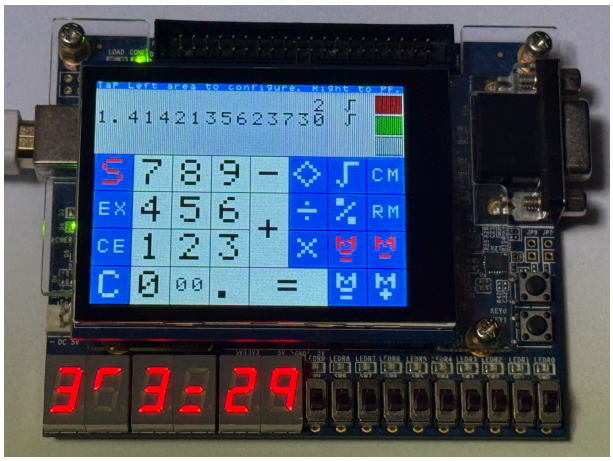
MCS-4 System Redesign

Technical Reference Manual

An ode to the Intel 4004 and Busicom 141-PF

Rev.01



Munetomo Maruyama

Revision History				Note
Rev	**Date**	**Author**	**Description**	
01	Jul.10 2025	MM	1st Release.	

Legal Notice

- Intel is a registered trade mark of the Intel Corporation
- This material is in no way affiliated with the Intel Corporation
- The opinions and statements expressed are my own.
- The Verilog HDL codes, software codes and engineering documents are provided "AS IS" with no warranty expressed or implied. Fitness for any particular purpose is not guaranteed. The authors do not accept any liability for use of this information.
- The works are licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 License.

Author

- Munetomo Maruyama
- X (Twitter) : @Processing_Unit
- GitHub : `https://github.com/munetomo-maruyama`

Contents

1 The Birth of the World's First Microcomputer **1**

1.1 The Birth of the World's First Microcomputer 1
1.2 Calculators Built with Discrete Components 1
1.3 The Move Toward LSI Integration 1
1.4 Busicom's Initiative 2
1.5 The Birth of the Microcomputer Concept 2
1.6 Development of the MCS-4 System 2
1.7 The Birth of the Busicom 141-PF Calculator Powered by the Microcomputer 2
1.8 Intel ' s Visionary Move 4
1.9 Japan's Contribution to the Birth of Microcomputers 4
1.10 Where to See the Intel 4004 and Busicom 141-PF Today 5
1.11 From 4-bit to 8-bit CPUs and Beyond 5

2 MCS-4 Chip Set and System **7**

2.1 Introduction 7
2.2 Getting Comfortable with 4 Bits 7
2.3 MCS-4 Chip Set 8
2.4 4004 CPU 8
2.5 4001 ROM 10
2.6 4002 RAM 12
2.7 4003 Shift Regieter 15
2.8 MCS-4 System Configuration 16
2.9 DC Characteristics of MCS-4 Chip Set 18

3 4004 CPU Instruction Set Architecture **19**

3.1 4004 CPU Programmer's Model 19
3.2 Instruction Timing of the CPU 20
3.3 Instruction Set Architecture and Opcode Tables 21
3.4 Types of Instruction Execution Timing 22
3.5 Instruction Fetch Operation from ROM 22
3.6 CPU Data Read Operation 27
3.7 CPU Data Write Operation 30
3.8 Subroutine Call Operation in the 4004 CPU 30
3.9 TEST Pin as an Alternative to Interrupts 31
3.10 Instructions Located at ROM Page Boundaries 32
3.11 Timing of PC Update 32
3.12 FIN Instruction and ROM Page Boundaries 32
3.13 Appreciating the Wisdom of the Pioneers 32

4 Development Tool for 4004 Assembler Program **35**
4.1 CPU Assembler Programming Tool ADS4004 35
4.2 ADS4004 Assembler Functionality 36
4.3 ADS4004 Disassembler Functionality 38
4.4 ADS4004 CPU Simulator Functionality 38
4.5 4004 CPU Sample Program (1): Memory Access Program – `memtst.src` . 42
4.6 4004 CPU Sample Program (2): BCD to Binary Conversion – `bcd2bin.src` 46

5 Logic Design of MCS-4 Chips **49**
5.1 Differences Between Physical Chip Design and RTL Design 49
5.2 Internal States of MCS-4 Chips and System Synchronization 49
5.3 Logic of the `MCS4_CPU` Module (4004) 51
5.4 Logic of the `MCS4_ROM` Module (4001) 66
5.5 Logic of the `MCS4_RAM` Module (4002) 69
5.6 Logic of the `MCS4_SHIFTER` Module (4003) 71

6 MCS-4 System and Busicom 141-PF **73**
6.1 Overview of the designed MCS-4 System 74
6.2 FPGA Board and Optional Components Used 75
6.3 Overview of the FPGA Top-Level Logic 76
6.4 Logic Description of MCS-4 Related Modules 79
6.5 Program for the 4004 Calculator Model 141-PF 86
6.6 RISC-V Subsystem 87
6.7 Logical Simulation of the MCS-4 System 92
6.8 Logical Simulation of the Entire FPGA System 94
6.9 FPGA Implementation 95
6.10 Key Operation Method of the 141-PF Calculator 98
6.11 The Charming Print Method of the 141-PF Calculator 100
6.12 Operation Method Using a Serial Terminal in This FPGA System 101
6.13 Operation Method Using the Touch LCD Shield in This FPGA System . . 103

7 A challenge: Calculating 500 Digits of Pi with the 4004 CPU **105**
7.1 Principle of Pi Calculation 105
7.2 Pi Calculation Algorithm 106
7.3 RAM Data Assignment 107
7.4 Source Program for Pi Computation 107
7.5 Printing the Result with 141-PF Printer 107

8 Conclusion **109**

113

List of Figures

1.1 The world ' s first microprocessor: the 4004 . . . 3
1.2 MCS-4 Chip Set[2] . . . 3
1.3 Die Photo of the 4004 chip[3] . . . 3
1.4 Busicom's 141-PF Calculator[4] . . . 4
1.5 Intel Museum (California, Santa Clara)[5] . . . 5
1.6 NEC ' s TK-80 single-board kit[6] . . . 5

2.1 Pinout of 4004 CPU chip . . . 10
2.2 Pinout of 4001 ROM chip . . . 11
2.3 Structure of 4001 ROM chip . . . 12
2.4 Pinout of 4002 RAM chip . . . 13
2.5 Structure of 4002 RAM chip . . . 14
2.6 Pinout of 4003 Shift Regisster chip . . . 15
2.7 Structure of 4003 Shift Register chip . . . 16
2.8 Example Memory Configuration of the MCS-4 System . . . 17
2.9 Example MCS-4 System Configuration (2) . . . 18

3.1 4004 CPU Programmers Model . . . 20
3.2 4004 CPU Instruction Timing . . . 21
3.3 4004 CPU Instruction Fetch . . . 27
3.4 4004 CPU Data Read Operation . . . 28
3.5 RAM Bank Expansion and DCL Decoder . . . 29
3.6 4004 CPU Data Write Operation . . . 30
3.7 4004 CPU Subroutine Call . . . 31

5.1 System synchronization method using `SYNC` signal . . . 50

6.1 Key Board and Switches on Calculator 141-PF . . . 73
6.2 Block Diagram of MCS-4 System . . . 74
6.3 Terasic DE10-Lite Board . . . 76
6.4 Adafruit 2.8" TFT Touch Shield for Arduino with Capacitive Touch (Product ID: 1947) . . . 77
6.5 Adafruit 2.8" TFT Touch Shield for Arduino with Resistive Touch Screen (Product ID: 1651) . . . 77
6.6 I/O Circuit of the Busicom 141-PF Calculator via MCS-4 System . . . 80
6.7 Printing Mechanism of the Calculator . . . 82
6.8 Waveform of 141-PF Simulation . . . 93
6.9 Waveform of FPGA Simulation . . . 94
6.10 Case A: MCS4_CPU (4004 CPU) Connection . . . 97
6.11 Case B: MCS4_CPU (4004 CPU) Connection . . . 97
6.12 Case C: MCS4_CPU (4004 CPU) Connection . . . 98

6.13 Connection of RISC-V USB-JTAG Interface Board 99
6.14 An example of schematics of RISC-V USB-JTAG Interface Board 99
6.15 How to use the 141-PF (Normal Calculation) 100
6.16 How to use the 141-PF (Memory Calculation) 101
6.17 Character Display on 7 segment LED 102
6.18 Operation on Touch LCD Panel 104

List of Tables

2.1 Overview of MCS-4 Chip Set Specifications 9
2.2 4002 RAM Chip Selection by P0 and SRC Address Bits 14
2.3 DC Characteristics of P-MOS Devices (MCS-4 Chipset) 18

3.1 CPU Instruciton Table (1) : Machine Instruction (1 word) 23
3.2 CPU Instruciton Table (2) : Machine Instruction (2 word) 24
3.3 CPU Instruciton Table (3) : Data Access 25
3.4 CPU Instruciton Table (4) : Accumulator 26
3.5 Correspondence between DCL values and $\overline{\text{CM-RAMx}}$ signals 28
3.6 SRC Address Structure 29

4.1 ADS4004 Assembler Syntax 39
4.2 MCS-4 System Configuration for ADS4004 Simulation 40
4.3 ADS4004 Simulator Command List 41

5.1 I/O Signals of `MCS4_CPU` 51
5.2 I/O Signals of `MCS4_ROM` 66
5.3 Assumed Metal Option Settings in `MCS4_ROM` 68
5.4 I/O Signals of `MCS4_RAM` 69
5.5 Decode of CM_RAM_N[3:0] 70
5.6 I/O Signals of `MCS4_SHIFTER` 71

6.1 MCS-4 System Specification 75
6.2 I/O Signals of `MCS4_SYS` 78
6.3 I/O Signals of `KEY_PRINTER` 78
6.4 Connections of ROM/RAM I/O Ports in the Calculator I/O Circuit 79
6.5 Connection Method of 4003 Registers Within the Calculator I/O Circuit . 80
6.6 Keyboard Matrix States and Meanings of the Calculator 81
6.7 Rotating Drum of the Printer 83
6.8 Printing Sequence Example of the Calculator's Printer 83
6.9 Command Signals Sent to `KEY_PRINTER` (Connected to RISC-V Subsystem Ports) 84
6.10 Response Signals from `KEY_PRINTER` (Connected to RISC-V Subsystem Ports) 85
6.11 Registers in the PORT Module 87
6.12 GPIO0 connected to External Pins 88
6.13 GPIO1 connected to External Pins 89
6.14 GPIO2 connected to External Pins 89
6.15 GPIO3 connected to Internal Signals 90
6.16 GPIO4 connected to Internal Signals 90
6.17 GPIO5 connected to Internal Signals 91

6.18 RISC-V related External Signals except for GPIO 96
6.19 MCS4_CPU (4004 CPU) related External Signals 96
6.20 Operation of Key Input on Serial Terminal 102

7.1 Values of `T1[]`, `T2[]`, `T3[]`, and `PI[]` for Each n 107

Listings

4.1 Assembly Source Code Example – `sample.src` . . . 36
4.2 Memory Access Program – `memtst.src` . . . 42
4.3 BCD to Binary Conversion – `bcd2bin.src` . . . 46
5.1 Definition of System State . . . 51
5.2 Definition of input/output signals and internal signals . . . 51
5.3 Control of Internal State Transitions . . . 52
5.4 Control for multi-cycle instructions . . . 52
5.5 Generation of the `sync_n` signal . . . 53
5.6 Program Counter (PC) and the Stack (Program Address Register) . . . 53
5.7 Instruction Fetch Processing . . . 53
5.8 Determining the next PC value . . . 54
5.9 Arithmetic Logic Unit (ALU) operations . . . 54
5.10 KBP instruction table conversion . . . 55
5.11 Processing for accumulator ACC . . . 56
5.12 Processing for CY (carry) bit . . . 56
5.13 Processing for index registers Rn and index register pair RnP . . . 56
5.14 Processing of DCL (Designate Command Line) . . . 57
5.15 Processing of SRC (Send Register Control) . . . 57
5.16 Generation of **cm_rom_n** signal . . . 58
5.17 Generation of **cm_ram_n[3:0]** signal . . . 58
5.18 Generation of data bus output **data_o[3:0]** . . . 58
5.19 Handling of **TEST** input signal . . . 59
5.20 Generation of condition signals for the JCN (Jump Conditional) instruction 59
5.21 Generation of DAA instruction correction conditions . . . 59
5.22 Instruction control unit . . . 60
5.23 End Module . . . 66

Chapter 1

The Birth of the World's First Microcomputer

This chapter describes the history of MCS-4 system.

1.1 The Birth of the World's First Microcomputer

Minicomputers were built using multiple IC chips to construct the processor (CPU). However, in 1971, the year following the release of the historically significant PDP-11, the world ' s first single chip microprocessor, the 4004 was born. This section provides an overview of its development. A Japanese engineers played a significant role in making this breakthrough a reality.[1]

1.2 Calculators Built with Discrete Components

From the 1960s to the 1970s, calculator manufacturers around the world fiercely competed to develop new products. Initially, these machines were assembled using discrete components such as TTL ICs, and their logic circuits were entirely constructed using combinational logic. However, as customer demands prompted frequent specification changes, manufacturers began shifting to a stored program approach that embedded macro instructions into ROM like circuits and processed operations sequentially. This architectural transition laid the groundwork for what would eventually become microcomputers. That said, each of these macro instructions represented a large, calculator specific function. Fetching a single instruction typically invoked a substantial sequence (such as waiting for key input, printing on paper, or performing multi digit addition).

1.3 The Move Toward LSI Integration

In 1969, the same year Apollo 11 landed humans on the moon, Sharp announced the QT-8D, the world ' s first eight-digit calculator using MOS LSI technology. Developed in collaboration with Rockwell in the U.S., this four chip configuration achieved both significantly reduced power consumption and a dramatic drop in component count. From this point on, many calculator makers began shifting their focus toward LSI development.

1.4 Busicom's Initiative

Busicom Corporation was one of Japan's prominent calculator manufacturers at the time. In 1969, Busicom began exploring LSI development for calculators and chose Intel, a U.S. company that specialized in producing LSI using the PMOS process, as a development partner. At the time, Intel was primarily a memory company, manufacturing DRAM and PROM.

1.5 The Birth of the Microcomputer Concept

The leading figure behind calculator LSI development at Busicom was Masatoshi Shima, a Japanese engineer. In June 1969, he traveled to the U.S. and initiated discussions on calculator LSI specifications at Intel. Initially, the idea was to implement calculator functions using macro instructions specific to calculators. However, the growing complexity of the LSI led to concerns about cost. At one point, Intel engineers proposed abandoning macro instructions in favor of microinstructions. These were simpler and more general purpose than calculator specific macros, closely resembling the instructions used in modern microcomputer CPUs This proposal drastically simplified the hardware, making it a cost effective solution. Thus, the idea of a 4-bit CPU, the 4004, as a microcomputer architecture was born (Figure 1.1). Although general purpose CPUs already existed in the form of minicomputers like DEC's PDP-8, it is likely that without the intense technical discussions between Busicom and Intel around macro instruction systems, the concept of an LSI based microcomputer (i.e., a 4-bit CPU) would never have emerged.

1.6 Development of the MCS-4 System

To realize Busicom's envisioned calculator system, development expanded beyond the 4004 CPU to include the 4001 ROM, 4002 RAM, and 4003 shift register (Figure 1.2). Design efforts for these chips took into account the 16 pin DIP packaging standard from Intel's DRAM products. By December 1969, the overall specifications were largely finalized. Next came the actual LSI design phase. As the client, Shima himself was incorporated into the team as a logic designer and participated directly in development. Samples of the 4001 and 4003 were completed in October 1970, followed by the 4002 in November, and finally the 4004 in March 1971. Fabricated using a 10 μ m PMOS process, the 4004 packed 2,300 transistors (roughly equivalent to 600 gates) into a 2mm x 3mm chip (Figure 1.3). Its operating frequency was 740 kHz.

1.7 The Birth of the Busicom 141-PF Calculator Powered by the Microcomputer

By April 1971, all the chips had been mounted on the calculator's circuit board, and astonishingly, the system worked perfectly on the first try. This calculator (Figure 1.4), released as the 141-PF in 1972, was a 15-digit model with a two color printer. It was already complete as a calculator product, featuring memory functions, percent calculations, and optional square root operations.

[1]Source: `http://www.intel-vintage.info/intelmcs.htm`
[2]Owned by the author.
[3]Source: `(https://en.wikichip.org/wiki/intel/mcs-4/4004`

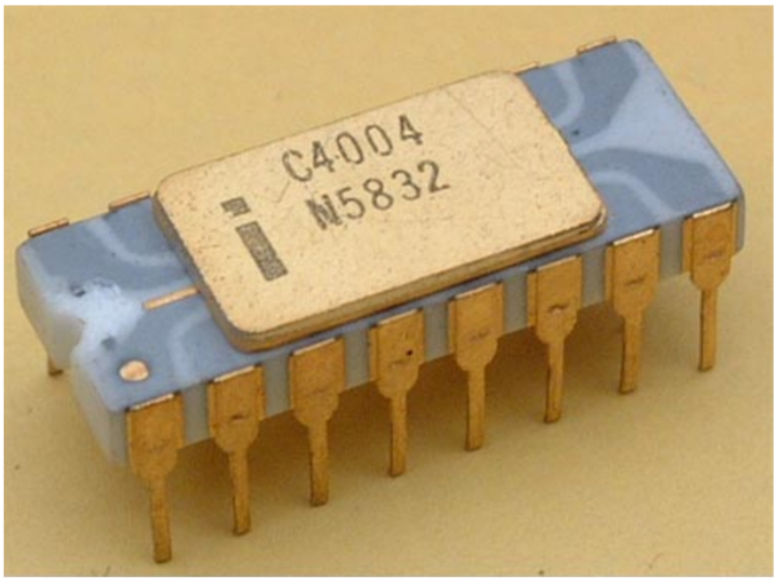


Figure 1.1: The world's first microprocessor: the 4004[1]

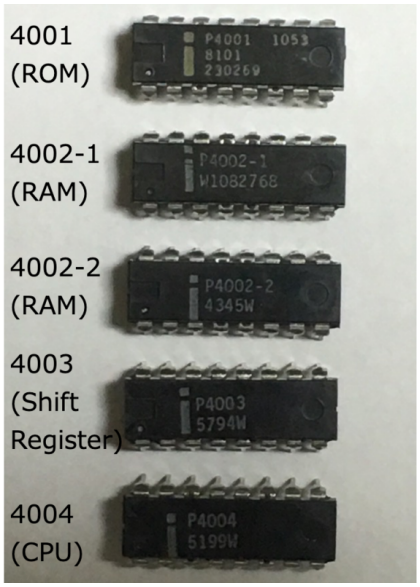


Figure 1.2: MCS-4 Chip Set[2]

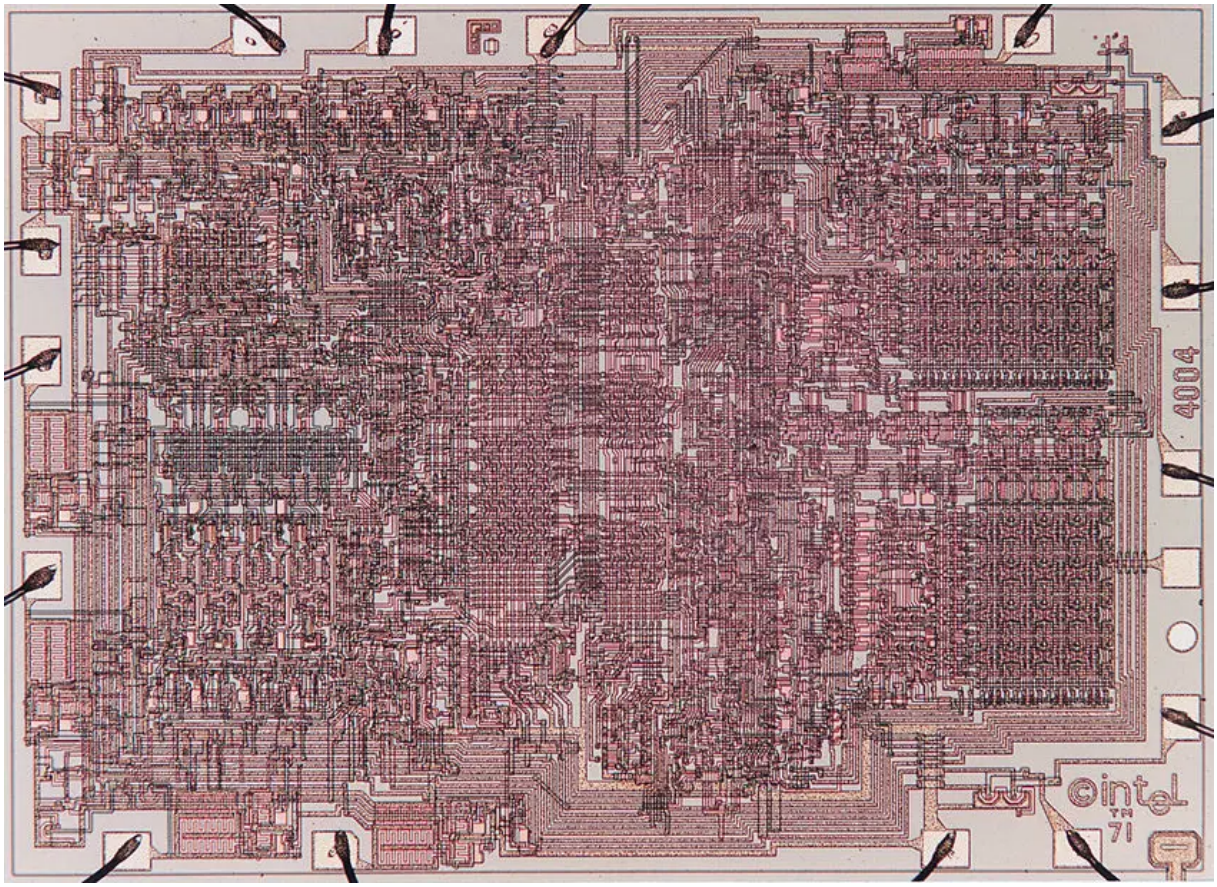


Figure 1.3: Die Photo of the 4004 chip[3]

Figure 1.4: Busicom's 141-PF Calculator[4]

1.8 Intel's Visionary Move

Initially, the 4001–4004 chipset was developed as a custom product exclusively for Busicom. However, recognizing the general purpose potential of the chips, and aligning with Busicom's need for capital, Intel renegotiated the contract, returning part of the payment in exchange for the right to sell the chips independently. Intel branded the chip set as the MCS-4 (Micro Computer Set-4) and began sales in November 1971. This marked the beginning of Intel's journey as a CPU manufacturer.

1.9 Japan's Contribution to the Birth of Microcomputers

Shima's role in the context of calculator application development was extremely significant. Among the many engineers involved in the joint Busicom–Intel project, his contributions to the motivation behind conceiving a general-purpose microcomputer are especially noteworthy. At a time when LSI design tools were nonexistent, logical circuits had to be painstakingly drawn by hand at the transistor level. The excellence of technical execution and the determination to complete a comprehensive system, including the calculator application itself, were truly remarkable.

