFH are output in the same way if that address is implemented in SFR space. If the bit address is of a non-implemented register the temp1 register is targeted. Used in the generation of addresses for Bit Addressing instructions e.g. CPL C, bit.

inc – Increments the ALU's tmp1 register by 1.

inc_pc - Increments PC by 1.

inc_dptr - Increments DPTR by 1.

jb - Initiates a jump if the jump check bit is high.

jnb - Initiates a jump if the jump check bit is low.

lda (load_direct_address) - Indicates that the internal memory address will be taken directly from the SRAM block's data input. All byte addresses between 00H-7FH are directly output. Byte addresses between 80H-FFH are directly output if that address is implemented in SFR space. If the byte address is of a non-implemented register the temp1 register is targeted. Used in generation of addresses for instructions using Direct Addressing e.g. Mv A, direct.

movb - Replaces bit 7 of tmp1 with a bit in tmp2 whose position is specified by value read from temp1.

movbc - Replaces a bit of tmp1 whose position is specified by value read from temp1 with bit 7 of tmp2.

mul - Multiplies tmp1 and tmp2.

obe (one_bit_encode) - Indicates that only I_0 along with the RSx bits are used to generate the address. Used in generation of addresses for instructions using Indirect Addressing e.g. Mv A, @Ri.

or - Logical OR of tmp1 and tmp2.

orlb - Logically ORs bit in tmp2 whose position is specified by value read from temp1 and bit 7 of tmp1 which contains the CY bit.

orlbc - Logically ORs the complement of a bit in tmp2 whose position is specified by value read from temp1 and bit 7 of tmp1 which contains the CY bit.

pc_addadptr - Adds the Accumulator to the DPTR and stores the result in the PC.

pc_addapc - Adds the Accumulator to the PC and stores the result in the PC.

pc_addrel - Adds an 8-bit relative offset to the PC.

rd_ac - Reads out the AC bit from the PSW.

rd_acc_zero - Activates zero checker logic of the Accumulator and distributes it to JUMP_CALC.

rd_alu_low - Reads out the low byte of the last ALU operation onto the ALU block output.

rd_alu_high - Reads out the low byte of the last ALU operation onto the ALU block output.

rd_bit_pos - Reads out the 3-bit encoded bit position from temp1.

rd_c - Reads out the CY bit from the PSW.

rd_comp - Reads out an SRAM byte or SFR register's contents onto the SRAM block output depending on the SRAM_BLOCK's address latch whose contents are generated by the Address Generator. As such, this signal's assertion is always preceded in some previous state by a computation of the Address Generator.

rd_hc00 - Reads out the hardcoded value x00 from the HC_BLOCK.

rd_hcff - Reads out the hardcoded value xff from the HC_BLOCK.

rl - Rotates tmp1 left.

rlc - Rotates tmp1 left through CY.

rr - Rotates tmp1 right.

rrc - Rotates tmp1 right through CY.

rri – read register information
Reads out the register select bits in the PSW for use by the Address Generator.

setb - Sets bit of tmp1 whose position is specified by value read from temp1.

subb - Subtraction of tmp1 and CY bit from tmp2.

swap - Exchanges the 4 low order bits of tmp1 with its 4 high order bits.

tbe (three_bit_encode) - Indicates that all 3 bits of the instruction $I_2 – I_0$ along with the RSx bits are used to generate the address. Used in generation of addresses for Register instructions e.g. Mv A, Rn

wr_acc - Writes the accumulator block's data input value into the Accumulator.

wr_b - Writes the B register block's data input value into SFR B.

wr_comp - Writes in the SRAM_Block's data latch to either an SRAM byte or SFR register's contents depending on the SRAM_BLOCK's address latch whose contents are generated by the Address Generator. As such, this signal's assertion is always preceded in some previous state by a computation of the Address Generator.

wr_dpl - Writes the DPTR block's data input value into SFR data pointer low (DPL).

wr_dph - Writes the DPTR block's data input value into SFR data pointer high (DPH).

wsd (write sram data) - Writes the SRAM block's data input value into the SRAM_Block's data latch.

wr_pc_low - Writes the PCDPTR block's data input value into the lower 8 bits of the PC.