[4]Source: Dentaku-Museum — `http://www.dentaku-museum.com/calc/calc/10-busicom/busicomd/busicomd.html`

Figure 1.5: Intel Museum (California, Santa Clara)[5]

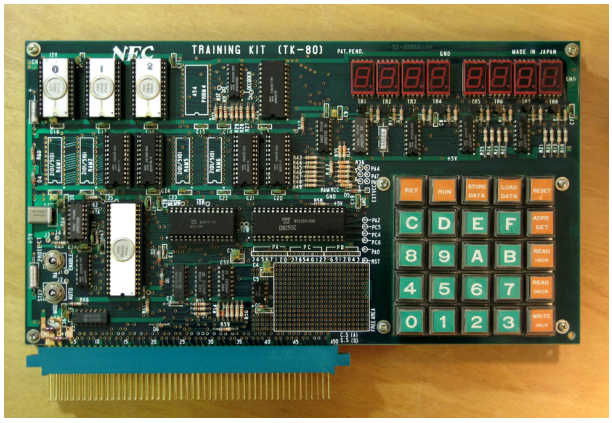


Figure 1.6: NEC' s TK-80 single-board kit[6]

1.10 Where to See the Intel 4004 and Busicom 141-PF Today

You can view the actual Intel 4004 and Busicom 141-PF units at the Intel Museum in Silicon Valley, U.S.A (Figure 1.5).

1.11 From 4-bit to 8-bit CPUs and Beyond

Decimal based calculators naturally paired well with 4-bit CPUs like the 4004. However, as systems started handling characters (e.g., ASCII), 8-bit processing became necessary. Following the 4004, Intel developed the 8008 and 8080, the 8- bit CPUs that went on to become major commercial successes. The author recalls being captivated by the charm of microcomputers through NEC ' s TK-80 single-board kit (Figure 1.6), which featured the 8080A (an enhanced version of the original 8080 with improved drive strength). From there, microcomputing evolved rapidly from 16-bit to 64-bit CPUs and ever more powerful architectures. The pace of progress, as we now know, has been astounding.

[5]The author visited here in July 2011.
[6]`https://en.wikipedia.org/wiki/TK-80`

Chapter 2

MCS-4 Chip Set and System

This chapter describes the MCS-4 Chip Set and System Architecture.

2.1 Introduction

This book aims to logically design the Intel 4004 (CPU), the centerpiece of the MCS-4 system, the world's first microcomputer, using Verilog HDL, implement it on an FPGA, and ultimately recreate the historic Busicom 141-PF calculator. In this chapter, we begin with a detailed explanation of the overall architecture of the MCS-4 system, starting with the 4004.[2]

2.2 Getting Comfortable with 4 Bits

Since the 4004 is a 4-bit CPU, data is fundamentally handled in 4-bit units. Accordingly, data addresses are also assigned in 4-bit increments. Those of you who grew up with microcomputers might be more familiar with 8-bit data and addressing, so working with 4-bit units might feel a bit strange at first. But I encourage you to embrace that sensation-find enjoyment in the unfamiliar. Note that instruction codes of 4004 are 8-bit, so their addresses are assigned in 8-bit units.

In the world of data, 8 bits make up a "byte", and likewise, 4 bits are referred to as a "nibble". This chapter uses the term nibble frequently, so take some time to get comfortable with it.

Incidentally, one theory behind the term "byte" is that it's a playful twist on the word "bite", as in a bite of data. The word "nibble", fittingly, means "to take a small bite". To add another layer of trivia: "data" is the plural form of "datum", which comes from the Latin word "dare", meaning "to give". In short, when something is given to you, you should chew or nibble thoughtfully.

Just don't expect much excitement from the word "bit"—it's short for "binary digit", and that etymology is as dry as it sounds.

While a byte is commonly understood today to be 8 bits, that hasn't always been the case. Historically, "byte" simply referred to the number of bits needed to encode a character, ranging anywhere from 5 to 12 bits (according to Wikipedia). These days, 8

bits are generally referred to as a byte, but if you want to be exact, the proper term for an 8-bit unit is “octet”. Similarly, a “nibble” precisely represents 4 bits.

2.3 MCS-4 Chip Set

To build a 4-bit microcomputer system, four types of chips are used: the 4004 (CPU), 4001 (ROM), 4002 (RAM), and 4003 (Shift Register). Typically, only one 4004 exists within the system, while multiple units of the 4001, 4002, and 4003 are used depending on the configuration. An overview of their specifications is shown in Table 2.1.

The 4001 functions as a mask ROM with input/output ports, the 4002 serves as RAM with output ports, the 4003 is a shift register for expanding output ports, and the 4004 is the 4-bit CPU. There are two variants of the 4002 RAM chip: the 4002-1 and the 4002-2. These are provided to handle differences in how chip numbers are assigned within RAM banks.

2.4 4004 CPU

2.4.1 Functional Overview of the 4004 CPU

The 4004 is a 4-bit CPU core at the heart of the MCS-4 system. It fetches instructions from ROM, decodes the instruction code, and executes operations accordingly.

Each instruction cycle consists of 8 clock cycles, during which the CPU transitions sequentially through eight internal states: A1, A2, A3, M1, M2, X1, X2, and X3.

With a maximum operating frequency of 740 kHz, each instruction cycle lasts approximately 10.8 µs (minimum). Naturally, the 4004 does not utilize pipelined control; instruction fetch, decode, and execution are processed one at a time in strict sequence.

Detailed timing of instruction execution and behavior of individual instructions will be described later.

2.4.2 Pin layout and Signal Functions of the 4004 CPU

Figure 2.1 shows the pin layout and the signal functions of the 4004 CPU. The role of each signal is described below:

(1) Φ1, Φ2 (Clock Inputs): The internal logic employs dynamic latch circuits, requiring a two-phase clock input. In this documentation ’s redesign of the MCS-4, all circuits use D flip-flops and a single-phase clock.

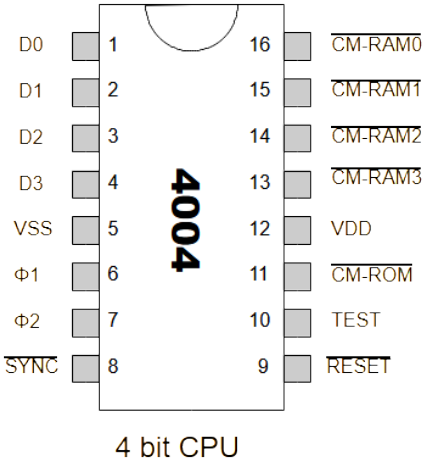
(2) $\overline{\text{RESET}}$ (Reset Input): A signal used to reset the internal logic of the CPU.

(3) D0–D3 (Data Bus): A 4-bit bidirectional data bus. The 4004 does not have a separate address bus; address information is also carried via the data bus, with time-division multiplexing used to exchange address and data between the CPU and ROM/RAM.

(4) $\overline{\text{SYNC}}$ (Synchronization Signal Output): This signal ensures synchronized operation among the CPU, ROM, and RAM so their internal states transition simultaneously. Each chip monitors the SYNC signal from the CPU to align its internal state progression.

Type	Name	Item	Details
CPU	4004	Function	4-bit CPU
		Instruction Set	46 instructions
		Instruction Length	8-bit or 16-bit
		Memory Space	ROM: 4KB; RAM: 1280 nibbles (direct), 2560 nibbles (with external circuits)
		Operating Frequency	Up to 740.7KHz (1.35 µs)
		Instruction Cycle	Minimum 10.8 µs
		Package	16-pin DIP
ROM	4001	Function	Mask ROM and I/O port
		ROM Capacity	256 words × 8 bits (256 bytes per chip)
		Chips per System	Up to 16 chips (chip number set via metal options)
		I/O Ports	One 4-bit port per chip (I/O direction and pull-up/down set via metal options)
		Package	16-pin DIP
RAM	4002	Function	RAM and output port
		RAM Capacity	4 registers per chip; 16 nibbles (main) + 4 nibbles (status) per register; total 320 bits per chip
		Banks per System	Max 4 banks (direct); Max 8 banks (external circuit)
		Chips per Bank	Max 4 chips per bank; 4002-1 for chips #0 and #1, 4002-2 for #2 and #3
		Output Port	One 4-bit output port per chip
		Package	16-pin DIP
Shifter	4003	Function	10-bit shift register (serial to parallel conversion)
		Cascade Capability	Supported
		Reset	Power-on reset clears shift register
		Output Control	OE=1: outputs data; OE=0: outputs all zero
		Package	16-pin DIP

Table 2.1: Overview of MCS-4 Chip Set Specifications



Pin No.	Name	In/Out	Pin Function	Note
1	D0	In/Out	Data Bus (LSB)	
2	D1	In/Out	Data Bus	
3	D2	In/Out	Data Bus	
4	D3	In/Out	Data Bus (MSB)	
5	VSS	Power	GND	
6	Φ1	In	Clock Phase 1	
7	Φ1	In	Clock Phase 2	
8	$\overline{\text{SYNC}}$	Out	Sync Output	
9	$\overline{\text{RESET}}$	In	Reset	
10	TEST	In	Test Condition Input	JCN Instruction
11	$\overline{\text{CM-ROM}}$	Out	Command Control for ROM	
12	VDD	Power	VDD (-15V)	
13	$\overline{\text{CM-RAM3}}$	Out	Command Control for RAM 3	DCL Instruction
14	$\overline{\text{CM-RAM2}}$	Out	Command Control for RAM 2	DCL Instruction
15	$\overline{\text{CM-RAM1}}$	Out	Command Control for RAM 1	DCL Instruction
16	$\overline{\text{CM-RAM0}}$	Out	Command Control for RAM 0	DCL Instruction

Figure 2.1: Pinout of 4004 CPU chip

(5) $\overline{\text{CM-ROM}}$ (ROM Command Control Output): A signal indicating how the ROM should interpret the data on the bus during each internal state.

(6) $\overline{\text{CM-RAM0}}$ to $\overline{\text{CM-RAM3}}$ (RAM Command Control Outputs): Signals indicating how each RAM chip should interpret the data bus contents during each internal state.

(7) TEST (Conditional Branch Input Signal): An input used to evaluate branch conditions in the JCN (Jump Conditional) instruction.

2.5 4001 ROM

2.5.1 Functional Overview of the 4001 ROM

The 4001 is a ROM designed to store programs for the MCS-4 system. Each ROM chip has a capacity of 256 words by 8 bits. Additionally, the 4001 includes input/output port functionality.

The 4004 CPU can be directly connected to up to 16 ROM chips (a total of 4 KB). It is also possible to extend this capacity further with additional external circuitry.

This is a mask ROM: the ROM code must be submitted upon ordering, and the ROM pattern is permanently burned into the chip during manufacturing.

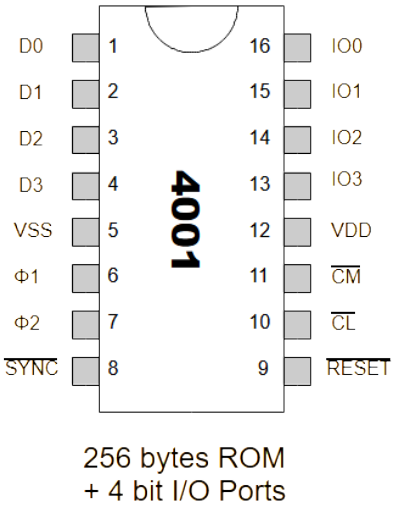
2.5.2 Pin layout and Signal Functions of the 4001 ROM

Figure 2.2 shows the pin layout and the signal functions of the 4001 ROM. The role of each signal is described below:

(1) Φ1, Φ2 (Clock Inputs): Two-phase clock inputs. The same clock used by the 4004 CPU is applied here.

(2) $\overline{\text{RESET}}$ (Reset Input): A signal used to reset the internal logic.

(3) D0–D3 (Data Bus): A 4-bit bidirectional data bus.



Pin No.	Name	In/Out	Pin Function	Note
1	D0	In/Out	Data Bus (LSB)	
2	D1	In/Out	Data Bus	
3	D2	In/Out	Data Bus	
4	D3	In/Out	Data Bus (MSB)	
5	VSS	Power	GND	
6	Φ1	In	Clock Phase 1	
7	Φ1	In	Clock Phase 2	
8	$\overline{\text{SYNC}}$	In	Sync Input	
9	$\overline{\text{RESET}}$	In	Reset	
10	$\overline{\text{CL}}$	In	Clear Input for I/O Ports	
11	$\overline{\text{CM}}$	In	Command Control Input	Connect $\overline{\text{CM-ROM}}$
12	VDD	Power	VDD (-15V)	
13	IO3	In/Out	I/O Port (MSB)	
14	IO2	In/Out	I/O Port	
15	IO1	In/Out	I/O Port	
16	IO0	In/Out	I/O Port (LSB)	

Figure 2.2: Pinout of 4001 ROM chip

(4) $\overline{\text{SYNC}}$ (Synchronization Input): An input signal used to synchronize operation with the CPU.

(5) $\overline{\text{CM}}$ (Command Control Input): This terminal connects to the $\overline{\text{CM-ROM}}$ signal generated by the CPU. The ROM interprets the data on the data bus based on the internal state in which the $\overline{\text{CM}}$ signal is asserted.

(6) IO0–IO3 (Input/Output Ports): A 4-bit I/O port. Unlike modern microcontrollers that configure port direction and pull-up/down resistors via software registers, the 4001's configuration is fixed during chip fabrication via metal options.

(7) $\overline{\text{CL}}$ (Register Clear Input for Output Ports): An input signal used to clear the output port's data register (flip-flops) to zero.

2.5.3 Internal Architecture of the 4001 ROM

The internal structure of the 4001 is shown in Figure 3.

Eeach 4001 chip contains ROM memory organized into 256 words by 8 bits.

When the CPU accesses data stored in ROM (the access method will be explained later), a single word (8 bits) is read over a 4-bit data bus. The ROM outputs the upper 4 bits (OPR: operation code) and the lower 4 bits (OPA: modifier) sequentially.

Additionally, the chip includes a 4-bit input/output port. The configuration of each port (input/output direction, presence of pull-up/pull-down resistors) is specified at the time of chip fabrication using metal options (M1 through M10), as illustrated in Figure 3(b).

The CPU provides dedicated instructions to read from or write to these I/O ports.

When placing an order for the 4001 chip, users must submit both the ROM pattern and their desired metal option configuration for the I/O ports.

2.5.4 CPU Addressing of the 4001 ROM

When the CPU accesses the 4001 ROM, it uses a 12-bit address space covering up to 4 KB. The upper 4 bits of the address serve as the chip number.

Each 4001 chip must be assigned a unique 4-bit chip number (ranging from 0 to 15), which is also specified as a metal option at the time of ordering.

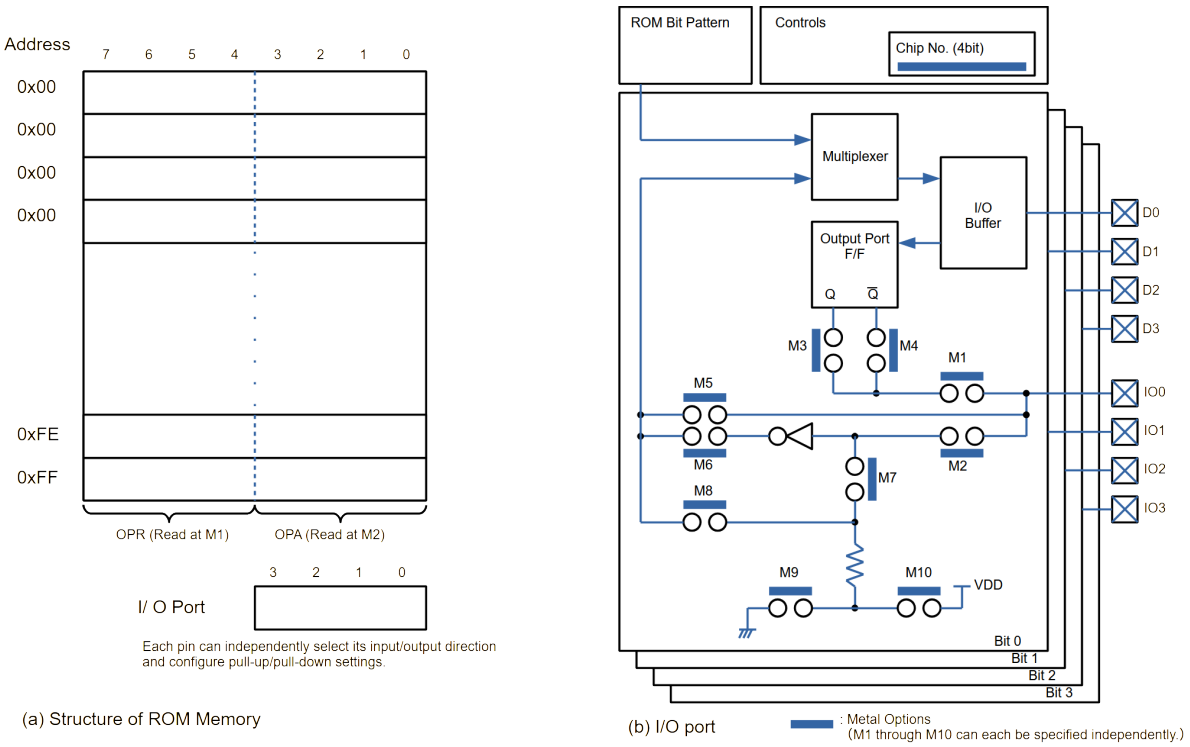


Figure 2.3: Structure of 4001 ROM chip

2.6 4002 RAM

2.6.1 Functional Overview of the 4002 RAM

The 4002 is a RAM chip used for storing temporary data. Each 4002 chip contains 320 bits (80 nibbles) of RAM. While the capacity may seem peculiar, the reason behind it will be discussed later.

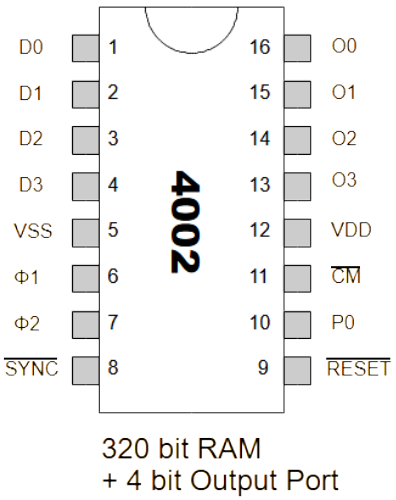
In addition, the 4002 also includes a 4-bit wide output port. Please note that the port direction is output-only.

From the CPU's perspective, the RAM chip group is organized hierarchically. The uppermost layer is called the *bank*, and the subordinate layer consists of individual chips. The signals $\overline{\text{CM-RAM0}}$ to $\overline{\text{CM-RAM3}}$ correspond to the bank layer, and the RAM chips within the bank that receives an asserted $\overline{\text{CM-RAMx}}$ signal are accessed accordingly.

Beneath the bank layer is the chip layer, with four 4002 RAM chips comprising a single bank.

When connecting RAM directly to the CPU without external circuitry, $\overline{\text{CM-RAM0}}$ to $\overline{\text{CM-RAM3}}$ are asserted using a one-hot method. In this configuration, the system supports up to four RAM banks, resulting in a total of 16 RAM chips (RAM capacity: 5120 bits = 1280 nibbles = 640 bytes).

If more than four RAM banks are desired within the system, the CPU can output encoded $\overline{\text{CM-RAM0}}$ to $\overline{\text{CM-RAM3}}$ signals. These signals are decoded externally to generate a corresponding set of one-hot $\overline{\text{CM-RAMx}}$ signals, each selecting a RAM bank. According to the CPU's instruction specifications, this encoding allows for a maximum of 8 selectable banks. In this extended configuration, the system accommodates up to 32 RAM chips (RAM capacity: 10,240 bits = 2560 nibbles = 1280 bytes).



Pin No.	Name	In/Out	Pin Function	Note
1	D0	In/Out	Data Bus (LSB)	
2	D1	In/Out	Data Bus	
3	D2	In/Out	Data Bus	
4	D3	In/Out	Data Bus (MSB)	
5	VSS	Power	GND	
6	Φ1	In	Clock Phase 1	
7	Φ1	In	Clock Phase 2	
8	$\overline{\text{SYNC}}$	In	Sync Input	
9	$\overline{\text{RESET}}$	In	Reset	
10	P0	In	Chip Select Condition Input	
11	$\overline{\text{CM}}$	In	Command Control Input	Connect $\overline{\text{CM-RAMx}}$
12	VDD	Power	VDD (-15V)	
13	O3	Out	Output Port (MSB)	
14	O2	Out	Output Port	
15	O1	Out	Output Port	
16	O0	Out	Output Port (LSB)	

Figure 2.4: Pinout of 4002 RAM chip

2.6.2 Pin layout and Signal Functions of the 4002 RAM

Figure 2.4 shows the pin layout and the signal functions of the 4002 RAM. The role of each signal is described below:

(1) Φ1, Φ2 (Clock Inputs): These are two-phase clock inputs. The same clock used by the 4004 CPU is applied.

(2) $\overline{\text{RESET}}$ (Reset Input): A signal used to reset the internal logic of the RAM. While $\overline{\text{RESET}}$ is asserted, the internal counter scans the RAM cells and clears its contents. Therefore, $\overline{\text{RESET}}$ must remain asserted for at least 32 instruction cycles, equivalent to 256 clock cycles.

(3) D0–D3 (Data Bus): A 4-bit bidirectional data bus.

(4) $\overline{\text{SYNC}}$ (Synchronization Input): An input signal used to synchronize the operation of the RAM with the CPU.

(5) $\overline{\text{CM}}$ (Command Control Input for RAM): This terminal connects to the $\overline{\text{CM-RAMx}}$ output from the CPU. The interpretation of data on the data bus depends on the internal state in which the $\overline{\text{CM}}$ signal is asserted.

(6) O0–O3 (Output Port): A 4-bit output-only port.

(7) P0 (Chip Select Condition Input): An input signal used to select the chip number of the 4002 RAM.

2.6.3 Internal Architecture of the 4002 RAM

The internal structure of the 4002 RAM is shown in Figure 2.5.

The 4002 RAM chip consists of four hierarchical blocks referred to as *registers*. Each register contains 16 main memory characters (nibbles, 64 bits) and 4 status characters (nibbles, 16 bits). All are accessible for read and write operations via CPU instructions.

Since one register comprises a total of 80 bits of RAM, and there are four registers, a single 4002 chip holds 320 bits of RAM in total—equivalent to 80 nibbles or 40 bytes.

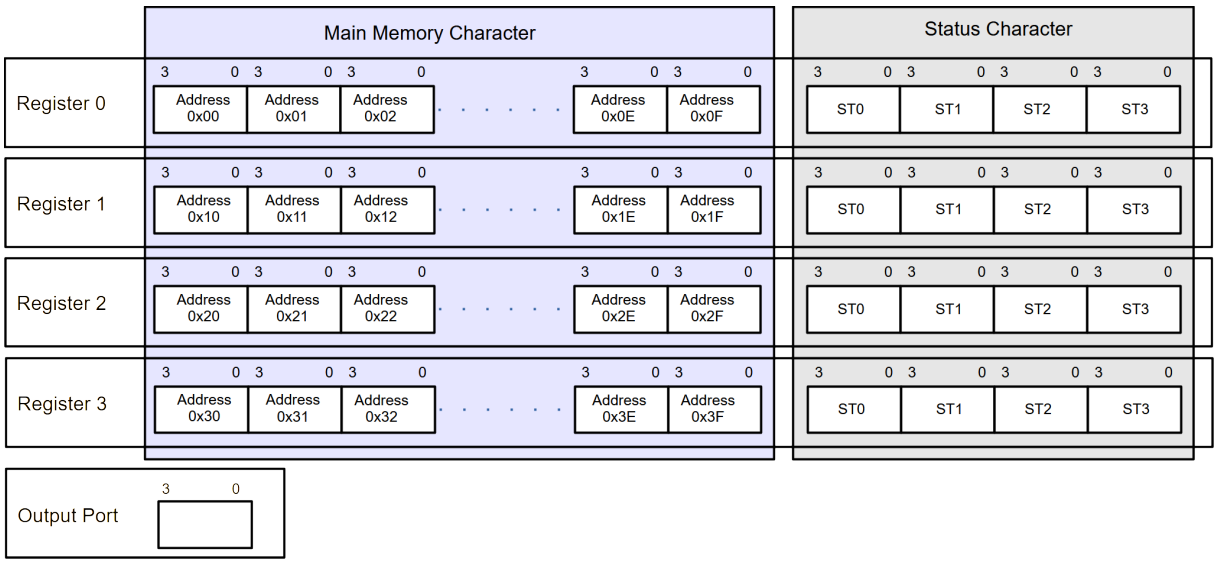


Figure 2.5: Structure of 4002 RAM chip

Within a single bank, the main memory character capacity reaches a maximum of 256 nibbles (128 bytes), while the status character capacity extends to 16 blocks, totaling 64 nibbles (32 bytes).