wr_pc_high - Writes the PCDPTR block's data input value into the higher 8 bits of the PC.

wr_psw - Writes the PSW block's data input value into SFR PSW.

wr_sp - Writes the SP block's data input value into SFR SP.

wr_temp1 - Writes the temp1 block's data input value into the temp1 register.

wr_temp2 - Writes the temp2 block's data input value into the temp2 register.

wr_tmp1 - Writes the ALU block's data 1 input value into the ALU's tmp1 register.

wr_tmp2 - Writes the ALU block's data 2 input value into the ALU's tmp2 register.

writex - Initiates the process of writing an internal 16-bit data value to an external memory address specified by the contents of DPTR.

writexlow - Initiates the process of writing an internal 8-bit data value to an external memory address specified by either R0 or R1.

xchd - Exchanges the 4 lower order bits of tmp1 and tmp2.

xor - Logical XOR of tmp1 and tmp2.

***name_name* –** Control signal used to indicate transfer of data from the component named first to the component named second. For example, the “temp1_acc” signal is used to represent the transfer of data from the temp1 register to the accumulator. These signals are used to generate the control signals for the MUX and DEMUX gates used in the data transfer path.

B. Appendix B: Instruction Set and State List

No Operation

NOP	opcode: 0
S0	fetch, port0_tac
S1	inc_pc

ALU Instructions

ADD A, Rn	opcode: 1
S0	fetch, port0_tac
S1	rri, tbe, inc_pc
S2	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2
S3	add
S4	flag0, rd_alu_low, alul_acc, wr_acc
ADD A, direct	**opcode: 2**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2, inc_pc
S4	add
S5	flag0, rd_alu_low, alul_acc, wr_acc
ADD A, @Ri	**opcode: 3**
S0	fetch, port0_tac
S1	rri, obe, inc_pc
S2	wr_temp1, rd_comp, sram_temp1
S3	lda, temp1_srama
S4	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2
S5	add
S6	flag0, rd_alu_low, alul_acc, wr_acc
ADD A, #data	**opcode: 4**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_tmp1, port0_tmp1, wr_tmp2, acc_tmp2
S3	add
S4	flag0, rd_alu_low, inc_pc, alul_acc, wr_acc
ADDC A, Rn	**opcode: 5**
S0	fetch, port0_tac
S1	rri, tbe, inc_pc
S2	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2
S3	addc, rd_c
S4	flag0, rd_alu_low, alul_acc, wr_acc

ADDC A, direct	opcode: 6
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2, inc_pc
S4	addc, rd_c
S5	flag0, rd_alu_low, alul_acc, wr_acc
ADDC A, @Ri	**opcode: 7**
S0	fetch, port0_tac
S1	rri, obe, inc_pc
S2	wr_temp1, rd_comp, sram_temp1
S3	lda, temp1_srama
S4	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2
S5	addc, rd_c
S6	flag0, rd_alu_low, alul_acc, wr_acc
ADDC A, #data	**opcode: 8**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_tmp1, port0_tmp1, wr_tmp2, acc_tmp2
S3	addc, rd_c
S4	flag0, rd_alu_low, inc_pc, alul_acc, wr_acc
SUBB A, Rn	**opcode: 9**
S0	fetch, port0_tac
S1	rri, tbe, inc_pc
S2	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2
S3	subb, rd_c
S4	flag0, rd_alu_low, alul_acc, wr_acc
SUBB A, direct	**opcode: 10**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2, inc_pc
S4	subb, rd_c
S5	flag0, rd_alu_low, alul_acc, wr_acc
SUBB A, @Ri	**opcode: 11**
S0	fetch, port0_tac
S1	rri, obe, inc_pc
S2	wr_temp1, rd_comp, sram_temp1
S3	lda, temp1_srama
S4	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2
S5	subb, rd_c
S6	flag0, rd_alu_low, alul_acc, wr_acc