Remarkably, the RAM cells in the 4002 are implemented as dynamic memory. As such, periodic refresh is required. This refresh is performed automatically by an internal counter, which scans the RAM cells during idle periods within the instruction execution cycle (M1 and M2 states).

Additionally, the 4002 RAM includes a 4-bit output port. The CPU provides dedicated instructions for writing data to this port.

2.6.4 CPU Addressing of the 4002 RAM

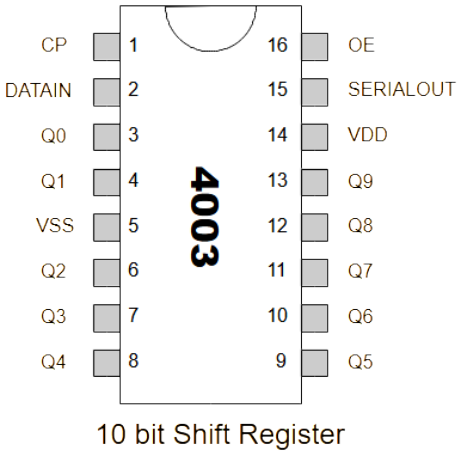
When addressing the 4002 RAM, the CPU specifies the following hierarchy: *bank* (4 to 8 units), *chip* (4 units), *register* (4 units), and either *main memory character* (16 units) or *status character* (4 units).

The bank is selected using the $\overline{\text{CM-RAMx}}$ signal. Lower hierarchical levels—chip, register, and character—are specified using 8-bit address information sent from the CPU to the RAM (known as the SRC address, described later).

Chip numbers must be hardware-assigned to each RAM chip in the system. Since there are four chips per bank, a 2-bit identifier is required for chip selection. One of these bits is designated via input pin `P0`. As there is no additional terminal available for the second bit, chip selection is achieved by differentiating between two types of RAM chips —`4002-1` and `4002-2`—as described in Table 2.2.

4002 Chip Type	P0 Input Level	Chip No. within Bank	SRC Address Bits 7 and 6
4002-1	GND	0	Selected when bits = `00`
4002-1	VDD	1	Selected when bits = `01`
4002-2	GND	2	Selected when bits = `10`
4002-2	VDD	3	Selected when bits = `11`

Table 2.2: 4002 RAM Chip Selection by P0 and SRC Address Bits



Pin No.	Name	In/Out	Pin Function	Note
1	CP	In	Shift Clock Input	
2	DATAIN	In	Serial Data Input	
3	Q0	Out	Parallel Data Output 0	
4	Q1	Out	Parallel Data Output 1	
5	VSS	Power	GND	
6	Q2	Out	Parallel Data Output 2	
7	Q3	Out	Parallel Data Output 3	
8	Q4	Out	Parallel Data Output 4	
9	Q5	Out	Parallel Data Output 5	
10	Q6	Out	Parallel Data Output 6	
11	Q7	Out	Parallel Data Output 7	
12	Q8	Out	Parallel Data Output 8	
13	Q9	Out	Parallel Data Output 9	
14	VDD	Power	VDD (-15V)	
15	SERIALOUT	Out	Serial Data Output	
16	OE	In	Parallel Data Output Enable	

Figure 2.6: Pinout of 4003 Shift Regisster chip

2.7 4003 Shift Regieter

2.7.1 Functional Overview of the 4003 Shift Register

The 4003 is a 10-bit shift register IC designed to perform serial-to-parallel data conversion. It also supports cascade connections, allowing for significant expansion of system output ports.

The 4003 is typically connected to the output ports of the 4001 ROM or the 4002 RAM for practical use in the MCS-4 system.

2.7.2 Pin layout and Signal Functions of the 4003 Shift Register

Figure 2.6 shows the pin layout and the signal functions of the 4003 Shift Register. The role of each signal is described below:

(1) CP (Clock Input): Clock signal for shift input. Data is shifted on the rising edge.

(2) DATAIN (Serial Data Input): Serial data input signal to the shift register.

(3) SERIALOUT (Serial Data Output): Serial data output signal from the shift register.

(4) Q0–Q9 (Parallel Outputs): Parallel outputs of the 10-bit shift register.

(5) OE (Output Enable): When OE = 1, the values stored in the shift register are output to Q0–Q9. When OE = 0, Q0–Q9 output all zeros.

2.7.3 Internal Architecture of the 4003 Shift Register

Figure 2.7 shows the internal configuration of the 4003, which implements a simple shift register.

The internal circuit of the CP clock input includes a delay line, allowing the serial data input signal `DATAIN` to change at the exact timing of the rising clock edge without requiring a setup time.

Similarly, the SERIALOUT signal also includes a delay circuit to ensure data hold time, thereby enabling cascade connections among multiple 4003 chips.

The shift register is cleared to zero via a power-on reset circuit.

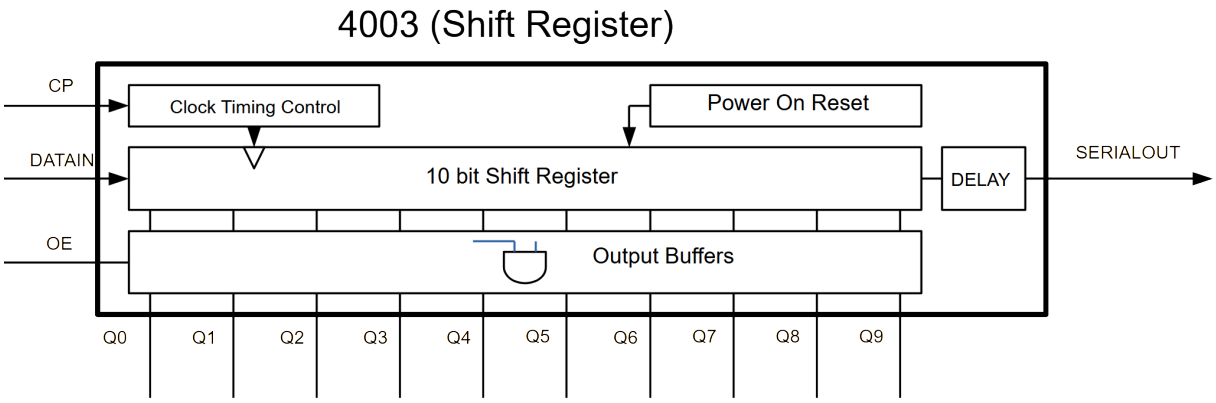


Figure 2.7: Structure of 4003 Shift Register chip

2.8 MCS-4 System Configuration

An example configuration of the MCS-4 system memory using the 4004 CPU, 4001 ROM, and 4002 RAM is shown in Figure 2.8.

This diagram illustrates the maximum system composition achieved by directly connecting each device to the CPU without any external circuitry.

It includes 16 ROM chips and 16 RAM chips spanning 4 banks.

2.8.1 Is Something Missing?

Carefully examine the system configuration in Figure 2.8. Is there anything missing for it to be a complete microcomputer system?

As previously mentioned in the 4004 CPU pin description, the system lacks a dedicated address bus. Instead, address information is transmitted via the 4-bit data bus using time-division multiplexing.

Beyond the absence of an address bus, consider other missing elements. For instance, the $\overline{\text{CM-ROM}}$ and $\overline{\text{CM-RAMx}}$ signals are fed into multiple ROM and RAM chips simultaneously. This suggests they cannot serve directly as individual chip select signals.

Additionally, there are no signals provided to designate read or write direction when accessing RAM or ports. Why is such a configuration still valid?

2.8.2 Fundamental Concept of Data Access Method

When the 4004 CPU accesses the ROM (I/O ports) or RAM (memory and output ports), it uses not a single instruction but a combination of instructions.

First, it executes an instruction to send address information, specifying the internal address of the target ROM or RAM chip. This address is stored internally, placing the addressed ROM or RAM in a ready state awaiting data access.

Next, the CPU executes a data access instruction. Data access instructions are prepared individually for each target and for each direction (read or write).

Here is the key point: When the CPU fetches a data access instruction, the ROM or RAM simultaneously monitors the fetched instruction. By doing so, they recognize

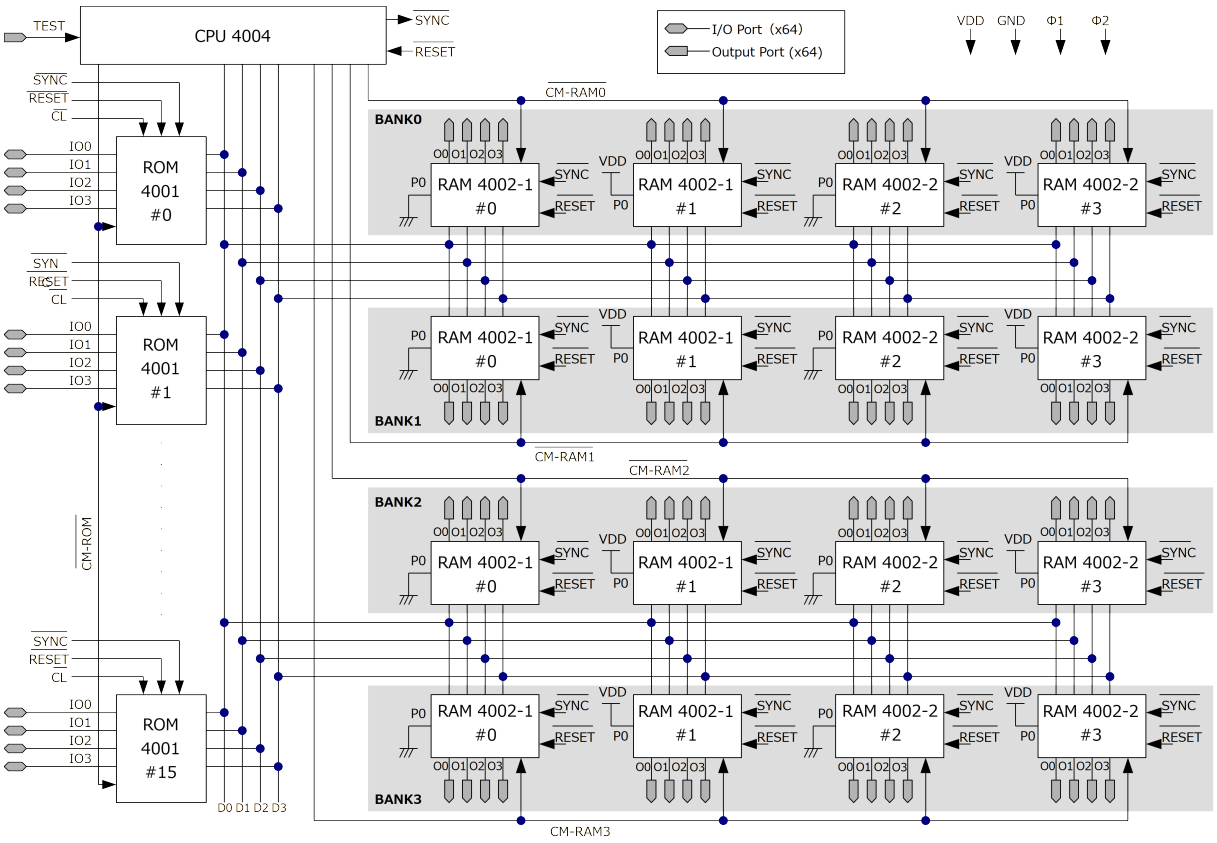


Figure 2.8: Example Memory Configuration of the MCS-4 System
This diagram illustrates a configuration of the 4004 CPU along with the 4001 ROM and 4002 RAM, showcasing a maximum system built through direct connection of each device to the CPU.

what type of access the CPU intends to perform. During the execution states of the same instruction cycle, the targeted memory devices act accordingly, performing read or write operations via the data bus.

Thus, the system does not require explicit read/write direction signals—memory chips infer access direction by interpreting the instruction type itself.

Details about each instruction's behavior during actual data access will be explained in subsequent sections.

2.8.3 Expansion of Output Ports in the MCS-4 System

In the system configuration shown in Figure 2.8, the 4001 ROM provides 64 I/O ports and the 4002 RAM offers 64 output ports.

While this yields a substantial number of ports, such memory capacity is typically excessive for applications like calculators.

To ensure sufficient output port availability even with fewer ROM/RAM chips, a serial-to-parallel conversion IC was developed—the 4003 shift register.

Figure 2.8 illustrates an example of output port expansion using the 4003. In this setup, one standalone 4003 and two cascade-connected 4003 chips expand 6 output lines into 30 individual ports.

Although shift operation introduces latency before output stabilization, it remains effective for human interface and similar applications.

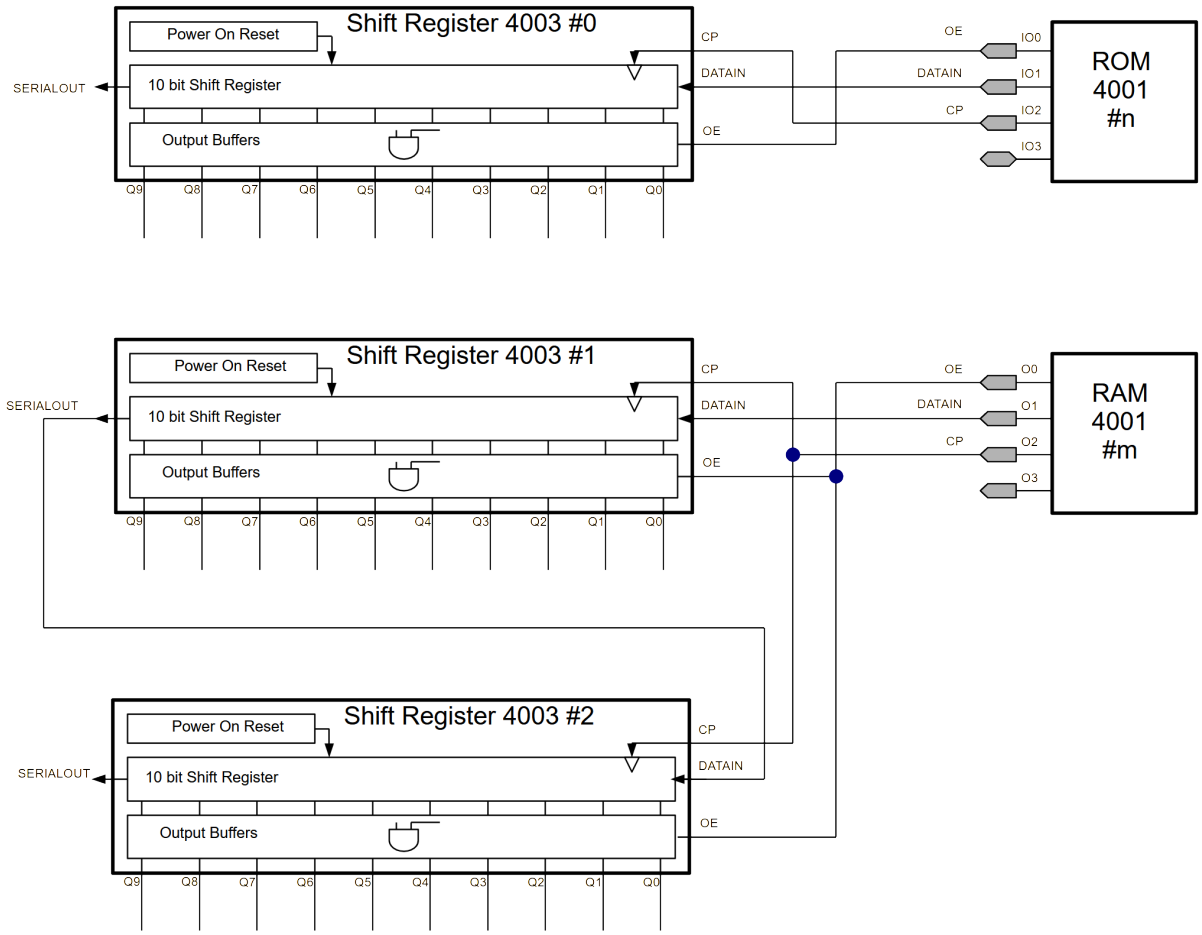


Figure 2.9: Example MCS-4 System Configuration (2)
A wiring example showing how 4003 shift registers are connected to expand output ports.

2.9 DC Characteristics of MCS-4 Chip Set

The actual MCS-4 chipset is manufactured using a 10 µm PMOS process.

Its DC characteristics are shown in Table 2.3. The power supply voltage `VDD` is a negative potential relative to the ground reference `VSS`.

Item	Symbol	Min	Max	Unit	Remarks
Power Supply	VDD	-15 -5%	-15 +5%	V	
Low-Level Input Voltage	VIL	VDD	VSS -5.5	V	Logic “1” level
Low-Level Output Voltage	VOL	VSS -12	VSS -6.5	V	
High-Level Input Voltage	VIH	VSS -1.5	VSS +0.3	V	Logic “0” level
High-Level Output Voltage	VOH	VSS -0.5	VSS	V	

Table 2.3: DC Characteristics of P-MOS Devices (MCS-4 Chipset)

Caution The MCS-4 system reproduced in this project is implemented on an FPGA. Under no circumstances should a `-15V` voltage be applied to the power supply or pins!

Chapter 3

4004 CPU Instruction Set Architecture

This chapter describes the detailed operation of the 4004 CPU, its specific interactions with ROM and RAM, and the specifications of its Instruction Set Archtecture.

3.1 4004 CPU Programmer's Model

Figure 3.1 illustrates the programmer's model of the 4004 CPU. Each resource is described below.

1. **Accumulator (ACC):** A 4-bit wide register used for arithmetic operations such as addition and subtraction, as well as for temporary data storage during ROM/RAM access.

2. **Carry Bit (CY):** Used for carry in/out during 4-bit addition and borrow in/out during subtraction. It is also utilized in rotate instructions and can serve as a condition flag for branching instructions.

3. **Index Register Group (R0–R15, R0P–R7P):** The CPU includes 16 x 4-bit index registers (R0 through R15). Even-numbered registers R[2n] serve as the upper 4 bits, and odd-numbered registers R[2n+1] serve as the lower 4 bits of 8-bit composite registers called index register pairs (RnP). For example, R0P consists of R0 (upper 4 bits) and R1 (lower 4 bits), while R4P comprises R8 and R9. These registers are used to store temporary data during computations or to hold partial address data for ROM/RAM access.

4. **Designate Command Line Register (DCL):** A 3-bit register used to determine how the RAM control signals $\overline{\text{CM-RAM0}}$ through $\overline{\text{CM-RAM3}}$ are generated. It is set by the DCL instruction and reset to `000`. Details on the correspondence between DCL values and $\overline{\text{CM-RAM}}$ signals are discussed later.

5. **Stack (Program Address Registers):** The CPU provides a 4-level stack structure composed of 12-bit program address registers. A virtual 2-bit stack pointer (SP) indicates which stack level is active. The program address register pointed to by SP functions as the program counter (PC). Upon executing the subroutine call instruction `JMS`, SP increments and a new program address register becomes the PC, storing the branch destination. The previous PC register holds the return address. When the return-from-subroutine instruction `BBL` is executed, SP decrements, and the return address becomes the new PC, allowing the routine to resume.

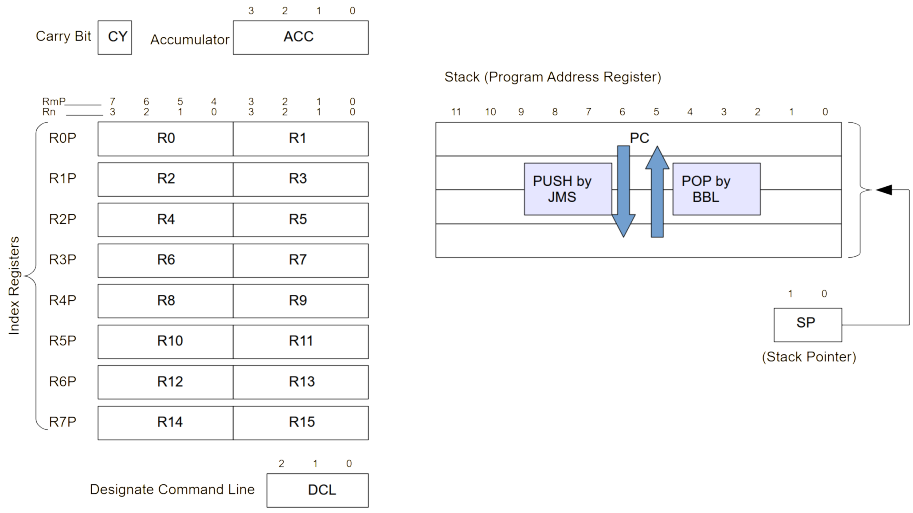


Figure 3.1: 4004 CPU Programmers Model

3.2 Instruction Timing of the CPU

Figure 3.2 illustrates the timing of CPU instructions. The CPU operates on an 8 clock instruction cycle, which consists of the following eight states:

1. **State A1:** Outputs the lower 4 bits of the ROM address, PC[3:0], onto the data bus.

2. **State A2:** Outputs the middle 4 bits of the ROM address, PC[7:4], onto the data bus.

3. **State A3:** Outputs the upper 4 bits of the ROM address, PC[11:8], onto the data bus. Simultaneously, asserts $\overline{\text{CM-ROM}}$ and the $\overline{\text{CM-RAMx}}$ signal selected via the DCL register.

4. **State M1:** Inputs the upper 4 bits of data (OPR) from ROM through the data bus.

5. **State M2:** Inputs the lower 4 bits of data (OPA) from ROM through the data bus. If the previously read OPR is a data access instruction, $\overline{\text{CM-ROM}}$ and the $\overline{\text{CM-RAMx}}$ signal selected via DCL are asserted to inform ROM/RAM of the access type.

6. **State X1:** Instruction execution state for internal processing.

7. **State X2:** Another instruction execution state, which may involve internal processing or outputting the upper 4 bits of a ROM/RAM (I/O ports or memory) address onto the data bus. It may also handle reading from or writing to ROM/RAM via the data bus. If outputting address bits, $\overline{\text{CM-ROM}}$ and DCL-selected $\overline{\text{CM-RAMx}}$ are asserted.

8. **State X3:** Instruction execution state that may perform internal processing or output the lower 4 bits of a ROM/RAM address onto the data bus.

3.2.1 Synchronization of CPU, ROM, and RAM

During State X3, the CPU asserts the $\overline{\text{SYNC}}$ signal to synchronize operations with ROM and RAM. All three units operate under a shared clock and proceed through their internal states in complete synchrony.

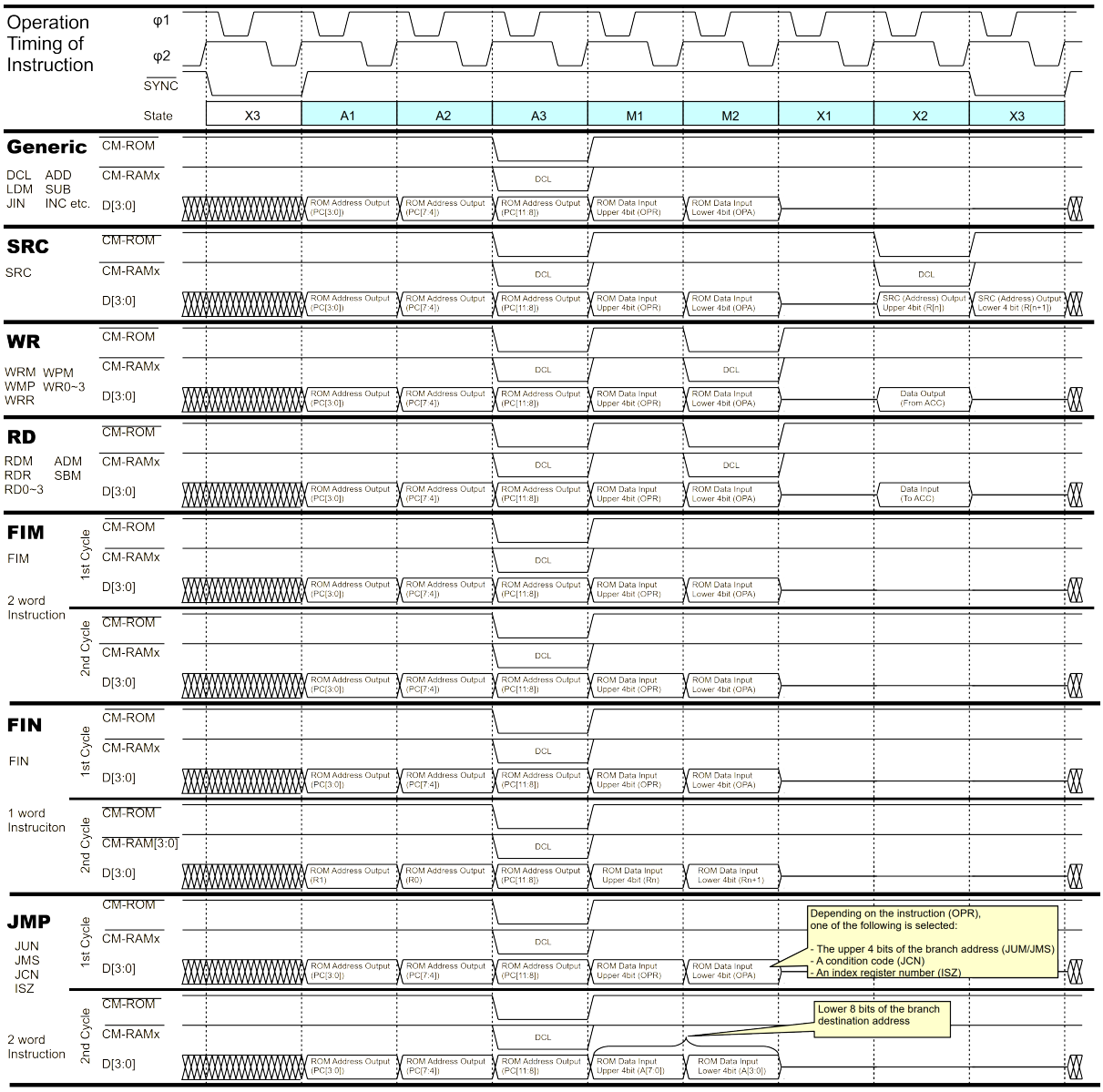


Figure 3.2: 4004 CPU Instruction Timing

3.3 Instruction Set Architecture and Opcode Tables

Specifications and opcodes for each 4004 CPU instruction Set Architecture are presented in Tables 3.1 through 3.4.

Table 3.1 contains single-word instructions (8 bits in length) categorized under the label “Machine Instructions”. While “machine instruction” typically refers to any instruction written in machine language, here it appears to serve as a specific category. It is possible that this grouping includes commands that are neither related to data access nor the accumulator. Among these, the `FIN` instruction stands out: although it is a single-word instruction, it requires two instruction cycles to execute.

Table 3.2 lists the machine instructions that span two words (16 bits total).

Table 3.3 covers single-word data access instructions (8 bits). These involve reading from and writing to ROM I/O ports, RAM memory, and RAM output ports.

Table 3.4 includes single-word accumulator-related instructions (8 bits).

Each instruction's operational characteristics, data flow, and associated control signals are detailed in their respective tables.

3.4 Types of Instruction Execution Timing

Refer again to Figure 3.2, which presents all types of instruction execution timing used in the 4004 CPU. Each category is described as follows:

1. **General Type:** Instructions that are executed entirely within the CPU without accessing external data. These are one-word instructions that complete in a single instruction cycle.

2. **SRC Type:** The `SRC` instruction outputs the lower 8 bits of the ROM/RAM address via states X2 and X3. During state X2, $\overline{\text{CM-ROM}}$ and $\overline{\text{CM-RAMx}}$ (selected by the DCL register) are asserted to transfer address information to ROM/RAM. This is a one-word instruction that completes in a single instruction cycle.

3. **WR Type (Write):** Instructions that write data to external targets such as ROM output ports, RAM main memory characters, RAM status characters, or RAM output ports. The opcode is fetched in state M2 and interpreted by both CPU and ROM/RAM, requiring assertion of $\overline{\text{CM-ROM}}$ and $\overline{\text{CM-RAMx}}$. Data is written during state X2. One-word, one-cycle execution.

4. **RD Type (Read):** Instructions that read data from ROM input ports, RAM main memory characters, or RAM status characters. Like WR instructions, the opcode is fetched in state M2 and interpreted by CPU and ROM/RAM. $\overline{\text{CM-ROM}}$ and $\overline{\text{CM-RAMx}}$ are asserted, and data is read during state X2. One-word, one-cycle execution.

5. **FIM Type:** The `FIM` instruction is a two-word instruction completed in two instruction cycles. The second word, fetched during M1 and M2 of the second cycle, is transferred as immediate data into index register pair RmP.

6. **FIN Type:** The `FIN` instruction is one word long but requires two instruction cycles. After recognition in the first cycle, the second cycle uses A1–A3 and M1–M2 states to read from a ROM address. The retrieved data is stored into index register pair RmP. This instruction reads constant values from ROM.

3.5 Instruction Fetch Operation from ROM

The detailed behavior of instruction fetch operations from ROM is summarized in Figure 3.3. During states A1 and A2, the address signals output from the CPU—specifically PC[3:0] and PC[7:4]—are received by all ROM chips. In state A3, when the $\overline{\text{CM-ROM}}$ signal is asserted, the address signal PC[11:7] is also received by all ROM chips; however, PC[11:8] corresponds to the ROM chip number. This enables selection of the specific ROM chip from which the CPU intends to fetch the instruction.

Type	Mne-monic	Assembly Example	Machine Code OPR (Upper)				OPA (Lower)				Operation Summary	Detailed Description
No Operation	**NOP**	`nop`	0	0	0	0	0	0	0	0	●PC+1→PCTEMP ●PCTEMP→PC	No operation performed.
Load Data to Accumulator	**LDM**	`ldm 0xa` `ldm 10` `ldm LABEL`	1	1	0	1	D				●PC+1→PCTEMP ●D→ACC ●PCTEMP→PC	Loads 4 bit immediate value (OPA field) into ACC. CY remains unchanged.
Load Index Register to Accumulator	**LD**	`ld 3` `ld 15`	1	0	1	0	n				●PC+1→PCTEMP ●Rn→ACC ●PCTEMP→PC	Loads 4 bit content of index register Rn into ACC. CY unchanged.
Exchange Index Register and Accumulator	**XCH**	`xch 3` `xch 15`	1	0	1	1	n				●PC+1→PCTEMP ●ACC→ACBR ●Rn→ACC ●ACBR→Rn ●PCTEMP→PC	Exchanges contents of ACC and Rn. CY unchanged.
Add Index Register to Accumulator with Carry	**ADD**	`add 3` `add 15`	1	0	0	0	n				●PC+1→PCTEMP ●ACC+Rn+CY→ACC,CY ●PCTEMP→PC	Performs 4-bit addition of ACC, Rn, and CY. Result to ACC; carry-out to CY. `a3 a2 a1 a0` `cy` `+) r3 r2 r1 r0` `c4 s3 s2 s1 s0` `c4→CY, {s3,s2,s1,s0}→ACC`
Subtract Index Register from Accumulator with Borrow	**SUB**	`sub 3` `sub 15`	1	0	0	1	n				●PC+1→PCTEMP ●ACC+$\overline{Rn}$+$\overline{CY}$→ACC,CY ●PCTEMP→PC	The contents of ACC, the bitwise inversion of the 4-bit value in index register Rn (as specified by number n), and the inverted CY bit are added together. The result is stored in ACC, and the carry-out is stored in CY. Note: If the CY value prior to calculation is 0, subtraction is performed without borrow. If the CY value prior to calculation is 1, subtraction is performed with borrow. After calculation, CY = 0 indicates a borrow occurred, and CY = 1 indicates no borrow occurred. Keep in mind that the interpretation of the CY bit is inverted after execution with respect to the borrow status. `a3 a2 a1 a0` $\overline{cy}$ `+)` $\overline{r3}$ $\overline{r2}$ $\overline{r1}$ $\overline{r0}$ `c4 s3 s2 s1 s0` `c4→CY, {s3,s2,s1,s0}→ACC`
Increment Index Register	**INC**	`inc 8` `inc 12`	0	1	1	0	n				●PC+1→PCTEMP ●Rn+1→Rn ●PCTEMP→PC	The 4 bit content of index register Rn, designated by number n, is incremented by 1. If the original value is 15, it wraps around to 0. The CY (carry) bit remains unchanged.
Branch Back and Load Data to Accumulator	**BBL**	`bbl 0xa` `bbl 12`	1	1	0	0	D				●PC+1→PCTEMP ●SP−1→SP ●(Stack)→PC ●D→ACC	This instruction performs a return from a subroutine. The stack pointer (SP) is decremented by one, and the return address is retrieved from the stack and loaded into the program counter (PC). The PC now points to the instruction following the subroutine call (JMS). Additionally, the 4-bit value D from the instruction's OPA field is transferred into the accumulator (ACC).
Jump indirect	**JIN**	`jin 3p` `jin 3<` `These are identical.`	0	0	1	1	m			1	●PC+1→PCTEMP ●PCTEMP[11:8]→PC[11:8] ●RmP[7:4]→PC[7:4] ●RmP[3:0]→PC[3:0]	After the PC is incremented (+1) to point to the next instruction, the lower 8 bits of the PC are overwritten with the 8-bit contents of index register pair RmP, designated by number m. Note: If PC[7:0] was 0xFF before executing the JIN instruction, incrementing the PC causes a carry into PC[11:8] (page number), which is then incremented. The lower 8 bits are overwritten afterward. If the JIN instruction resides at the last address of a page (PC[7:0] == 0xFF), the jump target ends up inside the next page.
Send Register Control	**SRC**	`src 5p` `src 5<` `src 10` `These are identical.`	0	0	1	0	m			1	●PC+1→PCTEMP ●RmP[7:4]→DB @X2 ●RmP[3:0]→DB @X3 ●PCTEMP→PC	The 8-bit contents of index register pair RmP, specified by number m, are output to the data bus: the upper 4 bits during state X2, and the lower 4 bits during state X3. These values serve as address information for chip selection when accessing ROM (I/O ports), or RAM (output ports, character data, status characters). ROM and RAM receive and retain these values for upcoming access instructions.
Fetch Index Register from ROM	**FIN**	`fin 7p` `fin 7<` `fin 14` `These are identical.`	0	0	1	1	m			0	命令サイクル1で ●PC+1→PCTEMP 命令サイクル2で ●R0P[3:0]→DB @A1 ●R0P[7:4]→DB @A2 ●PCTEMP[11:8]→DB @A3 ●DB→RmP[7:4] @M1(OPR) ●DB→RmP[3:0] @M2(OPA) ●PCTEMP→PC	Although this is a one-word instruction, it requires two instruction cycles to execute. In instruction cycle 1, the PC is incremented (+1) to point to the next instruction. In instruction cycle 2, the ROM address is built as follows: - During state A1: the lower 4 bits of index register pair R0P (i.e., R1) are output to the data bus - During state A2: the upper 4 bits of R0P (i.e., R0) are output to the data bus -During state A3: the upper bits of PCTEMP (PC[11:8], the page number) are output to the data bus ROM data is read via states M1 and M2: - M1 retrieves the upper 4 bits (OPR) and loads them into the upper part of RmP - M2 retrieves the lower 4 bits (OPA) and loads them into the lower part of RmP - Finally, PCTEMP is transferred to PC Note: If PC[7:0] was 0xFF before executing the FIN instruction, the PC increment causes a carry into PC[11:8]. The read address for ROM is then constructed using this new page value and the R0P pair. If the FIN instruction is placed at the final address of a page (PC[7:0] == 0xFF), the ROM access address will be inside the next page.

Table 3.1: CPU Instruciton Table (1) : Machine Instruction (1 word)

Type	Mne-monic	Assembly Example	Machine Code OPR (Upper)	Machine Code OPA (Lower)	Operation Summary	Detailed Description
Jump Unconditional	**JUN**	`jun LABEL` `jun 0x123`	0 1 0 0	A3	●PC+1→PCTEMP ●PCTEMP→PC	This is an unconditional branch instruction. The A3, A2, and A1 fields from the instruction word are transferred to the program counter (PC).
			A2	A1	●PC+1→PCTEMP ●A3→PC[11:8] ●A2→PC[7:4] ●A1→PC[3:0]	
Jump to Subroutine	**JMS**	`jms LABEL` `jms 0x123`	0 1 0 1	A3	●PC+1→PCTEMP ●PCTEMP→PC	This is a subroutine call instruction. The PC is first incremented by 2 to point to the instruction following the call, and this return address is stored in the stack. The stack pointer (SP) is incremented by 1, and the newly selected stack level becomes the active PC. The A3, A2, and A1 fields from the instruction word are then transferred to that PC. Since the stack has four levels, the JMS instruction supports nesting up to three levels.
			A2	A1	●PC+1→PCTEMP ●PCTEMP→(Stack) ●SP+1→SP ●A3→PC[11:8] in new Stack ●A2→PC[7:4] in new Stack ●A1→PC[3:0] in new Stack	

Jump Conditional	**JCN**	`jcn 0x4 LABEL`
		`jcn 9 0x123`
		`jcn TZ TARGET0`
		`jcn TN TARGET1`
		`jcn C0 TARGET2`
		`jcn C1 TARGET3`
		`jcn AZ TARGET4`
		`jcn AN TARGET5`
		
		`Reserved Words for Condition Bits`
		
		`TZ...CCCC=0001`
		`TN...CCCC=1001`
		`C0...CCCC=1010`
		`C1...CCCC=0010`
		`AZ...CCCC=0100`
		`AN...CCCC=1100`
			0 0 0 1	C1 C2 C3 C4	●PC+1→PCTEMP
					●PCTEMP→PC
						This is a conditional branch instruction. Regardless of whether branching occurs, the PC is first incremented by 2 to point to the next instruction. Then, if the condition flags C1, C2, C3, and C4 specified in the instruction word are satisfied, the branch is taken. Otherwise, the instruction following JCN is executed. When branching, the A2 and A1 fields are transferred to PC[7:0].
						
						Condition bit meanings (multiple conditions can be evaluated simultaneously):
						- C1 = 0: Do not invert the branch condition.
						- C1 = 1: Invert the branch condition.
						- C2 = 1: Branch if ACC equals 0.
						- C3 = 1: Branch if CY equals 1.
						- C4 = 1: Branch if TEST input signal equals 0.
						
						Branch evaluation logic:
						- JUMP = C1 XOR JUMP0
						- JUMP0 = C2·(ACC==0) + C3·CY + C4·(TEST==0)
						
						Note: If PC[7:0] is 0xFE or 0xFF before executing the JCN instruction, incrementing PC by 2 causes a carry into PC[11:8], advancing the page number. However, during a branch, only PC[7:0] is updated. Therefore, if JCN is located at the end of a page (PC[7:0] == 0xFE or 0xFF), the branch target will reside within the next page.

			A2	A1	●PC+1→PCTEMP ●If (C1,C2,C3,C4) is true, PCTEMP[11:8]→PC[11:8] A2→PC[7:4] A1→PC[3:0] ●Else PCTEMP→PC	
Increment Index Register, Skip if Zero	**ISZ**	`isz 9 LABEL` `isz 9 0x123`	0 1 1 1	n	●PC+1→PCTEMP ●PCTEMP→PC	This is a conditional branch instruction. Regardless of whether branching occurs, the PC is first incremented by 2 to point to the next instruction. Then, index register Rn (designated by number n) is incremented, and if the result is non-zero, the branch is taken. If the result is zero, the instruction following ISZ is executed. When branching, the A2 and A1 fields are transferred to PC[7:0]. Note: If PC[7:0] is 0xFE or 0xFF before executing the ISZ instruction, incrementing PC by 2 causes a carry into PC[11:8], advancing the page number. However, during a branch, only PC[7:0] is updated. Therefore, if ISZ is located at the end of a page (PC[7:0] == 0xFE or 0xFF), the branch target will reside within the next page.
			A2	A1	●PC+1→PCTEMP ●Rn+1→Rn ●If (Rn != 0), PCTEMP[11:8]→PC[11:8] A2→PC[7:4] A1→PC[3:0] ●Else PCTEMP→PC	

Fetch Immediate from ROM	**FIM**	`fim 1p 0xfe`
		`fim 1< 0xfe`
		`fim 2 0xfe`
		
		`These are identical.`
			0 0 1 0	m 0	●PC+1→PCTEMP
					●PCTEMP→PC
						The second word of the instruction is used as an immediate constant and stored into index register pair RmP.

			D2	D1	●PC+1→PCTEMP ●D2→RmP[7:4] ●D1→RmP[3:0] ●PCTEMP→PC	

Table 3.2: CPU Instruciton Table (2) : Machine Instruction (2 word)

Type	Mnemonic	Assembly Example	Machine Code OPR (Upper)				OPA (Lower)				Operation Summary	Detailed Description
Read RAM Character	**RDM**	rdm	1	1	1	0	1	0	0	1	●PC+1→PCTEMP ●(RAM_CH)→ACC ●PCTEMP→PC	Reads the RAM main memory character selected by the DCL and SRC instructions and stores it in the ACC. The CY bit remains unchanged.
Read RAM Status Character 0	**RD0**	rd0	1	1	1	0	1	1	0	0	●PC+1→PCTEMP ●(RAM_ST0)→ACC ●PCTEMP→PC	Reads RAM status character 0 selected by the DCL and SRC instructions and stores it in the ACC. The CY bit remains unchanged.
Read RAM Status Character 1	**RD1**	rd1	1	1	1	0	1	1	0	1	●PC+1→PCTEMP ●(RAM_ST1)→ACC ●PCTEMP→PC	Reads RAM status character 1 selected by the DCL and SRC instructions and stores it in the ACC. The CY bit remains unchanged.
Read RAM Status Character 2	**RD2**	rd2	1	1	1	0	1	1	1	0	●PC+1→PCTEMP ●(RAM_ST2)→ACC ●PCTEMP→PC	Reads RAM status character 2 selected by the DCL and SRC instructions and stores it in the ACC. The CY bit remains unchanged.
Read RAM Status Character 3	**RD3**	rd3	1	1	1	0	1	1	1	1	●PC+1→PCTEMP ●(RAM_ST3)→ACC ●PCTEMP→PC	Reads RAM status character 3 selected by the DCL and SRC instructions and stores it in the ACC. The CY bit remains unchanged.
Read ROM Port	**RDR**	rdr	1	1	1	0	1	0	1	0	●PC+1→PCTEMP ●(ROM_PORT_IN)→ACC ●PCTEMP→PC	Reads the ROM input port selected by the SRC instruction and stores it in the ACC. The CY bit remains unchanged. Note: The ROM (4001) port is bidirectional. When reading from a port set to output, the signal level depends on the metal options specified when ordering the 4001 chip.
Write Accumulator into RAM Character	**WRM**	wrm	1	1	1	0	0	0	0	0	●PC+1→PCTEMP ●ACC→(RAM_CH) ●PCTEMP→PC	Writes the contents of ACC to the RAM main memory character selected by the DCL and SRC instructions. The CY bit remains unchanged.
Write Accumulator into RAM Status Character 0	**WR0**	wr0	1	1	1	0	0	1	0	0	●PC+1→PCTEMP ●ACC→(RAM_ST0) ●PCTEMP→PC	Writes the contents of ACC to RAM status character 0 selected by the DCL and SRC instructions. The CY bit remains unchanged.
Write Accumulator into RAM Status Character 1	**WR1**	wr1	1	1	1	0	0	1	0	1	●PC+1→PCTEMP ●ACC→(RAM_ST1) ●PCTEMP→PC	Writes the contents of ACC to RAM status character 1 selected by the DCL and SRC instructions. The CY bit remains unchanged.
Write Accumulator into RAM Status Character 2	**WR2**	wr2	1	1	1	0	0	1	1	0	●PC+1→PCTEMP ●ACC→(RAM_ST2) ●PCTEMP→PC	Writes the contents of ACC to RAM status character 2 selected by the DCL and SRC instructions. The CY bit remains unchanged.
Write Accumulator into RAM Status Character 3	**WR3**	wr3	1	1	1	0	0	1	1	1	●PC+1→PCTEMP ●ACC→(RAM_ST3) ●PCTEMP→PC	Writes the contents of ACC to RAM status character 3 selected by the DCL and SRC instructions. The CY bit remains unchanged.
Write ROM Port	**WRR**	wrr	1	1	1	0	0	0	1	0	●PC+1→PCTEMP ●ACC→(ROM_PORT_OUT) ●PCTEMP→PC	Writes the contents of ACC to the ROM output port selected by the SRC instruction. The CY bit remains unchanged. Note: ROM (4001) ports are bidirectional. Writing to ports configured as input will have no effect.
Write Memory Port	**WMP**	wmp	1	1	1	0	0	0	0	1	●PC+1→PCTEMP ●ACC→(RAM_PORT_OUT) ●PCTEMP→PC	Writes the contents of ACC to the RAM output port selected by the DCL and SRC instructions. The CY bit remains unchanged.
Add from Memory with Carry	**ADM**	adm	1	1	1	0	1	0	1	1	●PC+1→PCTEMP ●ACC+(RAM_CH)+CY →ACC,CY ●PCTEMP→PC	Reads the RAM main memory character selected by the DCL and SRC instructions, adds it to the ACC with carry.
Subtract from Memory with Borrow	**SBM**	sbm	1	1	1	0	1	0	0	0	●PC+1→PCTEMP ●ACC+$\overline{(RAM\_CH)}+\overline{CY}$ →ACC,CY ●PCTEMP→PC	Reads the RAM main memory character selected by the DCL and SRC instructions, subtracts it from the ACC with borrow. Note: If the CY value before the calculation is 0, subtraction proceeds without borrow. If CY is 1 before the calculation, subtraction is performed with borrow. After the operation: CY = 0 indicates a borrow occurred; CY = 1 indicates no borrow occurred. Be aware that the meaning of the CY bit is inverted after the calculation with respect to borrow.
Write Program RAM	**WPM**	wpm	1	1	1	0	0	0	1	1	Refer to the main text.	Special instructions for reading and writing program memory in systems where program memory is implemented in RAM.