SUBB A, #data	opcode: 12
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_tmp1, port0_tmp1, wr_tmp2, acc_tmp2
S3	subb, rd_c
S4	flag0, rd_alu_low, inc_pc, alul_acc, wr_acc
INC A	**opcode: 13**
S0	fetch, port0_tac
S1	wr_tmp1, acc_tmp1, inc_pc
S2	inc
S3	rd_alu_low, alul_acc, wr_acc
INC Rn	**opcode: 14**
S0	fetch, port0_tac
S1	rri, tbe, inc_pc
S2	wr_tmp1, rd_comp, sram_tmp1
S3	inc
S4	rd_alu_low, wsd, alul_sram
S5	wr_comp
INC direct	**opcode: 15**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	wr_tmp1, rd_comp, sram_tmp1, inc_pc
S4	inc
S5	rd_alu_low, wsd, alul_sram
S6	wr_comp
INC @Ri	**opcode: 16**
S0	fetch, port0_tac
S1	rri, obe, inc_pc
S2	wr_temp1, rd_comp, sram_temp1
S3	lda, temp1_srama
S4	wr_tmp1, rd_comp, sram_tmp1
S5	inc
S6	rd_alu_low, wsd, alul_sram
S7	wr_comp
DEC A	**opcode: 17**
S0	fetch, port0_tac
S1	wr_tmp1, acc_tmp1, inc_pc
S2	dec
S3	rd_alu_low, alul_acc, wr_acc

DEC Rn	opcode: 18
S0	fetch, port0_tac
S1	rri, tbe, inc_pc
S2	wr_tmp1, rd_comp, sram_tmp1
S3	dec
S4	rd_alu_low, wsd, alul_sram
S5	wr_comp
DEC direct	**opcode: 19**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	wr_tmp1, rd_comp, sram_tmp1, inc_pc
S4	dec
S5	rd_alu_low, wsd, alul_sram
S6	wr_comp
DEC @Ri	**opcode: 20**
S0	fetch, port0_tac
S1	rri, obe, inc_pc
S2	wr_temp1, rd_comp, sram_temp1
S3	lda, temp1_srama
S4	wr_tmp1, rd_comp, sram_tmp1
S5	dec
S6	rd_alu_low, wsd, alul_sram
S7	wr_comp
INC DPTR	**opcode: 21**
S0	fetch, port0_tac
S1	inc_dptr, inc_pc
MUL A, B	**opcode: 22**
S0	fetch, port0_tac
S1	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, b_tmp2, inc_pc
S2	mul
S3	flag1, rd_alu_low, alul_acc, wr_acc, rd_alu_high, wr_b, aluh_b
DIV A, B	**opcode: 23**
S0	fetch, port0_tac
S1	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, b_tmp2, inc_pc
S2	div
S3	flag1, rd_alu_low, alul_acc, wr_acc, rd_alu_high, wr_b, aluh_b
DA A	**opcode: 24**
S0	fetch, port0_tac
S1	wr_tmp1, acc_tmp1, inc_pc
S2	rd_c, rd_ac, da
S3	flag2, rd_alu_low, alul_acc, wr_acc

ANL A, Rn	opcode: 25
S0	fetch, port0_tac
S1	rri, tbe, inc_pc
S2	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2
S3	and
S4	rd_alu_low, alul_acc, wr_acc
ANL A, direct	**opcode: 26**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2, inc_pc
S4	and
S5	flag0, rd_alu_low, alul_acc, wr_acc
ANL A, @Ri	**opcode: 27**
S0	fetch, port0_tac
S1	rri, obe, inc_pc
S2	wr_temp1, rd_comp, sram_temp1
S3	lda, temp1_srama
S4	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2
S5	and
S6	flag0, rd_alu_low, alul_acc, wr_acc
ANL A, #data	**opcode: 28**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_tmp1, port0_tmp1, wr_tmp2, acc_tmp2
S3	and
S4	flag0, rd_alu_low, inc_pc, alul_acc, wr_acc
ANL direct, A	**opcode: 29**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2, inc_pc
S4	and
S5	rd_alu_low, wsd, alul_sram
S6	wr_comp