Table 3.3: CPU Instruciton Table (3) : Data Access

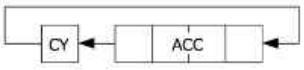
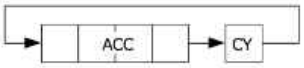
Type	Mnemonic	Assembly Example	Machine Code OPR (Upper)				OPA (Lower)				Operation Summary	Detailed Description
Clear Both	**CLB**	`clb`	1	1	1	1	0	0	0	0	●PC+1→PCTEMP ●0→ACC ●0→CY ●PCTEMP→PC	Clears both the ACC (accumulator) and CY (carry) to zero.
Clear Carry	**CLC**	`clc`	1	1	1	1	0	0	0	1	●PC+1→PCTEMP ●0→CY ●PCTEMP→PC	Clears CY to zero.
Complement Carry	**CMC**	`cmc`	1	1	1	1	0	0	1	1	●PC+1→PCTEMP ●$\overline{CY}$→CY ●PCTEMP→PC	Inverts the value of CY.
Set Carry	**STC**	`stc`	1	1	1	1	1	0	1	0	●PC+1→PCTEMP ●1→CY ●PCTEMP→PC	Sets CY to 1.
Complement Accumulator	**CMA**	`cma`	1	1	1	1	0	1	0	0	●PC+1→PCTEMP ●$\overline{ACC}$→ACC ●PCTEMP→PC	Inverts ACC. CY remains unchanged.
Increment Accumulator	**IAC**	`iac`	1	1	1	1	0	0	1	0	●PC+1→PCTEMP ●ACC+1→ACC,CY ●PCTEMP→PC	Increments ACC. If there is no overflow, CY becomes 0; if an overflow occurs, CY becomes 1.
Decrement Accumulator	**DAC**	`dac`	1	1	1	1	1	0	0	0	●PC+1→PCTEMP ●ACC−1→ACC,CY ●PCTEMP→PC	Decrements ACC. If there is no borrow, CY becomes 1; if a borrow occurs, CY becomes 0. `a3 a2 a1 a0` `+) 1 1 1 1` `c4 s3 s2 s1 s0` `c4→CY, {s3,s2,s1,s0}→ACC`
Rotate Left	**RAL**	`ral`	1	1	1	1	0	1	0	1	●PC+1→PCTEMP ●{ACC[3:0],CY} →{CY, ACC[3:0]} ●PCTEMP→PC	Performs a left rotate of ACC including CY. 
Rotate Right	**RAR**	`rar`	1	1	1	1	0	1	1	0	●PC+1→PCTEMP ●{CY,ACC[3:0]} →{ACC[3:0],CY} ●PCTEMP→PC	Performs a right rotate of ACC including CY. 
Transmit Carry and Clear	**TCC**	`tcc`	1	1	1	1	0	1	1	1	●PC+1→PCTEMP ●0→ACC ●CY→ACC[0] ●0→CY ●PCTEMP→PC	Clears ACC, then transfers the CY value to ACC's least significant bit (LSB), and finally clears CY to 0.
Decimal Adjust Accumulator	**DAA**	`daa`	1	1	1	1	1	0	1	1	●PC+1→PCTEMP ●If (CY|(ACC>9)), ACC+6→ACC ●If Carry, 1→CY ●PCTEMP→PC	Decimal adjust instruction: If CY is 1 or ACC is greater than 9, add 6 to ACC. If this addition generates a carry, set CY to 1; if there's no carry, CY remains unchanged.
Transfer Carry Subtract	**TCS**	`tcs`	1	1	1	1	1	0	0	1	●PC+1→PC ●If (CY==0),9→ACC ●If (CY==1),10→ACC ●0→CY	If CY = 0, store 9 into ACC; if CY = 1, store 10 into ACC. Then clear CY to 0.
Keyboard Process	**KBP**	`kbp`	1	1	1	1	1	1	0	0	●PC+1→PCTEMP ●KBP(ACC)→ACC ●PCTEMP→PC	Keyboard scan code conversion instruction: If the value in ACC has exactly one bit set to 1, it is converted to a numerical value. If multiple bits are set to 1, the result is set to 15 (indicating an error). CY remains unchanged. `ACC (Before) → ACC (After)` `0 0 0 0 → 0 0 0 0` `0 0 0 1 → 0 0 0 1` `0 0 1 0 → 0 0 1 0` `0 1 0 0 → 0 0 1 1` `1 0 0 0 → 0 1 0 0` `0 0 1 1 → 1 1 1 1` `0 1 0 1 → 1 1 1 1` `0 1 1 0 → 1 1 1 1` `0 1 1 1 → 1 1 1 1` `1 0 0 1 → 1 1 1 1` `1 0 1 0 → 1 1 1 1` `1 0 1 1 → 1 1 1 1` `1 1 0 0 → 1 1 1 1` `1 1 0 1 → 1 1 1 1` `1 1 1 0 → 1 1 1 1` `1 1 1 1 → 1 1 1 1`
Designate Command Line	**DCL**	`dcl`	1	1	1	1	1	1	0	1	●PC+1→PCTEMP ●ACC[2:0]→DCL ●PCTEMP→PC	Transfers the lower 3 bits of ACC to the DCL register inside the CPU, thereby specifying how $\overline{\text{CM-RAMx}}$ (RAM bank selection) signals are output from that point forward.

Table 3.4: CPU Instruciton Table (4) : Accumulator

Following this process, in states M1 and M2, the selected ROM chip outputs the high-order instruction code (OPR) and low-order instruction code (OPA), which the CPU then fetches.

In state A3, the $\overline{\text{CM-ROM}}$ signal is asserted. Although CPU and ROM operate synchronously—so $\overline{\text{CM-ROM}}$ is not strictly necessary—the use of $\overline{\text{CM-ROM}}$ becomes advantageous when expanding the number of connected ROM chips beyond 16. It simplifies the design of external circuits for ROM selection.

Additionally, $\overline{\text{CM-RAMx}}$ is also asserted during state A3. While this has no inherent significance in standard operation, it becomes meaningful if external circuitry has been added to allow ROM access via $\overline{\text{CM-RAMx}}$—for example, when expanding ROM capacity by repurposing RAM access signals.

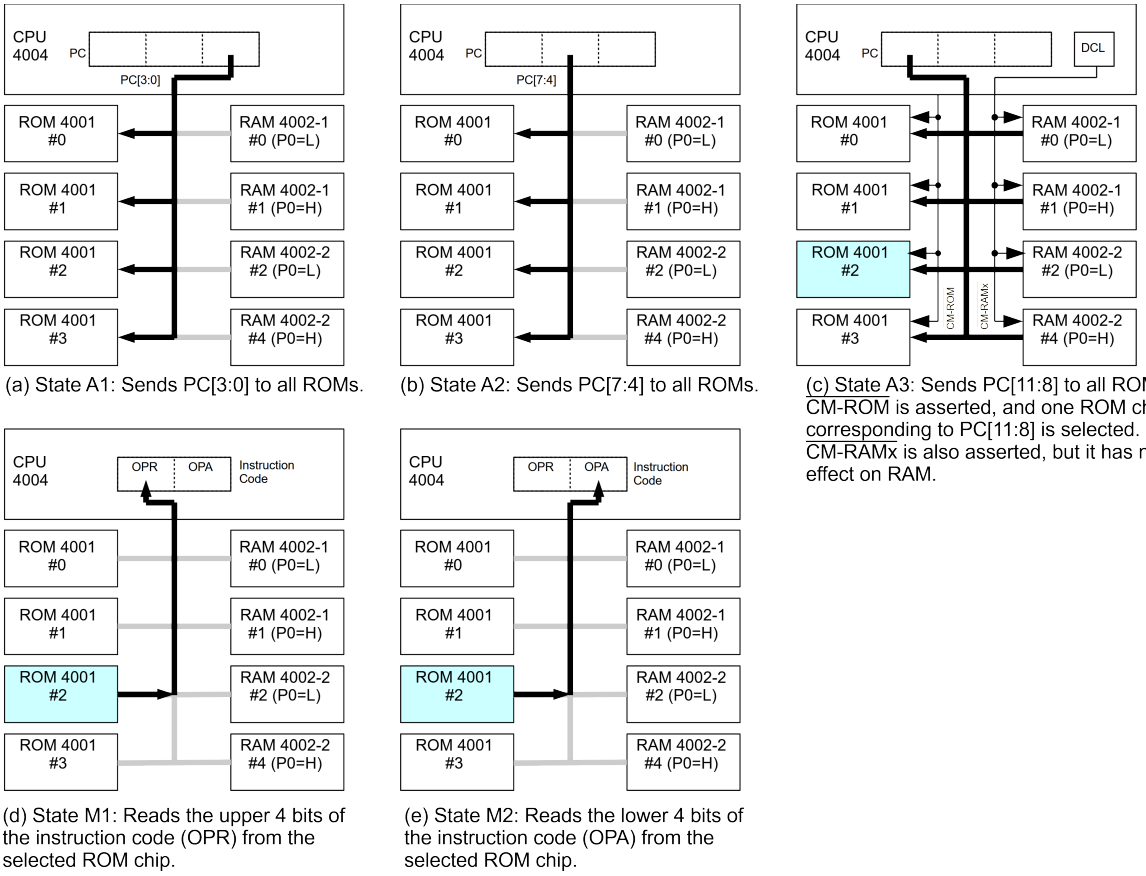


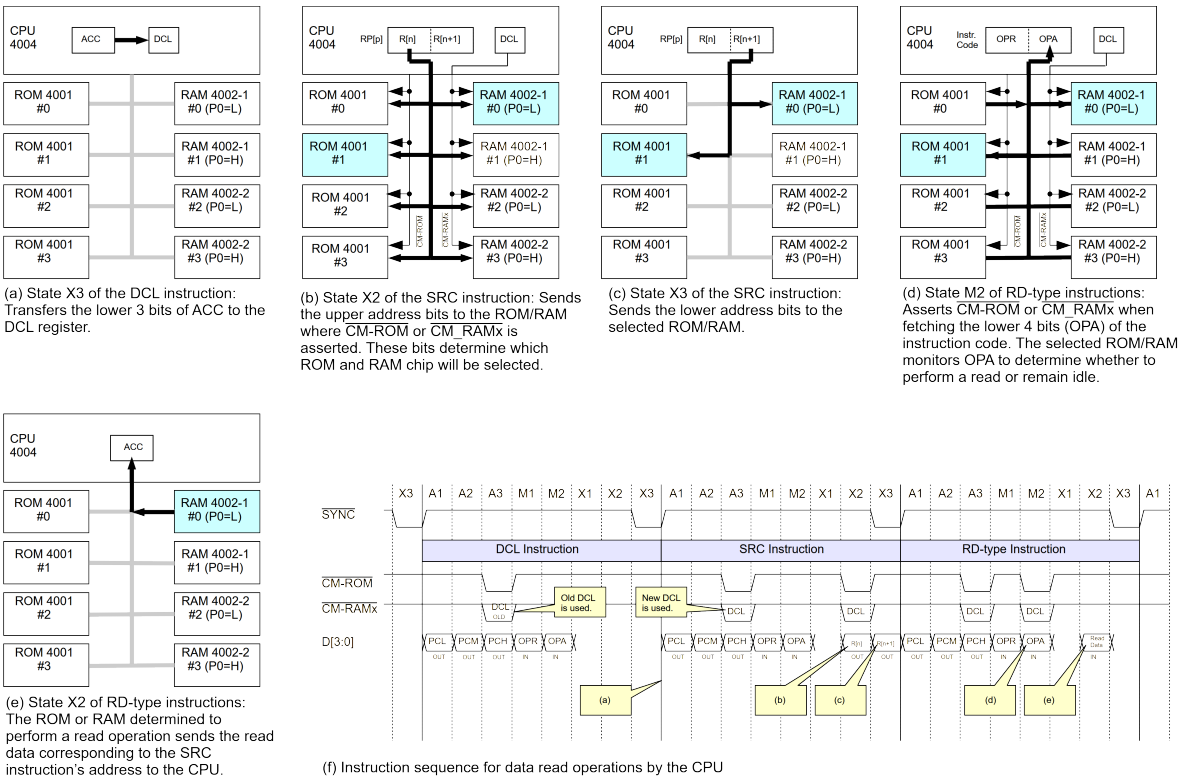
Figure 3.3: 4004 CPU Instruction Fetch

3.6 CPU Data Read Operation

The CPU's data read operation for external devices such as RAM main memory characters, RAM status characters, and ROM input ports is illustrated in Figure 3.4. When accessing external devices, the operation is split into two instructions: one to transmit address information and another to perform the actual data access.

3.6.1 DCL Instruction: RAM Bank Selection

When accessing RAM, the target RAM bank must first be designated by executing the `DCL` instruction as shown in Figure 3.4(a). This instruction stores the lower 3 bits of



(a) State X3 of the DCL instruction: Transfers the lower 3 bits of ACC to the DCL register.

(b) State X2 of the SRC instruction: Sends the upper address bits to the ROM/RAM where CM-ROM or CM_RAMx is asserted. These bits determine which ROM and RAM chip will be selected.

(c) State X3 of the SRC instruction: Sends the lower address bits to the selected ROM/RAM.

(d) State M2 of RD-type instructions: Asserts CM-ROM or CM_RAMx when fetching the lower 4 bits (OPA) of the instruction code. The selected ROM/RAM monitors OPA to determine whether to perform a read or remain idle.

(e) State X2 of RD-type instructions: The ROM or RAM determined to perform a read operation sends the read data corresponding to the SRC instruction's address to the CPU.

(f) Instruction sequence for data read operations by the CPU

Figure 3.4: 4004 CPU Data Read Operation

the accumulator into the DCL register. According to the value of DCL, a subsequent instruction will assert the corresponding CM-RAMx signal.

Table[reftb:DCLANDCMRAM shows the correspondence between DCL values and CM-RAMx signals. - For systems with up to 4 RAM banks, only four DCL patterns (3'b000, 3'b001, 3'b010, 3'b100) are used to assert CM-RAMx in a one-hot manner, eliminating the need for external circuits. - For systems with up to 8 RAM banks, all 8 combinations of the 3-bit DCL are used. Since CM-RAMx may not be one-hot in this case, external decoding circuits (see Figure 3.5) are used to expand the number of CM-RAMx lines.

If RAM is not being accessed, or the same bank is reused, or the default DCL value (3'b000) is retained, executing the DCL instruction is unnecessary.

ACC	DCL	CM-RAMx				Bank No.	Use with 4 banks	Use with 8 banks
		3	2	1	0			
x000	000	H	H	H	L	0	○	○
x001	001	H	H	L	H	1	○	○
x010	010	H	L	H	H	2	○	○
x011	011	H	L	L	H	3		○
x100	100	L	H	H	H	4	○	○
x101	101	L	H	L	H	5		○
x110	110	L	L	H	H	6		○
x111	111	L	L	L	H	7		○

Table 3.5: Correspondence between DCL values and CM-RAMx signals

3.6.2 SRC Instruction: Address Transmission

Next, the `SRC` instruction is executed as shown in Figures 3.4(b) and (c). The instruction transfers the 8-bit value stored in index register pair RnP to ROM/RAM as address

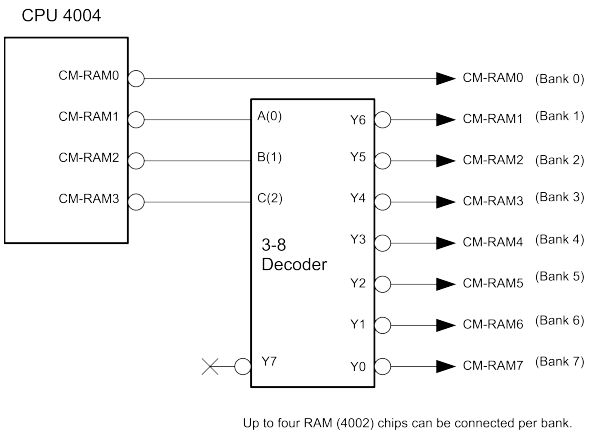


Figure 3.5: RAM Bank Expansion and DCL Decoder
To increase the number of CM-RAM lines. Up to 4 RAM (4002) chips can be connected per bank.

information. The address structure is summarized in Table 3.6.

- In state X2, CM-ROM and CM-RAMx are asserted. The upper 4 bits of the address are transmitted to all ROM chips and to RAM chips within the bank selected by CM-RAMx. As shown in Table 4, this portion contains chip select information. Once the ROM/RAM receives the upper 4 bits, the active ROM and RAM chips are determined.
- In state X3, the lower 4 bits of the address are sent. The ROM/RAM chips activated in state X2 receive this part and await subsequent data access instructions.

Note: The address stored inside the ROM/RAM chip by the SRC instruction is retained even after the data access instruction is executed. Thus, when accessing the same location again, the SRC instruction does not need to be re-executed.

The SRC address for RAM main memory characters is calculated as: `(Chip # × 64) + (Register # × 16) + (Character #)`

Bit Position			7	6	5	4	3	2	1	0	Related Instruction
SRC Address			R[n] (Output at X2)				R[n+1] (Output at X3)				
Access Target	ROM (4001)	I/O Port	Chip Selection				don't care				RDR WRR
	RAM (4002)	Out Port	Chip Selection in Bank		don't care						WMP
		Main Memory Character	Chip Selection in Bank		Register Selection in Chip		Character Selection in Register				RDM, WRM ADM, SBM
		Status Character	Chip Selection in Bank		Register Selection in Chip		don't care				RD0, RD1, RD2, RD3 WR0, WR1, WR2, WR3

Table 3.6: SRC Address Structure

3.6.3 Data Access Instructions

To perform actual data read/write operations, a data access instruction is executed. These instructions all have an opcode upper 4 bits (OPR) equal to 4'b1110, allowing the CPU to recognize them as data access instructions when fetched in state M1.

Then, during state M2—when the lower 4 bits of the opcode (OPA) are fetched—$\overline{\text{CM-ROM}}$ and the $\overline{\text{CM-RAMx}}$ signal corresponding to the DCL register are asserted, as shown in Figures 3.4(d).

During this M2 state, ROM/RAM interprets the OPA on the data bus to determine the type of access (read or write) and the target. If the instruction performs a read operation, the ROM/RAM outputs the data to the data bus in state X2 (same instruction cycle), as illustrated in Figures 3.4(e), and the CPU stores the data into the accumulator.

Refer to Table 3.3 for types of data read instructions.

3.7 CPU Data Write Operation

Figure 3.6 illustrates the sequence of operations when the CPU writes to external components such as RAM main memory characters, RAM status characters, or output ports of ROM/RAM. The write operation follows a sequence fundamentally similar to that of a read.

During state X2 of the data access instruction, the CPU outputs the write data—stored in the accumulator—onto the data bus. The target ROM or RAM component, having been selected by prior addressing signals, receives this data and writes it to the designated address.

Refer to Table 3.3 for the types of data write instructions available.

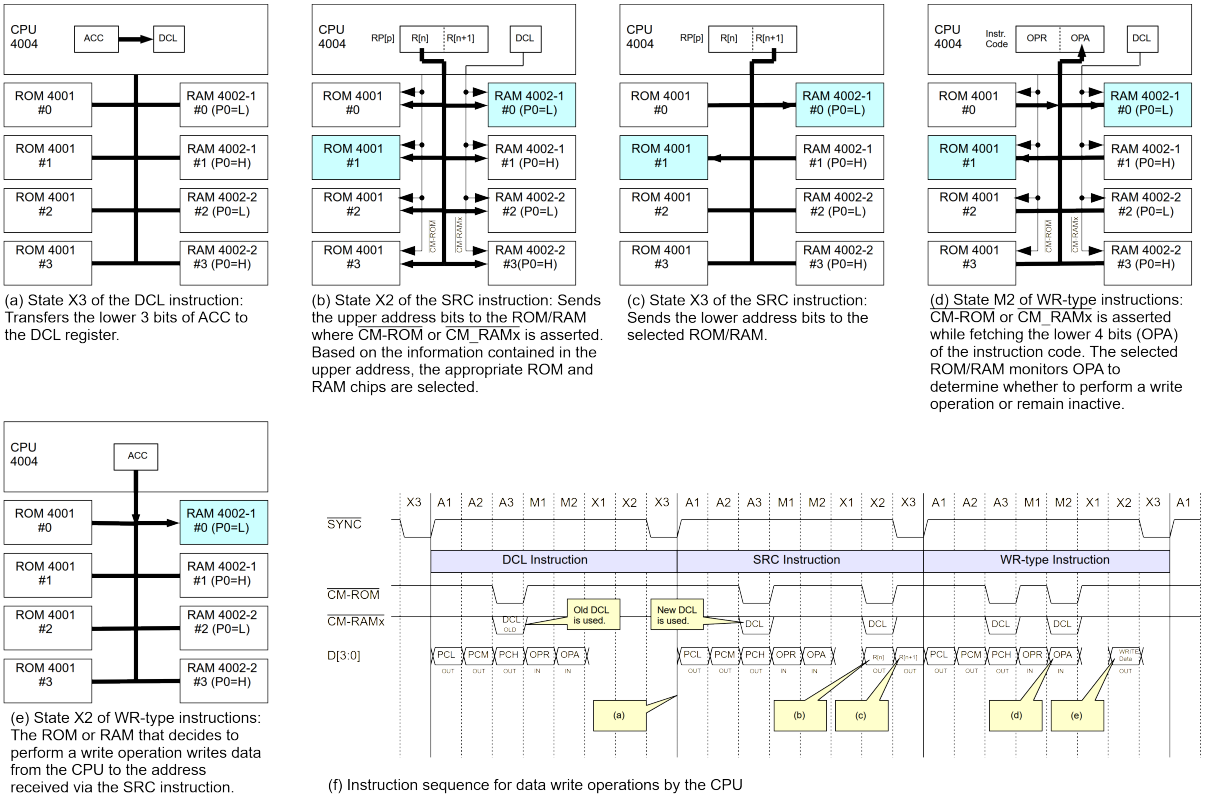


Figure 3.6: 4004 CPU Data Write Operation

3.8 Subroutine Call Operation in the 4004 CPU

Modern CPUs typically handle subroutine calls by saving the return address to a stack located in memory space. In contrast, the 4004 CPU employs a dedicated stack mechanism, consisting of four levels of 12-bit program address registers, to implement subroutine calls. This structure is illustrated in Figure 3.1.

Figure 3.7 outlines the operational behavior:

- **(a)** Initially, the program address register pointed to by the stack pointer (SP) serves as the program counter (PC).

- **(b)** When a subroutine call instruction (`JMS`) is executed, the SP is incremented, and the target address of the subroutine is stored in the newly selected program address register, which then functions as the new PC. Meanwhile, the previously selected register (prior to SP increment) retains the return address—the address of the instruction immediately following the `JMS` call.

- **(c, d)** Repeated executions of `JMS` result in the same behavior, each time nesting one level deeper in the stack.

- **(e)** Since the stack has only four levels, executing `JMS` four times consecutively will overwrite the earliest return address. Therefore, the 4004 CPU imposes a limit of three subroutine nesting levels.

- **(f)** When a subroutine return instruction (`BBL`) is executed, the SP is decremented, and the program address register that now contains the previous return address becomes the new PC, allowing instruction execution to resume at the correct return point.