ANL direct, #data	opcode: 30
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	wr_tmp1, rd_comp, sram_tmp1, inc_pc
S4	fetch, wr_tmp2, port0_tmp2
S5	and
S6	rd_alu_low, wsd, alul_sram, inc_pc
S7	wr_comp
ORL A, Rn	**opcode: 31**
S0	fetch, port0_tac
S1	rri, tbe, inc_pc
S2	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2
S3	or
S4	rd_alu_low, alul_acc, wr_acc
ORL A, direct	**opcode: 32**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2, inc_pc
S4	or
S5	flag0, rd_alu_low, alul_acc, wr_acc
ORL A, @Ri	**opcode: 33**
S0	fetch, port0_tac
S1	rri, obe, inc_pc
S2	wr_temp1, rd_comp, sram_temp1
S3	lda, temp1_srama
S4	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2
S5	or
S6	flag0, rd_alu_low, alul_acc, wr_acc
ORL A, #data	**opcode: 34**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_tmp1, port0_tmp1, wr_tmp2, acc_tmp2
S3	or
S4	flag0, rd_alu_low, inc_pc, alul_acc, wr_acc

ORL direct, A	opcode: 35
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2, inc_pc
S4	or
S5	rd_alu_low, wsd, alul_sram
S6	wr_comp
ORL direct, #data	**opcode: 36**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	wr_tmp1, rd_comp, sram_tmp1, inc_pc
S4	fetch, wr_tmp2, port0_tmp2
S5	or
S6	rd_alu_low, wsd, alul_sram, inc_pc
S7	wr_comp
XRL A, Rn	**opcode: 37**
S0	fetch, port0_tac
S1	rri, tbe, inc_pc
S2	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2
S3	xor
S4	rd_alu_low, alul_acc, wr_acc
XRL A, direct	**opcode: 38**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2, inc_pc
S4	xor
S5	flag0, rd_alu_low, alul_acc, wr_acc
XRL A, @Ri	**opcode: 39**
S0	fetch, port0_tac
S1	rri, obe, inc_pc
S2	wr_temp1, rd_comp, sram_temp1
S3	lda, temp1_srama
S4	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2
S5	xor
S6	flag0, rd_alu_low, alul_acc, wr_acc

XRL A, #data	opcode: 40
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_tmp1, port0_tmp1, wr_tmp2, acc_tmp2
S3	xor
S4	flag0, rd_alu_low, inc_pc, alul_acc, wr_acc
XRL direct, A	**opcode: 41**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	wr_tmp1, rd_comp, sram_tmp1, wr_tmp2, acc_tmp2, inc_pc
S4	xor
S5	rd_alu_low, wsd, alul_sram
S6	wr_comp
XRL direct, #data	**opcode: 42**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	wr_tmp1, rd_comp, sram_tmp1, inc_pc
S4	fetch, wr_tmp2, port0_tmp2
S5	xor
S6	rd_alu_low, wsd, alul_sram, inc_pc
S7	wr_comp
CLR A	**opcode: 43**
S0	fetch, port0_tac
S1	rd_hc00, hc00_acc, wr_acc, inc_pc
CPL A	**opcode: 44**
S0	fetch, port0_tac
S1	wr_tmp1, acc_tmp1, inc_pc
S2	cpl
S3	rd_alu_low, alul_acc, wr_acc
RL A	**opcode: 45**
S0	fetch, port0_tac
S1	wr_tmp1, acc_tmp1, inc_pc
S2	rl
S3	rd_alu_low, alul_acc, wr_acc
RLC A	**opcode: 46**
S0	fetch, port0_tac
S1	wr_tmp1, acc_tmp1, inc_pc
S2	rlc, rd_c
S3	rd_alu_low, alul_acc, wr_acc, flag2

RR A	opcode: 47
S0	fetch, port0_tac
S1	wr_tmp1, acc_tmp1, inc_pc
S2	rr
S3	rd_alu_low, alul_acc, wr_acc, flag2
RRC A	**opcode: 48**
S0	fetch, port0_tac
S1	wr_tmp1, acc_tmp1, inc_pc
S2	rrc, rd_c
S3	rd_alu_low, alul_acc, wr_acc, flag2
SWAP A	**opcode: 49**
S0	fetch, port0_tac
S1	wr_tmp1, acc_tmp1, inc_pc
S2	swap
S3	rd_alu_low, alul_acc, wr_acc

MOV Instructions

MOV A, Rn	opcode: 50
S0	fetch, port0_tac
S1	inc_pc, rri, tbe
S2	rd_comp, wr_acc, sram_acc
MOV A, direct	**opcode: 51**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	inc_pc, rd_comp, wr_acc, sram_acc
MOV A, @Ri	**opcode: 52**
S0	fetch, port0_tac
S1	inc_pc, rri, obe
S2	rd_comp, wr_temp1, sram_temp1
S3	lda, temp1_srama
S4	rd_comp, wr_acc, sram_acc
MOV A, #data	**opcode: 53**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_acc, port0_acc
S3	inc_pc
MOV Rn, A	**opcode: 54**
S0	fetch, port0_tac
S1	inc_pc, rri, tbe, wsd, acc_sram
S2	wr_comp

MOV Rn, direct	opcode: 55
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	inc_pc, rd_comp, wsd, sram_sram
S4	rri, tbe
S5	wr_comp
MOV Rn, #data	**opcode: 56**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wsd, port0_sram
S3	inc_pc, rri, tbe
S4	wr_comp
MOV direct, A	**opcode: 57**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama, wsd, acc_sram
S3	inc_pc, wr_comp
MOV direct, Rn	**opcode: 58**
S0	fetch, port0_tac
S1	inc_pc, rri, tbe
S2	rd_comp, wsd, sram_sram
S3	fetch, lda, port0_srama
S4	inc_pc, wr_comp
MOV direct, direct	**opcode: 59**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	inc_pc, rd_comp, wsd, sram_sram
S4	fetch, lda, port0_srama
S5	inc_pc, wr_comp
MOV direct, @Ri	**opcode: 60**
S0	fetch, port0_tac
S1	inc_pc, rri, obe
S2	rd_comp, wr_temp1, sram_temp1
S3	lda, temp1_srama
S4	rd_comp, wsd, sram_sram
S5	fetch, lda, port0_srama
S6	inc_pc, wr_comp

MOV direct, #data	opcode: 61
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	inc_pc
S4	fetch, wsd, port0_sram
S5	inc_pc, wr_comp
MOV @Ri, A	**opcode: 62**
S0	fetch, port0_tac
S1	inc_pc, rri, obe
S2	rd_comp, wr_temp1, sram_temp1
S3	lda, temp1_srama, wsd, acc_sram
S4	wr_comp
MOV @Ri, direct	**opcode: 63**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	inc_pc, rd_comp, wsd, sram_sram
S4	rri, obe
S5	rd_comp, wr_temp1, sram_temp1
S6	lda, temp1_srama
S7	wr_comp
MOV @Ri, #data	**opcode: 64**
S0	fetch, port0_tac
S1	inc_pc, rri, obe
S2	rd_comp, wr_temp1, sram_temp1
S3	lda, temp1_srama
S4	fetch, wsd, port0_sram
S5	inc_pc, wr_comp
MOV DPTR, #data16	**opcode: 65**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_dptrh, port0_pcdptr
S3	inc_pc
S4	fetch, wr_dptrl, port0_pcdptr
S5	inc_pc

Miscellaneous Instructions

MOVC A, @A + DPTR	opcode: 66
S0	fetch, port0_tac
S1	inc_pc
S2	pcl_temp1, wr_temp1, pch_temp2, wr_temp2
S3	acc_pcdptr, pc_addadptr
S4	fetch, wr_acc, port0_acc
S5	temp1_pcdptr, wr_pc_low
S6	temp2_pcdptr, wr_pc_high
MOVC A, @A + PC	**opcode: 67**
S0	fetch, port0_tac
S1	inc_pc
S2	pcl_temp1, wr_temp1, pch_temp2, wr_temp2
S3	acc_pcdptr, pc_addapc
S4	fetch, wr_acc, port0_acc
S5	temp1_pcdptr, wr_pc_low
S6	temp2_pcdptr, wr_pc_high
MOVX A, @RI	**opcode: 68**
S0	fetch, port0_tac
S1	inc_pc, rri, obe
S2	dptrl_temp1, wr_temp1
S3	rd_comp, wr_dptrl, sram_pcdptr
S4	fetchxlow, wr_acc, port0_acc
S5	temp1_pcdptr, wr_dptrl
MOVX A, @DPTR	**opcode: 69**
S0	fetch, port0_tac
S1	inc_pc, fetchx, wr_acc, port0_acc
MOVX @RI, A	**opcode: 70**
S0	fetch, port0_tac
S1	inc_pc, rri, obe
S2	dptrl_temp1, wr_temp1
S3	rd_comp, wr_dptrl, sram_pcdptr
S4	writexlow
S5	temp1_pcdptr, wr_dptrl
MOVX @DPTR, A	**opcode: 71**
S0	fetch, port0_tac
S1	inc_pc, writex