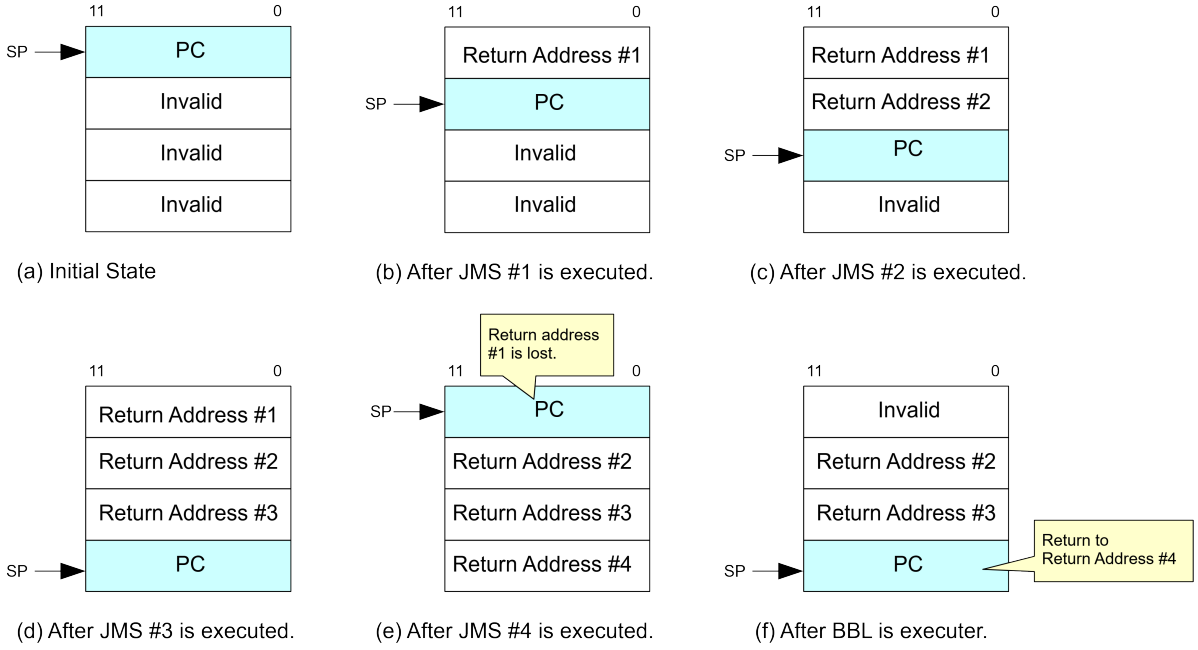


Figure 3.7: 4004 CPU Subroutine Call

3.9 TEST Pin as an Alternative to Interrupts

Unlike modern CPUs that naturally support interrupt handling, the 4004 CPU does not have a built-in interrupt feature. However, it does provide a mechanism that can function similarly to an interrupt: the `TEST` input pin.

The TEST signal can be included as part of the conditional branching logic within the `JCN` (Jump Conditional) instruction. By monitoring the input level of the TEST pin and executing `JCN` instructions at regular intervals, the system can respond to external conditions with a limited latency. Thus, the TEST pin can effectively serve as an interrupt trigger.

3.10 Instructions Located at ROM Page Boundaries

For one-word instructions that modify only the lower 8 bits of the PC (program counter), if such instructions reside at the final address of a 256-byte ROM page (`0x..FF`), they will branch to an address within the next page. This is because the system generates `PC+1` and then replaces only the lower 8 bits for the branch.

Similarly, for two-word instructions that modify only the lower 8 bits of the PC, if they are located at the final address or its immediate predecessor (`0x..FF` or `0x..FE`), they will also branch into the next ROM page. In this case, the system generates `PC+2` and updates only the lower 8 bits of the PC for branching.

3.11 Timing of PC Update

From these observations, the timing for PC updates has been interpreted and incorporated into the next chapter's RTL design as follows:

Whether the instruction is one-word or two-word, the address for the next instruction cycle (`PC+1`) is stored in a temporary signal named `PCTEMP` at the beginning of state A1. Any references to PC during instruction execution refer to `PCTEMP`.

For non-branching instructions or conditional branches that do not result in a branch, `PCTEMP` is transferred to `PC` during the final state X3 of the instruction cycle.

If a branch occurs, the final state X3 updates `PC` with the branch destination address. In cases where only the lower 8 bits of the PC are modified, the upper 4 bits `PC[11:8]` are assigned from `PCTEMP[11:8]`.

3.12 FIN Instruction and ROM Page Boundaries

The `FIN` instruction is unique in that although it is a one-word instruction, it spans two instruction cycles and requires careful handling of PC updates.

If a FIN instruction resides at the last address of a ROM page (`0x..FF`), it fetches data from the next page. At the beginning of the first cycle, `PC+1` is stored in `PCTEMP`, but no transfer to `PC` occurs at the end of X3.

During the second instruction cycle, states A1 to A3 output the ROM read address: - A1 outputs `ROP[3:0]` - A2 outputs `ROP[7:4]` - A3 outputs `PCTEMP[11:7]`

Finally, PC is updated with the value of `PCTEMP` during the last X3 state of the second cycle.

3.13 Appreciating the Wisdom of the Pioneers

3.13.1 Ingenuity in Packing Maximum Functionality into Minimal Hardware

As we have seen, the MCS-4 system is filled with ingenious techniques for extracting the maximum functionality from minimal hardware. Each chip is housed in a compact 16-pin package, and address and data are transmitted time-divisionally over a 4-bit data bus. Through tightly coordinated operation of the CPU, ROM, and RAM, interface signals are minimized, enabling the construction of a microcomputer system with simple interconnections. One striking revelation is that instruction decoding is not limited to the CPU—the ROM and RAM also participate in it.

3.13.2 Characteristics Resembling Modern RISC Architectures

The CPU ' s index register group, R0–R15, provides numerous registers for temporary data storage within the CPU itself. This reduces reliance on memory access, which can degrade performance—an idea remarkably similar to the general-purpose registers used in modern RISC (Reduced Instruction Set Computer) architectures. Quite fascinating, isn ' t it?

3.13.3 Learning from the Wisdom of the Past

In this way, the MCS-4 demonstrates a remarkably well-crafted architecture for a microcontroller. Its approach to hardware simplification offers valuable insights that can be applied to the design of future embedded systems. Studying it is genuinely enriching.

Chapter 4

Development Tool for 4004 Assembler Program

This chapter describes the Development Tool for 4004 Assmebler Program.

4.1 CPU Assembler Programming Tool ADS4004

To develop programs for the 4004, Intel provided an assembler back in the day[3]. However, since that tool is no longer available, I created my own this time. The tool I developed is called ADS4004, which integrates assembler, disassembler, and CPU simulator functionalities. All features are combined into a single application, written entirely in standard C, making it easy to build and run on virtually any platform.

The assembler I designed employs a two-pass method that supports labels. Labels are human-readable strings used to represent target addresses for branches or memory locations for constants, making programming more intuitive. As for literals (constant values), expressions involving addition, subtraction, multiplication, and division can be used—including those with labels.

During assembly, the first pass analyzes the entire program to determine the address values assigned to labels. In the second pass, the assembler generates instruction codes while referencing those assigned label values.

4.1.1 Building ADS4004

ADS4004 is written in standard C, making it easy to build on virtually any platform. This guide illustrates an example of building and running ADS4004 using `gcc` on WSL2 (Windows Subsystem for Linux 2) in a Windows 11 environment. If the necessary tools are not installed in your WSL2 setup, please refer to relevant documentation and install them accordingly.

First, clone the repository containing the project files from GitHub:

```
$ git clone https://github.com/munetomo-maruyama/MCS4_SYSTEM.git
```

Navigate to the ADS4004 source directory and compile it using `gcc`. The C source file for ADS4004 is `ads4004.c`:

```
$ cd MCS4_SYSTEM/SOFTWARE/ads4004
$ gcc -o ads4004.exe ads4004.c
```

Finally, run ADS4004 without any command-line arguments. If a help message is displayed as shown below, the build was successful:

```
$ ./ads4004.exe
Input File is not Specified.
------------------------------------------------------------------------
ADS4004 Command Usage
------------------------------------------------------------------------
$ ads4004 [options] InputFile
------------------------------------------------------------------------
Function Selector
    --asm, -a : Assembler (Default)
    --dis, -d : Disassembler
    --sim, -s : Simulator
------------------------------------------------------------------------
Assembler : InputFile is a Source List.
    --obj, -o : Object Hex File (Intel Hex) (Default: InputFile.hex)
    --lis, -l : Assemble List (Default: InputFile.asm)
------------------------------------------------------------------------
Dis Assembler : InputFile is a Object Hex File.
    --lis, -l : Assemble List (Default: InputFile.dis)
------------------------------------------------------------------------
Simulator : InputFile is a Object Hex File.
    --log     : Log File (Default: InputFile.sim)
    [Interactive Commands]
        Q     : Quit
        R     : System Reset
        G adr : Go until Specified Address
        S     : Step
        M bnk : Dump Data RAM in Specified Bank
        P     : Dump I/O Port Status of ROM/RAM
------------------------------------------------------------------------
```

4.2 ADS4004 Assembler Functionality

This section explains how to use the assembler functionality in ADS4004. Begin by creating a 4004 assembly source program. For demonstration purposes, a sample source file named `sample.src` is provided in the `ads4004` directory (see Listing 4.1). To invoke the assembler, use the `-a` option as shown below:

```
$ ./ads4004.exe -a sample.src
```

If no errors are encountered, the assembler will generate the following output files:

- A listing file: `sample.src.lis`
- A binary file: `sample.src.hex`

At the end of the listing file, a value table showing the addresses assigned to labels is displayed. The binary file is formatted using Intel HEX format.

By default, the listing and binary file names are derived by appending the `.lis` and `.obj` extensions to the input file name, respectively. However, the output file names can be customized using the `-l` and `-o` options.

The assembler syntax for ADS4004 is shown in Table 4.1. Please refer to this table along with the sample program when writing your own assembly code.

```
// Sample Source for ADS4004
    org 0x000
    jun RESET

; Label and Literal
LABEL0  123
```

```
LABEL1, 3 * (1 + 2)
LABEL2: 0x12, 0x34, 0x56
LABEL3  -5 // negative
LABEL4  0xa // negative

; Equation
LABELA= 456
LABELB= LABEL0 + LABEL1
LABELC equ LABELA * 3

; Origin
    = 0x040
    0x44 0x45 // literal

    org 0x050
    0x55 0x56 ; literal

    equ 0x060
    0x66 0x67 ; literals
RESET

; Machine Instruction (1-word)
MACHINE1
    nop
    ldm 0xa
    ldm 10
    ldm LABEL0
    ld 3
    ld 15
    xch 3
    xch 15
    add 3
    add 15
    sub 3
    sub 15
    inc 8
    inc 12
    bbl 0xa
    bbl 12
    jin 3p
    jin 3<
    src 5p
    src 5<
    src 10
    fin 7p
    fin 7<
    fin 14
; Machine Instruction (2-word)
MACHINE2
    jun MACHINE1
    jun 0x123
    jms MACHINE1
    jms 0x0abc
    jcn 0x4 MACHINE2
    jcn 9 0x0ab
    jcn TZ MACHINE2
    jcn tn MACHINE2
    jcn C0 MACHINE2
    jcn c1 MACHINE2
    jcn AZ MACHINE2
    jcn an MACHINE2
    isz 9 MACHINE2
    isz 9 0x0ab
    fim 1p 0xfe
    fim 1< 0xfe
    fim 2 0xfe
; Data Access Instruction
    rdm
    rd0
    rd1
    rd2
    rd3
    rdr
    wrm
    wr0
    wr1
```

```
    wr2
    wr3
    wrr
    wmp
    adm
    sbm
    wpm
; Accumulator Instruction
    clb
    clc
    cmc
    stc
    cma
    iac
    dac
    ral
    rar
    tcc
    daa
    tcs
    kbp
    dcl
; End of Assembler
    end
```

Listing 4.1: Assembly Source Code Example – `sample.src`

4.3 ADS4004 Disassembler Functionality

This section describes how to use the disassembler functionality of ADS4004. The input must be a binary file in Intel HEX format. Here, we'll disassemble the binary file `sample.src.hex`, which was generated by assembling the sample program introduced earlier.

To invoke the disassembler, run ADS4004 with the `-d` option as shown below:

```
$ ./ads4004.exe -d sample.src.hex
```

If successful, a disassembled listing file named `sample.src.hex.lis` will be produced. By default, the listing filename is created by appending the extension `.lis` to the input filename. However, you can specify a custom output filename using the `-l` option.

4.4 ADS4004 CPU Simulator Functionality

4.4.1 MCS-4 System Configuration for ADS4004 Simulation

The CPU simulator feature of ADS4004 is based on the system configuration shown in Table 4.2, which assumes an MCS-4 system. The ROM/RAM memory capacities are set to their maximum specifications. The system includes a feature for self-modification by structuring the ROM described under "Others" in Table 4.2 using RAM; this functionality will be discussed later.

4.4.2 How to use the CPU simulator in ADS4004

This section explains how to use the CPU simulator in ADS4004. The input must be a binary file in Intel HEX format. Note that the earlier sample program `sample.src` was intended only to demonstrate assembler syntax and does not function as a valid program.

For this simulation, we will use the program `test.src`, which was created to validate the full functionality of ADS4004.

Type		Meaning	Position in Source Line	Description Method	Example
Comment		Treats everything after as a comment.	Anywhere	;..... //.....	; This is a comment. // This is a comment,too. MAIN // Main Routine ldm 0xa ; A=0xa xch 0 // exchange
Number		Numeric Value	Within expressions or standalone	Values are expressed in either decimal or hexadecimal format. Hexadecimal values are prefixed with 0x. The letters A–F in hexadecimal may be written in either uppercase or lowercase. In decimal notation, adding a minus sign (-) at the beginning indicates a negative number. For hexadecimal values, if the most significant bit (MSB) is 1, the value is treated as negative.	10 -5 ; negative 0xc 0xAB // negative 0x0AB // positive
Expression		Arithmetic Expression	Within label assignment expressions or within literals	Arithmetic expressions involving numbers and labels. You can use the four standard arithmetic operators: +, -, *, and /. Parentheses () are also supported for grouping expressions.	10*20 (1+2)*3 LABEL+8
Label		A label is assigned the starting ROM address corresponding to its line. Labels are referenced as branch destinations or within expressions. Uppercase and lowercase letters in labels are distinguished—they are case-sensitive. Label values fall within the ROM address space, ranging from 0x000 to 0x3FF.	At beginning of line	Labels are represented by non-numeric character strings, and must be followed by a delimiter—either a space, tab, comma, or colon. Reserved keywords (instruction mnemonics and pseudo-instruction mnemonics) cannot be used as labels. Lines containing only a label are also permitted.	RESET ldm 0x4 LOOP jcn tz LOOP add 14
Literal (Constant)		Stores specified literals (constants) starting from the beginning ROM address corresponding to the line. Each literal constant is 8 bits wide.	Start after inserting one or more spaces or tabs at the beginning of the line.	Expressions or numeric values are listed, separated by spaces, tabs, or commas.	TABLE 123 0xab 0xcd (1+2)*3,0x12,0x23 TABLE+4 255
Pseudo Instruction	Assignment of numeric values to labels	Assign a numeric value to a label using the expression on the right-hand side. Be aware that the valid range of a label's value depends on how it is referenced—whether in a literal or as part of an instruction.	Begin at the start of the line, including the label.	Label=Expression Label equ Expression Label EQU Expression	CONST0=0x12 CONST1 equ CONST0*11
	Origin (Start Address)	Directly specify the starting ROM address for the next line.	Begin after inserting one or more spaces or tabs at the start of the line.	org ROM Address ORG ROM Address or (without any label) = ROMAddress equ ROM Address EQU ROM Address	org 0x000 RESET jun MAIN … =0x200 MAIN ldm 0xa … equ 0x300 FUNC ldm 0xb
	End of Assemble	Ends assembly process at beginning of line	Begin after inserting one or more spaces or tabs at the start of the line.	end END	LOOP … jun LOOP end
CPU Instructions		Follow the CPU instruction table. Instruction mnemonics may be written in either uppercase or lowercase.			

Table 4.1: ADS4004 Assembler Syntax

The assembler syntax used is an original design by the author for the 4004.

Category	Item	Description
ROM	Number of Chips	8 units (#0 to #7)
	Total Capacity	4096 bytes (0x000 to 0xFFF)
	Input Port	4 bits per chip (independent from output port)
	Output Port	4 bits per chip (independent from input port)
	Number of Banks	8 banks (#0 to #7)
RAM	Chips per Bank	4 chips per bank, total of 32 units
	Total Capacity	2560 nibbles = 1280 bytes
	Input Port	4 bits per chip
	Output Port	4 bits per chip
Others	Support for self-modification (WPM instruction) when part of the program memory (typically built with 4001 ROM) is implemented using RAM	

Table 4.2: MCS-4 System Configuration for ADS4004 Simulation

First, assemble the program:

```
$ ./ads4004.exe -a test.src
```

Next, launch the ADS4004 simulator using the `-s` option:

```
$ ./ads4004.exe -s test.src.hex
```

The simulation results are output to the log file `test.src.hex.sim`, which can be reviewed and analyzed later. By default, the log filename is generated by appending the extension `.sim` to the input filename. However, you can specify a custom log filename using the `-log` option.

Upon launching the simulator, the following output will appear. All values are displayed in hexadecimal:

```
PC=000 ACC=0 CY=0 R0-15=0000000000000000 DCL=0 SRC=00  (000 40 70    JUN 0x070)
### Q/R/G adr/S/M bnk/P >
```

The line beginning with `PC=...` displays the internal state of the CPU resources—ACC, CY, R0–R15, DCL, SRC—immediately before executing the instruction at address 0x000. The instruction code and its disassembled result are shown in parentheses at the far right.

4.4.3 Interractive Commands of CPU simulator in ADS4004

When the display begins with `###...`, the simulator is waiting for interactive command input. The available interactive commands are shown in Table 4.3.

These commands include:

- CPU reset
- Execution of instructions until a specified address
- Single-step instruction execution
- Display of RAM contents

Command (case-insensitive)	Function	Remarks
`Q`	Exit the simulation	–
`R`	Reset the CPU and stop execution	–
`G adr`	Run until the Program Counter (PC) reaches address `adr`	Press `Ctrl-C` to interrupt. Address range (adr): `000`–`FFF` (no `0x` prefix used).
`S`	Execute one instruction step	–
`M bnk`	Display data in RAM bank `bnk`	Bank number (bnk): `0`–`7`
`P`	Show the state of ROM/RAM I/O ports	–

Table 4.3: ADS4004 Simulator Command List

- Display of ROM/RAM I/O port states

If instruction execution results in abnormal behavior or if you wish to interrupt execution midway, press `Ctrl-C` to return to the interactive command input prompt.

The program `test.src` ultimately halts at an infinite loop instruction `JUN FINISH` located at address `0x192`. To simulate up to this address, enter the command `g 192`:

```
### Q/R/G adr/S/M bnk/P >g 192
  PC=070 ACC=0 CY=0 R0-15=0000000000000000 DCL=0 SRC=00  (070 00       NOP)
  PC=071 ACC=0 CY=0 R0-15=0000000000000000 DCL=0 SRC=00  (071 68       INC 8)
...
  PC=0FD ACC=F CY=1 R0-15=A20640EEAB00A040 DCL=0 SRC=00  (0FD D0       LDM 0x0)
  PC=0FE ACC=0 CY=1 R0-15=A20640EEAB00A040 DCL=0 SRC=00  (0FE 11 01    JCN TZ 0x01)
>>>Input TEST Pin Level in Hex (Now 0, RET to unchanged)=
```

At this point, the message `>>Input TEST Pin Level...` appears and execution halts. This occurs before executing the conditional branch instruction `JCN TZ 0x01`, which depends on the level of the TEST pin. To continue simulation, you must specify the input level of the TEST pin (either `0` or `1`). If the same value as the previous input is desired (initial value is `0`), simply press Enter.

After repeating this process several times, the following prompt appears:

```
...
  PC=15C ACC=0 CY=1 R0-15=A23440EEAA00A040 DCL=2 SRC=60  (15C 25       SRC 2P)
  PC=15D ACC=0 CY=1 R0-15=A23440EEAA00A040 DCL=2 SRC=40  (15D EA       RDR)
>>>Input ROM(4) Port Level in Hex (Now 0, RET to unchanged)=
```

Execution stops again before reaching address `0x192`. This time, the simulator halts prior to executing the `RDR` instruction, which reads the input level of ROM chip #4. To proceed, enter the 4-bit hexadecimal input level (`0`–`F`) for the ROM input port. Pressing Enter without input will reuse the previous value (default is `0`).

Repeat this interaction as needed. Once execution reaches the target address, the simulator returns to the interactive command prompt. To re-run the simulation from the beginning, use the reset command `R`. To exit, use the quit command `Q`.

Note: While awaiting input for the TEST pin or ROM input port, interactive commands listed in Table 4.3 are *not accepted*.

4.5 4004 CPU Sample Program (1): Memory Access Program – `memtst.src`

4.5.1 Memory Access Program `memtst.src`

As a sample program, the memory access program `memtst.src` is shown in Listing 4.2. It demonstrates reading and writing operations to the main memory characters of RAM, the status characters of RAM, reading from ROM input ports, writing to ROM output ports, and writing to RAM output ports. These instruction sequences are useful as reference patterns for handling such tasks.

Furthermore, the program includes examples using the special instruction `WPM`, designed for reading and writing to program memory constructed from RAM devices. The details of the `WPM` instruction are explained below.

```
// Memory Access Test for ADS4004
        org 0x000 //
MEMTST
        ; RAM W&R
        ldm 1
        dcl
        fim 1P 0x34
        src 1P
        ldm 0xa
        wrm
        ldm 0x5
        rdm

        ; RAM Status Character(RSC) W&R
        ldm 2
        dcl
        fim 2P 0x60
        src 2P
        ldm 0xb
        wr2
        ldm 0x4
        rd2
        rd0

        ; Read ROM Port
        fim 2P 0x40
        src 2p
        rdr

        ; Write ROM Port
        fim 2P 0xc0
        src 2p
        ldm 10
        wrr

        ; Write RAM Port
        ldm 3 ; bank4
        dcl
        fim 3p 0x55
        src 3p
        wmp

        ; ADD RAM
        ldm 2
        dcl
        fim 1P 0x56
        src 1P
        ldm 0xa
        wrm
        stc ; Carry=1
        ldm 0x6
        adm

        ; SUB RAM
        ldm 2
        dcl
```

```
        fim 1P 0x57
        src 1P
        ldm 0x5
        wrm
        stc ; Borrow=0
        ldm 0x6
        sbm

        ; Program RAM R&W
        ldm 0
        dcl
        fim 0p 0
        src 0p
        ldm 0x0
        wr0
        ldm 0x2
        wr1
        ldm 0x4
        wr2
        ;
        jms PGM_RD
        ;
        fim 1P 0xab
        jms PGM_WR
        ;
        fim 1P 0x0
        jms PGM_RD

// End of Test
FINISH jun FINISH

// Program RAM, Read & Write
//   Program RAM Location = Bank0, Chip0, Reg0: {RSC0, RSC1, RSC2}
//   Program RAM Data = {R2:R3}
PGM_WR
        fim 0p 224
        src 0p
        ldm 1
        wrr
        jms PGMCOM
        ld  2
        wpm
        ld  3
        wpm
        fim 0p 224
        src 0p
        clb
        wrr
        bbl 0
PGM_RD
        jms PGMCOM
        wpm
        wpm
        fim 0p 224
        src 0p
        rdr
        xch 2
        inc 0
        src 0p
        rdr
        xch 3
        bbl 0
PGMCOM
        fim 0p 0
        src 0p
        rd1
        xch 10
        rd2
        xch 11
        rd0
        fim 0p 240
        src 0p
        wrr
        src 5p
        bbl 0
```

```
        end  ; end of assemble source
```

Listing 4.2: Memory Access Program – `memtst.src`

4.5.2 Special Instruction `WPM` for Program RAM Access

The `WPM` instruction is used to access "program RAM"—a portion of program memory typically built from 4001 ROM, but here implemented using RAM devices. Although the execution timing of `WPM` follows that of WR-type instructions (as shown in Figure 3.4), it is used not only for writing but also for reading program RAM.