PUSH direct	opcode: 72
S0	fetch, port0_tac
S1	inc_pc, sp_tmp1, wr_tmp1
S2	inc
S3	fetch, lda, port0_srama
S4	inc_pc, rd_comp, wsd, sram_sram, rd_alu_low, lda, alul_srama
S5	wr_comp, rd_alu_low, wr_sp, alul_sp
POP direct	**opcode: 73**
S0	fetch, port0_tac
S1	inc_pc, sp_srama, lda
S2	rd_comp, wsd, sram_sram
S3	fetch, lda, port0_srama, sp_tmp1, wr_tmp1
S4	inc_pc, dec
S5	rd_alu_low, wr_sp, alul_sp, wr_comp
XCH A, Rn	**opcode: 74**
S0	fetch, port0_tac
S1	inc_pc, rri, tbe
S2	rd_comp, wr_temp1, sram_temp1, wsd, acc_sram
S3	wr_comp, temp1_acc, wr_acc
XCH A, Direct	**opcode: 75**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	inc_pc, rd_comp, wr_temp1, sram_temp1, wsd, acc_sram
S4	wr_comp, temp1_acc, wr_acc
XCH A, @Ri	**opcode: 76**
S0	fetch, port0_tac
S1	inc_pc, rri, obe
S2	wr_temp1, rd_comp, sram_temp1
S3	lda, temp1_srama
S4	rd_comp, wr_temp1, sram_temp1, wsd, acc_sram
S5	wr_comp, temp1_acc, wr_acc
XCHD A, @Ri	**opcode: 77**
S0	fetch, port0_tac
S1	inc_pc, rri, obe
S2	rd_comp, wr_temp1, sram_temp1
S3	lda, temp1_srama
S4	rd_comp, wr_tmp2, sram_tmp2, wr_tmp1, acc_tmp1
S5	xchd
S6	rd_alu_low, alul_acc, rd_alu_high, wsd, aluh_sram
S7	wr_comp

BIT Instructions

CLR C	opcode: 78
S0	fetch, port0_tac
S1	inc_pc, psw_tmp1, wr_tmp1, rd_hcff, hcff_temp1, wr_temp1
S2	clrb, rd_bit_pos
S3	rd_alu_low, wr_psw, alul_psw
CLR bit	**opcode: 79**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc, temp1_srama, gba
S4	wr_tmp1, rd_comp, sram_tmp1
S5	clrb, rd_bit_pos
S6	rd_alu_low, wsd, alul_sram
S7	wr_comp
SETB C	**opcode: 80**
S0	fetch, port0_tac
S1	inc_pc, psw_tmp1, wr_tmp1, rd_hcff, hcff_temp1, wr_temp1
S2	setb, rd_bit_pos
S3	rd_alu_low, wr_psw, alul_psw
SETB bit	**opcode: 81**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc, temp1_srama, gba
S4	wr_tmp1, rd_comp, sram_tmp1
S5	setb, rd_bit_pos
S6	rd_alu_low, wsd, alul_sram
S7	wr_comp
CPL C	**opcode: 82**
S0	fetch, port0_tac
S1	inc_pc, psw_tmp1, wr_tmp1, rd_hcff, hcff_temp1, wr_temp1
S2	cplb, rd_bit_pos
S3	rd_alu_low, wr_psw, alul_psw