4.5.3 External Hardware Support Required for `WPM`

Accessing program RAM via `WPM` requires external hardware support. This hardware monitors CPU fetch cycles for `WPM` and performs the necessary operations. It uses the I/O ports of ROM chips #14 and #15 for reading and writing to program RAM. Since both input and output ports are used simultaneously, the standard 4001 cannot handle this. Instead, the support hardware must emulate the port functionality of chips #14 and #15.

The ADS4004 simulator emulates this integrated behavior, including the interaction between the `WPM` instruction and support hardware. Additionally, it allows for reading and writing the ROM content of 4001 chips—functionality not available in actual hardware.

4.5.4 Operation When Writing to Program RAM via `WPM`

The following sequence is performed when writing to program RAM:

1. Write the value `0x1` to the output port of ROM chip #14 to signal the start of a program RAM write.
2. Write the upper 4 bits of the 12-bit program RAM address to the output port of ROM chip #15.
3. Output the lower 8 bits of the address via the `SRC` instruction to the data bus.
4. Perform two executions of the `WPM` instruction:
   - First `WPM` writes the upper 4 bits from the accumulator (ACC).
   - Second `WPM` writes the lower 4 bits from ACC.

   The hardware counts the executions to determine which half is being written.
5. Write the value `0x0` to the ROM chip #14 output port to signal the end of write operation.

4.5.5 Operation When Reading from Program RAM via `WPM`

Reading from program RAM via `WPM` follows this sequence:

1. Write the upper 4 bits of the 12-bit program RAM address to the output port of ROM chip #15.
2. Output the lower 8 bits of the address via the `SRC` instruction.

3. Perform two executions of the WPM instruction:

   - First WPM causes the upper 4 bits of data to appear at ROM chip #14's input port.
   - Second WPM causes the lower 4 bits to appear at ROM chip #15's input port.

   These values are then acquired using the RDR instruction. The support hardware handles the bit selection based on execution count.

4.5.6 Accessing Program RAM in Sample Program memtst.src

In Listing 4.2, the sample program writes to program RAM by encoding the 12-bit target address into RAM bank #0, chip #0, register #0 status characters:

- Upper 4 bits → status character 0
- Middle 4 bits → status character 1
- Lower 4 bits → status character 2

The write data (8 bits) is stored in index register pair R1P (R2, R3) before calling the subroutine PGM_WR.

To read from program RAM, the same address encoding is performed, followed by calling PGM_RD, which places the read data into R1P.

4.5.7 Simulation of Sample Program memtst.src

Let's simulate memtst.src using ADS4004:

```
$ ./ads4004.exe -a memtst.src
$ ./ads4004.exe -s memtst.src.hex
```

The program ends at address 0x4B, so execute until then:

```
### Q/R/G adr/S/M bnk/P >g 4b
```

During execution, the program will request input for ROM chip #4 ' s input port during an RDR instruction. You can press Enter to accept the default value.

```
>>>Input ROM(4) Port Level in Hex (Now 0, RET if unchanged)=
```

In the subroutine PGM_RD, inputs to ROM chips #14 and #15 are requested prior to RDR execution. The simulator automatically provides the correct program memory value to the corresponding input port, so pressing Enter is sufficient:

```
>>>Input ROM(14) Port Level in Hex (Now 2, RET if unchanged)=
...
>>>Input ROM(15) Port Level in Hex (Now 2, RET if unchanged)=
...
>>>Input ROM(14) Port Level in Hex (Now a, RET if unchanged)=
...
>>>Input ROM(15) Port Level in Hex (Now b, RET if unchanged)=
```

4.6 4004 CPU Sample Program (2): BCD to Binary Conversion – `bcd2bin.src`

4.6.1 BCD to Binary Conversion – `bcd2bin.src`

This sample program, `bcd2bin.src`, performs conversion from BCD code (Binary Coded Decimal) to binary (BIN). The source program is shown in Listing 4.3. The supported BCD range is from 0 to 255. For example, BCD value `123` is converted to binary `0x7B`.

```
// BCD to BIN for ADS4004
    org 0x000

    // Main Routine
MAIN
    // Store BCD in Bank0/Chip0/Reg0/Chr2-Chr0
    ldm 0
    dcl
    fim 0p 0x02
    src 0p
    ldm 0x1   ; MSB of BCD Code
    wrm
    fim 0p 0x01
    src 0p
    ldm 0x2   ; Middle of BCD Code
    wrm
    fim 0p 0x00
    src 0p
    ldm 0x3   ; LSB of BCD Code
    wrm
    // Call BCD to BIN Converter
    jms BCDBIN
    // Store BIN Result in Bank0/Chip0/Reg0/Chr4-Chr3
    fim 0p 0x04
    src 0p
    xch 2     ; MSB of R1P (Result)
    wrm
    fim 0p 0x03
    src 0p
    xch 3     ; LSB of R1P (Result)
    wrm
FINISH
    jun FINISH

// Convert BCD to Binary
BCDBIN
    fim 0p 0   ; R0P=0
BCDBIN_1 // 1st Digit
    fim 1p 0   ; R1P=0 (Result)
    src 0p
    rdm        ; Read 1st (LSB) BCD Digit
    xch 3      ; R1P(LSB)=1st BCD Digit
BCDBIN_2 // 2nd Digit
    fim 2p 10  ; R2P=10, for 2nd Digit
    jms BCDADD ; Repeat (R1P=R1P+R2P) for Digit times
BCDBIN_3 // 3rd Digit
    fim 2p 100 ; R2P=100, for 3rd Digit
    jms BCDADD ; Repeat (R1P=R1P+R2P) for Digit times
BCDBIN_END
     bbl 0     ; Return
BCDADD
    inc 1      ; R1=R1+1
    src 0p     ; Prepare to Read Next Digit
BA1
    rdm        ; Read the BCD Digit
    jcn az BA2 ; if ACC==0, jump to BA2
    dac        ; ACC=ACC-1
    wrm        ; Write Back the Decremented BCD Digit to RAM
    clc        ; R1P=R1P+R2P
    ld 3       ;   ACC=R3
    add 5      ;   ACC=ACC+R5
    xch 3      ;   R3=ACC
    ld 2       ;   ACC=R2
    ADD 4      ;   ACC=ACC+R4
```

```
    xch 2       ;   R2=ACC
    jun BA1     ; jump to BA1
BA2
    bbl 0       ; Return

    end
```

Listing 4.3: BCD to Binary Conversion – bcd2bin.src

4.6.2 Conversion Algorithm

The conversion algorithm is simple. Take the BCD code `123` as an example. The binary result is calculated as follows:

- Store the lowest BCD digit (`3`) in the binary result.
- Add `10` to the result for each count of the middle BCD digit (`2` times).
- Add `100` to the result for each count of the highest BCD digit (`1` time).

4.6.3 Program Description

The core subroutine for conversion is named `BCDBIN`. The input BCD code is stored in RAM bank #0 at SRC addresses `0x00`–`0x02` (from least to most significant digit). After calling `BCDBIN`, the converted binary value is stored at addresses `0x03`–`0x04`, also in bank #0, with the lower byte first.

4.6.4 Simulation Using ADS4004

To simulate the sample program using ADS4004:

```
$ ./ads4004.exe -a bcd2bin.src
$ ./ads4004.exe -s bcd2bin.src.hex
```

Run the program up to address `0x11`, where the BCD input is set and `BCDBIN` is called:

```
### Q/R/G adr/S/M bnk/P >g 11
  PC=001 ACC=0 CY=0 R0-15=0000000000000000 DCL=0 SRC=00  (001 FD       DCL)
  PC=002 ACC=0 CY=0 R0-15=0000000000000000 DCL=0 SRC=00  (002 20 02    FIM 0 0x02)
...
  PC=010 ACC=3 CY=0 R0-15=0000000000000000 DCL=0 SRC=00  (010 E0       WRM)
  PC=011 ACC=3 CY=0 R0-15=0000000000000000 DCL=0 SRC=00  (011 50 1F    JMS 0x01F)
### Q/R/G adr/S/M bnk/P >m 0
Bank=0 Chip=0 Reg=0 Addr=000  3 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0  Status= 0 0 0 0
Bank=0 Chip=0 Reg=1 Addr=010  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  Status= 0 0 0 0
Bank=0 Chip=0 Reg=2 Addr=020  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  Status= 0 0 0 0
...
```

Now continue execution up to address `0x1D` and verify that the binary result `0x7B` is stored correctly:

```
### Q/R/G adr/S/M bnk/P >g 1d
  PC=01F ACC=3 CY=0 R0-15=0000000000000000 DCL=0 SRC=00  (01F 20 00    FIM 0 0x00)
  PC=021 ACC=3 CY=0 R0-15=0000000000000000 DCL=0 SRC=00  (021 22 00    FIM 2 0x00)
...
  PC=01B ACC=7 CY=0 R0-15=030B640000000000 DCL=0 SRC=03  (01B B3       XCH 3)
  PC=01C ACC=B CY=0 R0-15=0307640000000000 DCL=0 SRC=03  (01C E0       WRM)
### Q/R/G adr/S/M bnk/P >m 0
Bank=0 Chip=0 Reg=0 Addr=000  3 0 0 b 7 0 0 0 0 0 0 0 0 0 0 0  Status= 0 0 0 0
Bank=0 Chip=0 Reg=1 Addr=010  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  Status= 0 0 0 0
Bank=0 Chip=0 Reg=2 Addr=020  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  Status= 0 0 0 0
```

Follow the progression of CPU resources and memory state to deepen your understanding of the 4004 CPU operation and verify correctness of your implementation.

Chapter 5

Logic Design of MCS-4 Chips

This chapter presents design examples of each chip in the MCS-4 system—namely, the 4004 (CPU), 4001 (ROM), 4002 (RAM), and 4003 (Shift Register)—implemented using Verilog HDL.

The RTL descriptions for each MCS-4 chip are stored in the directory `RTL/MCS4/`.

5.1 Differences Between Physical Chip Design and RTL Design

In this project, each chip of the MCS-4 system is implemented with equivalent functionality using Verilog HDL and deployed onto an FPGA. However, due to the legacy nature of the original chip design, the following modifications were necessary:

- **System Clock:** The original chips employ a latch-based design and utilize a two-phase non-overlapping clock system. In contrast, the FPGA implementation adopts D flip-flop–based circuits synchronized with the rising edge of the clock, using a single-phase clock.
- **Data Bus:** The 4004 (CPU), 4001 (ROM), and 4002 (RAM) chips feature a 4-bit bidirectional data bus. When integrating these chips into a single FPGA, bidirectional buses cannot be directly implemented. Therefore, the data bus `DATA[3:0]` is divided into an input bus `DATA_I[3:0]` and an output bus `DATA_O[3:0]`, along with an output enable signal `DATA_OE` to indicate when the output bus is active.
- **External Data Bus Connection:** If the data bus is routed to an external device outside the FPGA, it will be configured as a bidirectional bus. To prevent signal contention, it is driven in open-drain mode and internally equipped with a pull-up resistor within the FPGA.

5.2 Internal States of MCS-4 Chips and System Synchronization

In the MCS-4 system, all chips—including the 4004 (CPU), 4001 (ROM), and 4002 (RAM)—must operate in synchronization. The CPU has eight internal states:

- A1, A2, A3
- M1, M2

- X1, X2, X3

It is essential that the internal state of each chip remains fully synchronized with the CPU.

5.2.1 State Signal Definition

To represent internal states, an 8-bit signal `state[7:0]` is defined. Each of the eight bits corresponds to a specific state. For example:

- A1: `8'b0000_0001`
- X3: `8'b1000_0000`

At no time are multiple bits set simultaneously. Each chip (CPU, ROM, and RAM) shares the same `state[7:0]` signal, which must be synchronized across the system.

5.2.2 System Synchronization Using `sync_n`

The method of synchronizing internal states is illustrated in Figure 5.1. The system synchronization signal `sync_n` is used.

Figure 5.1 shows the transition from system reset to the beginning of CPU instruction execution. During reset, `state[7:0]` is cleared to all zeros.

At the rising edge of the first clock after reset is released:

- CPU sets `state_next[7:0]` to A1
- `sync_n` is asserted

Afterward, the CPU sequentially updates `state_next[7:0]` and transfers it to `state[7:0]` on each rising clock edge, advancing the internal state.

On the ROM/RAM side:

- When `sync_n` is asserted at the clock ' s rising edge, `state[7:0]` is set to A1
- Then the internal states progress in sync with the CPU

This ensures all chips start execution in a synchronized manner.

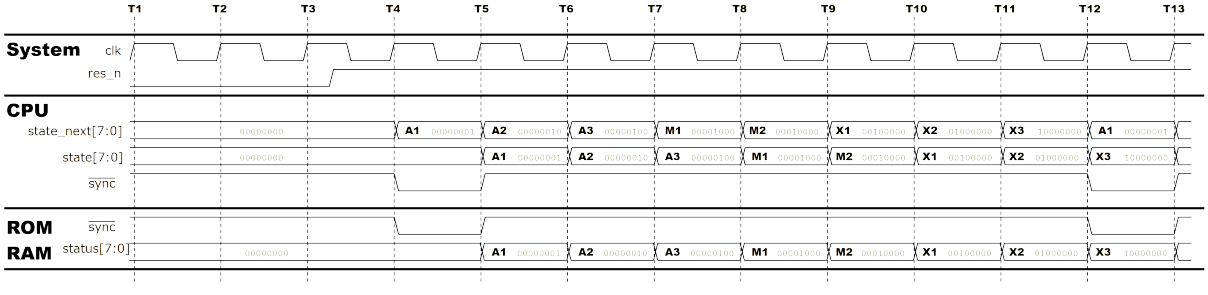


Figure 5.1: System synchronization method using `SYNC` signal

5.3 Logic of the MCS4_CPU Module (4004)

The entire CPU datapath and control unit are described within the RTL code file mcs4_cpu.v. The implementation is compact, consisting of fewer than 1000 lines, including comments. The code is written with readability in mind, allowing for a clear understanding of its structure and behavior from the RTL alone. Table 5.1 lists the input and output signals of the MCS4_CPU module. Below is a brief explanation of each major section of the CPU design.

MCS4_CPU (mcs4_cpu.v)				
Group	Signal Name	In/Out	Description	Note
System	CLK	Input	Clock Input	
	RES_N	Input	Reset Input	
CPU	SYNC_N	Output	Synchronization Signal Output	
	DATA_I[3:0]	Input	Data Inputs	
	DATA_O[3:0]	Output	Data Outputs	
	DATA_OE	Output	Data Output Enable	
	CM_ROM_N	Output	ROM Command Control Output	
	CM_RAM_N[3:0]	Output	RAM Command Control Outputs	
	TEST	Input	Conditional Branch Test Condition Input	
Debug	PC[11:0]	Output	Program Counter Outputs for Debug	

Table 5.1: I/O Signals of MCS4_CPU

(1) Definition of System State

```
//---------------
// State Number
//---------------
`define A1 0
`define A2 1
`define A3 2
`define M1 3
`define M2 4
`define X1 5
`define X2 6
`define X3 7
```

Listing 5.1: Definition of System State

(2) Definition of input/output signals and internal signals

The 12-bit wide program counter (PC) is exposed externally to drive the LED indicators on the FPGA board.

```
//====================================
// Module : CPU Core
//====================================
module MCS4_CPU
(
    input  wire       CLK,      // clock
    input  wire       RES_N,    // reset_n
    //
    output wire       SYNC_N,   // Sync Signal
    input  wire [3:0] DATA_I,   // Data Input
    output wire [3:0] DATA_O,   // Data Output
    output wire       DATA_OE,  // Data Output Enable
    output wire       CM_ROM_N, // Memory Control for ROM
    output wire [3:0] CM_RAM_N, // Memory Control for RAM
    input  wire       TEST,     // Test Input
    //
    output wire [11:0] PC  // debug
);

integer i;
```

```
//--------------------
// Main CPU Resource
//--------------------
wire [11:0] pc;      // Program Counter
wire [11:0] pc_next; // Next PC
reg  [ 3:0] acc;     // Accumulator
reg         cy;      // Carry Borrow Flag
reg  [ 3:0] r[0:15]; // Index Registers
wire [ 3:0] rn;      // Index Register specified by opropa0
wire [ 7:0] rp;      // Index Register Pair specified by opropa0
reg  [ 2:0] dcl;     // Designate Comand Line
reg  [ 7:0] src;     // Send Register Control

assign PC = pc;
```

Listing 5.2: Definition of input/output signals and internal signals

(3) **Control of Internal State Transitions**

```
//---------------------------------
// State Count
//---------------------------------
reg  [7:0] state;
reg  [7:0] state_next;
reg        multi_cycle;
reg        multi_cycle_inc;
//
always @(posedge CLK, negedge RES_N)
begin
    if (~RES_N)
        state <= 8'b00000000;
    else
        state <= state_next;
end
//
always @*
begin
    casez(state)
        8'b00000000 : state_next = 8'b00000001;
        8'b10000000 : state_next = 8'b00000001;
        default     : state_next = {state[6:0], state[7]}; // rotate left
    endcase
end
```

Listing 5.3: Control of Internal State Transitions

(4) **Control for multi-cycle instructions**
The following describes the control for multi-cycle instructions requiring two instruction cycles. During the first cycle, `multi_cycle` is `1'b0`; in the second, it becomes `1'b1`.

```
//---------------------------------
// Control of 2-Cycle Instruction
//---------------------------------
always @(posedge CLK, negedge RES_N)
begin
    if (~RES_N)
        multi_cycle <= 1'b0;
    else if (multi_cycle_inc)
        multi_cycle <= ~multi_cycle;
end
```

Listing 5.4: Control for multi-cycle instructions

(5) **Generation of the `sync_n` signal**

```
//------------------------
// Generate Sync Signal
//------------------------
assign SYNC_N = ~state_next[`A1];
```

Listing 5.5: Generation of the sync_n signal

(6) **Program Counter (PC) and the Stack (Program Address Register)**

```
//----------------------------
// Program Counter and Stack
//----------------------------
reg  [11:0] stack[0:3];
reg  [ 1:0] sp;
wire [11:0] pc_plus_one;
reg         pc_inc;
reg         pc_set;
reg         pc_push;
reg         pc_pop;
//
assign pc = stack[sp];
//
always @(posedge CLK, negedge RES_N)
begin
    if (~RES_N)
    begin
        stack[0] <= 12'h000;
        stack[1] <= 12'h000;
        stack[2] <= 12'h000;
        stack[3] <= 12'h000;
    end
    else if (pc_inc | pc_set)
        stack[sp] <= pc_next; // PC
end
//
always @(posedge CLK, negedge RES_N)
begin
    if (~RES_N)
        sp <= 2'b00;
    else if (pc_push)
        sp <= sp + 2'b01;
    else if (pc_pop)
        sp <= sp + 2'b11; // minus one
end
```

Listing 5.6: Program Counter (PC) and the Stack (Program Address Register)

(7) **Instruction Fetch Processing**
States A1–A3 output instruction addresses; states M1–M2 capture instruction codes. Multi-cycle instructions fetch their second-cycle instruction codes here. Codes fetched in the first cycle are stored in `opropa0[7:0]`; in the second cycle, in `opropa1[7:0]`. Data fetch for the FIN instruction's second cycle occurs within index register logic.

```
//-------------------------
// Instruction Fetch
//-------------------------
reg  [7:0] opropa0;
reg  [7:0] opropa1;
wire [3:0] data_o_rom_addr;
reg        do_fin;
//
assign data_o_rom_addr = (state[`A1] & do_fin)? r[1] // Lower Bits
                       : (state[`A2] & do_fin)? r[0] // Middle Bits
                       : (state[`A3] & do_fin)? pc_plus_one[11:8]
                       : (state[`A1])? pc[ 3:0]
                       : (state[`A2])? pc[ 7:4]
```

```
                      : (state[`A3])? pc[11:8]
                      : 4'b0000;
//
always @(posedge CLK, negedge RES_N)
begin
    if (~RES_N)
    begin
        opropa0 <= 8'h00;
        opropa1 <= 8'h00;
    end
    else if ((state[`M1]) & ~multi_cycle & ~do_fin)
        opropa0[7:4] <= DATA_I; // OPR
    else if ((state[`M2]) & ~multi_cycle & ~do_fin)
        opropa0[3:0] <= DATA_I; // OPA
    else if ((state[`M1]) &  multi_cycle & ~do_fin)
        opropa1[7:4] <= DATA_I; // OPR
    else if ((state[`M2]) &  multi_cycle & ~do_fin)
        opropa1[3:0] <= DATA_I; // OPA
end
```

Listing 5.7: Instruction Fetch Processing

(8) **Determining the next PC value**
This process includes both simple PC increment and PC update via branch instruction.

```
//--------------------
// Next PC
//--------------------
reg pc_target_jcn;
reg pc_target_jun;
reg pc_target_jin;
//
assign pc_plus_one = pc + 12'h001;
assign pc_next = (pc_inc       )?  pc_plus_one
               : (pc_target_jcn)? {pc_plus_one[11:8], opropa1[7:0]}
               : (pc_target_jun)? {opropa0[3:0]     , opropa1[7:0]}
               : (pc_target_jin)? {pc_plus_one[11:8], rp}
               : pc;
```

Listing 5.8: Determining the next PC value

(9) **Arithmetic Logic Unit (ALU) operations**
The ALU behavior changes based on control signals and receives data from various resources like the accumulator (ACC), fetched instructions (`opropa0`, `opropa1`), and data bus (`data_i[3:0]`). Outputs include passthrough values, addition with carry, subtraction with borrow, rotate results, and DAA-adjusted values; carry/borrow signals (`alu_cy`) are also produced.

```
//---------------------------------
// ALU : Arithmetic Logical Unit
//---------------------------------
reg  [3:0] alu_a;
reg        alu_a_acc;
reg        alu_a_rn;
reg        alu_a_opropa;
//
reg  [3:0] alu_b;
reg        alu_b_acc;
reg        alu_b_rn;
reg        alu_b_data_i;
//
reg        alu_c;
reg        alu_c_cy;
reg        alu_c_set;
//
reg  [4:0] alu;
```

```
reg         alu_thru_a;
reg         alu_thru_b;
reg         alu_add;
reg         alu_sub;
reg         alu_ral;
reg         alu_rar;
reg         alu_daa;
//
always @*
begin
    casez({alu_a_acc, alu_a_rn, alu_a_opropa})
        3'b1??  : alu_a = acc;
        3'b?1?  : alu_a = rn;
        3'b??1  : alu_a = opropa0[3:0];
        default : alu_a = 4'b0000;
    endcase
end
//
always @*
begin
    casez({alu_b_acc, alu_b_rn, alu_b_data_i})
        3'b1??  : alu_b = acc;
        3'b?1?  : alu_b = rn;
        3'b??1  : alu_b = DATA_I;
        default : alu_b = 4'b0000;
    endcase
end
//
always @*
begin
    casez({alu_c_cy, alu_c_set})
        2'b1?   : alu_c = cy;
        2'b?1   : alu_c = 1'b1;
        default : alu_c = 1'b0;
    endcase
end
//
always @*
begin
    casez({alu_thru_a, alu_thru_b, alu_add, alu_sub, alu_ral, alu_rar, alu_daa})
        7'b1?????? : alu = {1'b0, alu_a};
        7'b?1????? : alu = {1'b0, alu_b};
        7'b??1???? : alu = {1'b0, alu_a} + {1'b0,  alu_b} + {4'b0000,  alu_c};
        7'b???1??? : alu = {1'b0, alu_a} + {1'b0, ~alu_b} + {4'b0000, ~alu_c};
        7'b????1?? : alu = {alu_a[3], alu_a[2:0], cy};
        7'b?????1? : alu = {alu_a[0], cy, alu_a[3:1]};
        7'b??????1 : alu = {1'b0, alu_a} + 5'b00110;
        default    : alu = 5'b0_0000;
    endcase
end
```