CPL bit	opcode: 83
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc, temp1_srama, gba
S4	wr_tmp1, rd_comp, sram_tmp1
S5	cplb, rd_bit_pos
S6	rd_alu_low, wsd, alul_sram
S7	wr_comp
ANL C, bit	**opcode: 84**
S0	fetch, port0_tac
S1	inc_pc, psw_tmp1, wr_tmp1
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc, temp1_srama, gba
S4	rd_comp, wr_tmp2, sram_tmp2
S5	anlb, rd_bit_pos
S6	rd_alu_low, wr_psw, alul_psw
ANL C, /bit	**opcode: 85**
S0	fetch, port0_tac
S1	inc_pc, psw_tmp1, wr_tmp1
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc, temp1_srama, gba
S4	rd_comp, wr_tmp2, sram_tmp2
S5	anlbc, rd_bit_pos
S6	rd_alu_low, wr_psw, alul_psw
ORL C, bit	**opcode: 86**
S0	fetch, port0_tac
S1	inc_pc, psw_tmp1, wr_tmp1
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc, temp1_srama, gba
S4	rd_comp, wr_tmp2, sram_tmp2
S5	orlb, rd_bit_pos
S6	rd_alu_low, wr_psw, alul_psw
ORL C, /bit	**opcode: 87**
S0	fetch, port0_tac
S1	inc_pc, psw_tmp1, wr_tmp1
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc, temp1_srama, gba
S4	rd_comp, wr_tmp2, sram_tmp2
S5	orlbc, rd_bit_pos
S6	rd_alu_low, wr_psw, alul_psw

MOV C, bit	opcode: 88
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc, psw_tmp1, wr_tmp1, temp1_srama, gba
S4	rd_comp, wr_tmp2, sram_tmp2
S5	movb, rd_bit_pos
S6	rd_alu_low, wr_psw, alul_psw
MOV bit, C	**opcode: 89**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc, temp1_srama, gba
S4	wr_tmp1, rd_comp, sram_tmp1, psw_tmp2, wr_tmp2
S5	movbc, rd_bit_pos
S6	rd_alu_low, wsd, alul_sram
S7	wr_comp

Branch Instructions

JC rel	opcode: 90
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3*	inc_pc, rd_cy, jb
S4	temp1_pcdptr, pc_addrel
JNC rel	**opcode: 91**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3*	inc_pc, rd_cy, jnb
S4	temp1_pcdptr, pc_addrel
JB bit, rel	**opcode: 92**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc
S4	fetch, wr_temp2, port0_temp2
S5	inc_pc, temp1_srama, gba
S6	wr_tmp1, rd_comp, sram_tmp1
S7*	rd_alu_bit, jb
S8	temp2_pcdptr, pc_addrel

JNB bit, rel	opcode: 93
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc
S4	fetch, wr_temp2, port0_temp2
S5	inc_pc, temp1_srama, gba
S6	wr_tmp1, rd_comp, sram_tmp1
S7*	rd_alu_bit, jnb
S8	temp2_pcdptr, pc_addrel
JBC bit, rel	**opcode: 94**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc
S4	fetch, wr_temp2, port0_temp2
S5	inc_pc, temp1_srama, gba
S6	wr_tmp1, rd_comp, sram_tmp1
S7*	rd_alu_bit, jb
S8	temp2_pcdptr, pc_addrel
S9	clrb, rd_bit_pos
S10	rd_alu_low, wsd, alul_sram
S11	wr_comp
ACALL	**opcode: 95**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc, sp_tmp1, wr_tmp1
S4	inc
S5	rd_alu_low, lda, alul_srama, pcl_sram, wsd
S6	wr_comp
S7	rd_alu_low, wr_tmp1, alul_tmp1
S8	inc
S9	rd_alu_low, lda, alul_srama, pch_sram, wsd
S10	wr_comp
S11	temp1_pcdptr, absolute
S12	rd_alu_low, wr_sp, alul_sp

LCALL	opcode: 96
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc
S4	fetch, wr_temp2, port0_temp2
S5	inc_pc, sp_tmp1, wr_tmp1
S6	inc
S7	rd_alu_low, lda, alul_srama, pcl_sram, wsd
S8	wr_comp, rd_alu_low, wr_tmp1, alul_tmp1
S9	inc
S10	rd_alu_low, lda, alul_srama, pch_sram, wsd, temp2_pcdptr, wr_pc_low
S11	wr_comp, temp1_pcdptr, wr_pc_high, rd_alu_low, wr_sp, alul_sp
RET	**opcode: 97**
S0	fetch, port0_tac
S1	inc_pc, sp_srama, lda
S2	rd_comp, wr_pc_high, sram_pcdptr, sp_tmp1, wr_tmp1
S3	dec
S4	rd_alu_low, lda, alul_srama
S5	rd_comp, wr_pc_low, sram_pcdptr, rd_alu_low, wr_tmp1, alul_tmp1
S6	dec
S7	rd_alu_low, wr_sp, alul_sp
RETI	**opcode: 98**
S0	fetch, port0_tac
S1	inc_pc, sp_srama, lda
S2	rd_comp, wr_pc_high, sram_pcdptr, sp_tmp1, wr_tmp1
S3	dec
S4	rd_alu_low, lda, alul_srama
S5	rd_comp, wr_pc_low, sram_pcdptr, rd_alu_low, wr_tmp1, alul_tmp1
S6	dec
S7	rd_alu_low, wr_sp, alul_sp
S8	reti
AJMP	**opcode: 99**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc, temp1_pcdptr, absolute

LJMP	opcode: 100
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc
S4	fetch, wr_temp2, port0_temp2
S5	temp1_pcdptr, wr_pc_high
S6	temp2_pcdptr, wr_pc_low
SJMP	**opcode: 101**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc, temp1_pcdptr, pc_addrel
JMP @A + DPTR	**opcode: 102**
S0	fetch, port0_tac
S1	inc_pc
S2	acc_pcdptr, pc_addadptr
JZ	**opcode: 103**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3*	inc_pc, rd_acc_zero, jb
S4	temp1_pcdptr, pc_addrel
JNZ	**opcode: 104**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3*	inc_pc, rd_acc_zero, jnb
S4	temp1_pcdptr, pc_addrel
CJNE A, direct, rel	**opcode: 105**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc
S4	fetch, wr_temp2, port0_temp2
S5	inc_pc, wr_tmp1, acc_tmp1
S6	lda, temp1_srama
S7	rd_comp, wr_tmp2, sram_tmp2
S8	subb, c_zero
S9	flag2
S10*	rd_alu_zero, jnb
S11	temp2_pcdptr, pc_addrel

CJNE A, #data, rel	opcode: 106
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_tmp2, port0_tmp2
S3	inc_pc
S4	fetch, wr_temp2, port0_temp2
S5	inc_pc, wr_tmp1, acc_tmp1
S6	subb, c_zero
S7	flag2
S8*	rd_alu_zero, jnb
S9	temp2_pcdptr, pc_addrel
CJNE Rn, #data, rel	**opcode: 107**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_tmp2, port0_tmp2
S3	inc_pc
S4	fetch, wr_temp2, port0_temp1
S5	inc_pc, rri, tbe
S6	wr_tmp1, rd_comp, sram_tmp1
S7	subb, c_zero
S8	flag2
S9*	rd_alu_zero, jnb
S10	temp2_pcdptr, pc_addrel
CJNE @Ri, #data, rel	**opcode: 108**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_tmp2, port0_tmp2
S3	inc_pc
S4	fetch, wr_temp2, port0_temp2
S5	inc_pc, rri, obe
S6	rd_comp, wr_temp1, sram_temp1
S7	lda, temp1_srama
S8	wr_tmp1, rd_comp, sram_tmp1
S9	subb, c_zero
S10	flag2
S11*	rd_alu_zero, jnb
S12	temp2_pcdptr, pc_addrel

DJNZ Rn, rel	opcode: 109
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, wr_temp1, port0_temp1
S3	inc_pc, rri, tbe
S4	wr_tmp1, rd_comp, sram_tmp1
S5	dec
S6	rd_alu_low, wsd, alul_sram
S7	wr_comp
S8*	rd_alu_zero, jnb
S9	temp1_pcdptr, pc_addrel
DJNZ direct, rel	**opcode: 110**
S0	fetch, port0_tac
S1	inc_pc
S2	fetch, lda, port0_srama
S3	inc_pc
S4	fetch, wr_temp1, port0_temp1
S5	inc_pc, wr_tmp1, rd_comp, sram_tmp1
S6	dec
S7	rd_alu_low, wsd, alul_sram
S8	wr_comp
S9*	rd_alu_zero, jnb
S10	temp1_pcdptr, pc_addrel

***Branch State**