Listing 5.9: Arithmetic Logic Unit (ALU) operations

(10) **KBP instruction table conversion**
The conversion result of KBP instruction is `kbp[3:0]`.

```
//---------------
// KBP Table
//---------------
reg [3:0] kbp;
//
always @*
begin
    casez (acc)
         4'b0000 : kbp = 4'b0000;
         4'b0001 : kbp = 4'b0001;
         4'b0010 : kbp = 4'b0010;
         4'b0100 : kbp = 4'b0011;
         4'b1000 : kbp = 4'b0100;
         default : kbp = 4'b1111;
    endcase
end
```

Listing 5.10: KBP instruction table conversion

(11) **Processing for accumulator ACC**
Depending on control signals, inputs to ACC may be ALU output, or KBP output.

```
//-------------------------------
// ACC : Accumulator
//-------------------------------
reg acc_alu;
reg acc_kbp;
//
always @(posedge CLK, negedge RES_N)
begin
    if (~RES_N)
        acc <= 4'b0000;
    else if (acc_alu)
        acc <= alu[3:0];
    else if (acc_kbp)
        acc <= kbp;
end
```

Listing 5.11: Processing for accumulator ACC

(12) **Processing for CY (carry) bit**
Depending on control signals, CY is set from ALU output, or directly via `1'b0` or `1'b1`.

```
//-------------------
// Carry Flag CY
//-------------------
wire cy_next;
reg  cy_set;
reg  cy_inv;
reg  cy_wrt;
//
assign cy_next = alu[4];
//
always @(posedge CLK, negedge RES_N)
begin
    if (~RES_N)
        cy <= 1'b0;
    else if (cy_set)
        cy <= 1'b1;
    else if (cy_inv)
        cy <= ~cy;
    else if (cy_wrt)
        cy <= cy_next;
end
```

Listing 5.12: Processing for CY (carry) bit

(13) **Processing for index registers Rn and index register pair RnP**
Inputs to Rn or RnP may be fetched instruction codes, data bus, ALU output, or accumulator, depending on control signals.

```
//-------------------------------
// Rn : Index Registers
//-------------------------------
wire [3:0] rp0;
wire [3:0] rp1;
reg        rp_fim;
reg        rn_alu;
reg        rn_acc;
```

```
wire       rn_zero;
wire [3:0] rn_index;
wire [3:0] rn_index0;
wire [3:0] rn_index1;
//
assign rn_index  = opropa0[3:0];
assign rn_index0 = opropa0[3:0] & 4'b1110;
assign rn_index1 = opropa0[3:0] | 4'b0001;
//
always @(posedge CLK, negedge RES_N)
begin
    if (~RES_N)
        for (i = 0; i < 16; i = i + 1) r[i] = 4'b0000;
    else if (rp_fim)
    begin
        r[rn_index0] <= opropa1[7:4];
        r[rn_index1] <= opropa1[3:0];
    end
    else if (do_fin & state[`M1])
        r[rn_index0] <= DATA_I;
    else if (do_fin & state[`M2])
        r[rn_index1] <= DATA_I;
    else if (rn_alu)
        r[rn_index] <= alu[3:0];
    else if (rn_acc)
        r[rn_index] <= acc;
end
//
assign rn  = r[rn_index];
assign rp0 = r[rn_index0];
assign rp1 = r[rn_index1];
assign rp  = {rp0, rp1};
assign rn_zero = (rn == 4'b0000);
```

Listing 5.13: Processing for index registers Rn and index register pair RnP

(14) **Processing of DCL (Designate Command Line)**
Input to DCL is lower 3bits of accumulator value. `dcl_convert[3:0]` is generated to produce `cm_ram_n[3:0]`.

```
//--------------------------------
// DCL : Designate Command Line
//--------------------------------
reg        dcl_set;
wire [3:0] dcl_convert;
//
always @(posedge CLK, negedge RES_N)
begin
    if (~RES_N)
        dcl <= 3'b000;
    else if (dcl_set)
        dcl <= acc[2:0];
end
//
assign dcl_convert[0] = (dcl == 3'b000);
assign dcl_convert[1] = dcl[0];
assign dcl_convert[2] = dcl[1];
assign dcl_convert[3] = dcl[2];
```

Listing 5.14: Processing of DCL (Designate Command Line)

(15) **Processing of SRC (Send Register Control)**
SRC receives index register pair RnP, depending on control signals.

```
//--------------------------------
// SRC : Send Register Control
//--------------------------------
reg       src_set;
//
```

```
always @(posedge CLK, negedge RES_N)
begin
    if (~RES_N)
        src <= 8'h00;
    else if (src_set)
        src <= rp;
end
```

Listing 5.15: Processing of SRC (Send Register Control)

(16) **Generation of cm_rom_n signal**
`cm_rom_n` signal is generated based on internal state. If the fetched instruction is a data access instruction (`opropa0[7:4] == 4'b1110`) in state M1, the signal is asserted in state M2 as well.

```
//-------------------
// CM_ROM Output
//-------------------
wire cm_rom_at_a3; // Assert at A3 always
wire cm_rom_at_m2; // Assert at M2 of I/O and RAM Access Instruction
reg  cm_rom_at_x2; // Assert at X2 of SRC Instruction
//
assign cm_rom_at_a3 = state[`A3];
assign cm_rom_at_m2 = state[`M2] & (opropa0[7:4] == 4'b1110);
//
assign CM_ROM_N = ~cm_rom_at_a3  // NOR
                & ~cm_rom_at_m2
                & ~cm_rom_at_x2;
```

Listing 5.16: Generation of cm_rom_n signal

(17) **Generation of cm_ram_n[3:0] signal**
`cm_ram_n[3:0]` signal is generated using `dcl_convert[3:0]`, depending on internal states. As with ROM, the signal is also asserted in state M2 for data access instructions.

```
//-------------------
// CM_RAM Output
//-------------------
wire [3:0] cm_ram_at_a3; // Assert at A3 always
wire       cm_ram_at_m2; // Assert at M2 of I/O and RAM Access Instruction
reg        cm_ram_at_x2; // Assert at X2 of SRC Instruction
//
assign cm_ram_at_a3 = {4{state[`A3]}} & dcl_convert;
assign cm_ram_at_m2 = state[`M2] & (opropa0[7:4] == 4'b1110);
//
assign CM_RAM_N = ~cm_ram_at_a3 // NOR
                & ~({4{cm_ram_at_m2}} & dcl_convert)
                & ~({4{cm_ram_at_x2}} & dcl_convert);
```

Listing 5.17: Generation of cm_ram_n[3:0] signal

(18) **Generation of data bus output data_o[3:0]**
`data_o[3:0]` output is selected based on internal state and instruction control: ROM address fields, upper/lower halves of `src[7:0]`, or accumulator value.

```
//----------------------
// Data Bus Output
//----------------------
wire [3:0] data_o_src;
reg        data_o_src_at_x2;
reg        data_o_src_at_x3;
wire [3:0] data_o_acc;
```

```
reg        data_o_acc_at_x2;
//
assign data_o_src = (data_o_src_at_x2)? src[7:4]
                  : (data_o_src_at_x3)? src[3:0]
                  : 4'b0000;
assign data_o_acc = (data_o_acc_at_x2)? acc : 4'b0000;
//
assign DATA_O = data_o_rom_addr
              | data_o_src
              | data_o_acc;
//
assign DATA_OE = state[`A1] | state[`A2] | state[`A3]
               | data_o_src_at_x2 | data_o_src_at_x3
               | data_o_acc_at_x2;
```

Listing 5.18: Generation of data bus output data_o[3:0]

(19) **Handling of TEST input signal**
TEST signal is synchronized through one flip-flop stage before being passed to the instruction control unit.

```
//-----------------
// TEST Input
//-----------------
reg test_sync;
//
always @(posedge CLK, negedge RES_N)
begin
    if (~RES_N)
        test_sync <= 1'b0;
    else
        test_sync <= TEST;
end
```

Listing 5.19: Handling of TEST input signal

(20) **Generation of condition signals for the JCN (Jump Conditional) instruction**
Condition signals for the JCN (Jump Conditional) instruction are passed to the instruction control unit.

```
//-------------------
// Condition  for JCN
//-------------------
wire c1, c2, c3, c4;
wire jcn;
//
assign c1 = opropa0[3]; // C1: Invert Jump Condition
assign c2 = opropa0[2]; // C2: Jump if ACC==0
assign c3 = opropa0[1]; // C3: Jump if CY==1
assign c4 = opropa0[0]; // C4: Jump if TEST==0
//
assign jcn = c1 ^ (c2 & (acc == 4'b0000) | c3 & cy | c4 & ~test_sync);
```

Listing 5.20: Generation of condition signals for the JCN (Jump Conditional) instruction

(21) **Generation of DAA instruction correction conditions**
Correction conditions of DAA instruction is passed to the instruction control unit.

```
//-------------------
// Condition for DAA
//-------------------
// Input   |Output
```

```
// ACC  CY |ACC  CY
// --------+---------
// 0-9  0  |0-9  0 (No Carry, CY unchanged)
// 0-9  1  |6-F  1 (No Carry, CY unchanged)
// A-F  0  |0-5  1 (Carry, CY changed)
// A-F  1  |0-5  1 (Carry, CY changed)
wire daa;
//
assign daa = cy | (acc > 4'b1001);
```

Listing 5.21: Generation of DAA instruction correction conditions

(22) **Instruction control unit**
For each fetched instruction code, specific behavior is defined in response to the progress of `state[7:0]`. Control signals are asserted accordingly. Default assignments are made via blocking assignments, with selective overrides using `casez(opropa0)`.

```
//-------------------------
// Instruction Control
//-------------------------
always @*
begin
    // Default Control Signal Level
    pc_inc  = 1'b0;
    pc_set  = 1'b0;
    pc_push = 1'b0;
    pc_pop  = 1'b0;
    multi_cycle_inc = 1'b0;
    //
    dcl_set = 1'b0;
    src_set = 1'b0;
    //
    rp_fim  = 1'b0;
    rn_alu  = 1'b0;
    rn_acc  = 1'b0;
    //
    acc_alu = 1'b0;
    acc_kbp = 1'b0;
    //
    alu_a_acc    = 1'b0;
    alu_a_rn     = 1'b0;
    alu_a_opropa = 1'b0;
    alu_b_acc    = 1'b0;
    alu_b_rn     = 1'b0;
    alu_b_data_i = 1'b0;
    alu_c_cy     = 1'b0;
    alu_c_set    = 1'b0;
    alu_thru_a   = 1'b0;
    alu_thru_b   = 1'b0;
    alu_add      = 1'b0;
    alu_sub      = 1'b0;
    alu_ral      = 1'b0;
    alu_rar      = 1'b0;
    alu_daa      = 1'b0;
    //
    cy_set = 1'b0;
    cy_inv = 1'b0;
    cy_wrt = 1'b0;
    //
    cm_rom_at_x2 = 1'b0;
    cm_ram_at_x2 = 1'b0;
    data_o_src_at_x2 = 1'b0;
    data_o_src_at_x3 = 1'b0;
    data_o_acc_at_x2 = 1'b0;
    //
    pc_target_jcn = 1'b0;
    pc_target_jun = 1'b0;
    pc_target_jin = 1'b0;
    //
    do_fin = 1'b0;
    //
    // Set Contol Signals for each Instruction Sequence
```

```
casez(opropa0)
    //-------------------------------------
    // NOP : No Operation
    8'b0000_0000 :
    begin
        pc_inc  = state[`X3];
    end
    //-------------------------------------
    // JCN : Jump Conditional
    8'b0001_???? :
    begin
        if (~multi_cycle | ~jcn)
        begin
            pc_inc = state[`X3];
        end
        else //  if (multi_cycle & jcn)
        begin
            pc_target_jcn = state[`X3];
            pc_set        = state[`X3];
        end
        //
        multi_cycle_inc = state[`X3];
    end
    //-------------------------------------
    // FIM : Fetch Immediate from ROM
    8'b0010_???0 :
    begin
        if (~multi_cycle)
        begin
           // do nothing
        end
        else
        begin
            rp_fim = state[`X3];
        end
        //
        multi_cycle_inc = state[`X3];
        pc_inc  = state[`X3];
    end
    //-------------------------------------
    // SRC : Send Register Control
    8'b0010_???1 :
    begin
        src_set          = state[`X1];
        cm_rom_at_x2     = state[`X2];
        cm_ram_at_x2     = state[`X2];
        data_o_src_at_x2 = state[`X2];
        data_o_src_at_x3 = state[`X3];
        pc_inc           = state[`X3];
    end
    //-------------------------------------
    // FIN : Fetch Indirect from ROM
    8'b0011_???0 :
    begin
        if (~multi_cycle)
        begin
           // do nothing
        end
        else
        begin
            do_fin = 1'b1;
        end
        //
        multi_cycle_inc = state[`X3];
        pc_inc  = state[`X3] & multi_cycle;
    end
    //-------------------------------------
    // JIN : Jump Indirect
    8'b0011_???1 :
    begin
        pc_target_jin = state[`X3];
        pc_set        = state[`X3];
    end
    //-------------------------------------
    // JUN : Jump Unconditional
    8'b0100_???? :
```

```
        begin
            if (~multi_cycle)
            begin
                pc_inc = state[`X3];
            end
            else // if (multi_cycle)
            begin
                pc_target_jun = state[`X3];
                pc_set        = state[`X3];
            end
            //
            multi_cycle_inc = state[`X3];
        end
        //--------------------------------------
        // JMS : Jump to Subroutine
        8'b0101_???? :
        begin
            if (~multi_cycle)
            begin
                pc_inc = state[`X3];
            end
            else // if (multi_cycle)
            begin
                pc_inc        = state[`X2];
                pc_push       = state[`X2];
                //
                pc_target_jun = state[`X3];
                pc_set        = state[`X3];
            end
            //
            multi_cycle_inc = state[`X3];
        end
        //--------------------------------------
        // INC : Increment Index Register
        8'b0110_???? :
        begin
            alu_a_rn  = state[`X3];
            alu_c_set = state[`X3];
            alu_add   = state[`X3];
            rn_alu    = state[`X3];
            pc_inc    = state[`X3];
        end
        //--------------------------------------
        // ISZ : Increment Index Register, Skip if Zero
        8'b0111_???? :
        begin
            if (~multi_cycle)
            begin
                alu_a_rn  = state[`X3];
                alu_c_set = state[`X3];
                alu_add   = state[`X3];
                rn_alu    = state[`X3];
                pc_inc    = state[`X3];
            end
            else
            begin
                pc_target_jcn = (~rn_zero)? state[`X3] : 1'b0;
                pc_set        = (~rn_zero)? state[`X3] : 1'b0;
                //
                pc_inc        = ( rn_zero)? state[`X3] : 1'b0;
            end
            //
            multi_cycle_inc = state[`X3];
        end
        //--------------------------------------
        // ADD : Add Index Register to ACC
        8'b1000_???? :
        begin
            alu_a_acc = state[`X3];
            alu_b_rn  = state[`X3];
            alu_c_cy  = state[`X3];
            alu_add   = state[`X3];
            acc_alu   = state[`X3];
            cy_wrt    = state[`X3];
            pc_inc    = state[`X3];
        end
```

```
        //--------------------------------------
        // SUB : Subtract Index Register from ACC
        8'b1001_???? :
        begin
            alu_a_acc = state[`X3];
            alu_b_rn  = state[`X3];
            alu_c_cy  = state[`X3];
            alu_sub   = state[`X3];
            acc_alu   = state[`X3];
            cy_wrt    = state[`X3];
            pc_inc    = state[`X3];
        end
        //--------------------------------------
        // LD : Load Index Register to ACC
        8'b1010_???? :
        begin
            alu_a_rn   = state[`X3];
            alu_thru_a = state[`X3];
            acc_alu    = state[`X3];
            pc_inc     = state[`X3];
        end
        //--------------------------------------
        // XCH : Exchange Load Index Register and ACC
        8'b1011_???? :
        begin
            rn_acc     = state[`X3];
            alu_a_rn   = state[`X3];
            alu_thru_a = state[`X3];
            acc_alu    = state[`X3];
            pc_inc     = state[`X3];
        end
        //--------------------------------------
        // BBL : Branch Back and Load to ACC
        8'b1100_???? :
        begin
            pc_pop       = state[`X2];
            alu_a_opropa = state[`X3];
            alu_thru_a   = state[`X3];
            acc_alu      = state[`X3];
            pc_set       = state[`X3];
        end
        //--------------------------------------
        // LDM : Load Imm4 to ACC
        8'b1101_???? :
        begin
            alu_a_opropa = state[`X3];
            alu_thru_a   = state[`X3];
            acc_alu      = state[`X3];
            pc_inc       = state[`X3];
        end
        //--------------------------------------
        // WRM : Write RAM_CH from ACC             8'b1110_0000
        // WMP : Write RAM Output Port from ACC    8'b1110_0001
        // WRR : Write ROM Output Port from ACC    8'b1110_0010
        // WPM : Write R/W Program Memory from ACC 8'b1110_0011
        // WR0 : Write RAM Status 0 from ACC       8'b1110_0100
        // WR1 : Write RAM Status 1 from ACC       8'b1110_0101
        // WR2 : Write RAM Status 2 from ACC       8'b1110_0110
        // WR3 : Write RAM Status 3 from ACC       8'b1110_0111
        8'b1110_0??? :
        begin
            data_o_acc_at_x2 = state[`X2];
            pc_inc           = state[`X3];
        end
        //--------------------------------------
        // RDM : Read RAM_CH to ACC         8'b1110_1001
        // RDR : Read ROM Input Port to ACC 8'b1110_1010
        // RD0 : Read RAM Status 0 into ACC 8'b1110_1100
        // RD1 : Read RAM Status 1 into ACC 8'b1110_1101
        // RD2 : Read RAM Status 2 into ACC 8'b1110_1110
        // RD3 : Read RAM Status 3 into ACC 8'b1110_1111
        8'b1110_1001,
        8'b1110_1010,
        8'b1110_11?? :
        begin
            alu_b_data_i = 1'b1;
```

```
        alu_thru_b   = 1'b1;
        acc_alu      = state[`X2];
        pc_inc       = state[`X3];
    end
    //--------------------------------------
    // SBM : Subtract RAM_CH from ACC
    8'b1110_1000 :
    begin
        alu_a_acc    = state[`X2];
        alu_b_data_i = state[`X2];
        alu_c_cy     = state[`X2];
        alu_sub      = state[`X2];
        acc_alu      = state[`X2];
        cy_wrt       = state[`X2];
        pc_inc       = state[`X3];
    end
    //--------------------------------------
    // ADM : Add RAM_CH to ACC
    8'b1110_1011 :
    begin
        alu_a_acc    = state[`X2];
        alu_b_data_i = state[`X2];
        alu_c_cy     = state[`X2];
        alu_add      = state[`X2];
        acc_alu      = state[`X2];
        cy_wrt       = state[`X2];
        pc_inc       = state[`X3];
    end
    //--------------------------------------
    // CLB : Clear Both ACC and CY
    8'b1111_0000 :
    begin
        acc_alu = state[`X3];
        cy_wrt  = state[`X3];
        pc_inc  = state[`X3];
    end
    //--------------------------------------
    // CLC : Clear CY
    8'b1111_0001 :
    begin
        cy_wrt = state[`X3];
        pc_inc = state[`X3];
    end
    //--------------------------------------
    // IAC : Increment ACC
    8'b1111_0010 :
    begin
        alu_a_acc = state[`X3];
        alu_c_set = state[`X3];
        alu_add   = state[`X3];
        acc_alu   = state[`X3];
        cy_wrt    = state[`X3];
        pc_inc    = state[`X3];
    end
    //--------------------------------------
    // CMC : Complement CY
    8'b1111_0011 :
    begin
        cy_inv = state[`X3];
        pc_inc = state[`X3];
    end
    //--------------------------------------
    // CMA : Complement ACC
    8'b1111_0100 :
    begin
        alu_b_acc = state[`X3];
        alu_c_set = state[`X3];
        alu_sub   = state[`X3];
        acc_alu   = state[`X3];
        pc_inc    = state[`X3];
    end
    //--------------------------------------
    // RAL : Rotate Left ACC and CY
    8'b1111_0101 :
    begin
        alu_a_acc = state[`X3];
```

```
            alu_ral   = state[`X3];
            acc_alu   = state[`X3];
            cy_wrt    = state[`X3];
            pc_inc    = state[`X3];
        end
        //--------------------------------------
        // RAR : Rotate Right ACC and CY
        8'b1111_0110 :
        begin
            alu_a_acc = state[`X3];
            alu_rar   = state[`X3];
            acc_alu   = state[`X3];
            cy_wrt    = state[`X3];
            pc_inc    = state[`X3];
        end
        //--------------------------------------
        // TCC : Transmit CY to ACC and Clear CY
        8'b1111_0111 :
        begin
            alu_c_cy = state[`X3];
            alu_add  = state[`X3]; // add only CY
            acc_alu  = state[`X3];
            cy_wrt   = state[`X3];
            pc_inc   = state[`X3];
        end
        //--------------------------------------
        // DAC : Decrement ACC
        8'b1111_1000 :
        begin
            alu_a_acc = state[`X3];
            alu_c_set = state[`X3];
            alu_sub   = state[`X3];
            acc_alu   = state[`X3];
            cy_wrt    = state[`X3];
            pc_inc    = state[`X3];
        end
        //--------------------------------------
        // TCS : Transfer CY Subtract and Clear CY
        8'b1111_1001 :
        begin
            alu_a_opropa = state[`X3];
            alu_c_cy     = state[`X3];
            alu_add      = state[`X3];
            acc_alu      = state[`X3];
            cy_wrt       = state[`X3];
            pc_inc       = state[`X3];
        end
        //--------------------------------------
        // STC : Set CY
        8'b1111_1010 :
        begin
            cy_set = state[`X3];
            pc_inc = state[`X3];
        end
        //--------------------------------------
        // DAA : Decimal Adjust ACC
        8'b1111_1011 :
        begin
            alu_a_acc = state[`X3];
            alu_daa   = state[`X3];
            acc_alu   = state[`X3] & daa;
            cy_wrt    = state[`X3] & cy_next; // if non carry, do not affect
            pc_inc    = state[`X3];
        end
        //--------------------------------------
        // KBP : Keyboard Process
        8'b1111_1100 :
        begin
            acc_kbp = state[`X3];
            pc_inc  = state[`X3];
        end
        //--------------------------------------
        // DCL : Designate Control Line
        8'b1111_1101 :
        begin
            dcl_set = state[`X3];
```

```
            pc_inc  = state[`X3];
        end
        //--------------------------------------
        // Others : Same as NOP
        default :
        begin
            pc_inc = state[`X3];
        end
        //--------------------------------------
    endcase
```

Listing 5.22: Instruction control unit

(23) **End of the module**

```
endmodule
```

Listing 5.23: End Module

5.4 Logic of the `MCS4_ROM` Module (4001)

5.4.1 Module `MCS4_ROM` Incorporates 16 `4001` (ROM) Chips

The module `MCS4_ROM` consists of a block comprising 16 chips of `4001` (ROM), totaling 4096 bytes. Chips numbered from 0 to 15 are implemented. Please note that the `MCS4_ROM` module implements sixteen `4001` (ROM) chips as a single unit. These chips are not instantiated and combined individually; they are integrated collectively.

The RTL implementation is defined in `mcs4_rom.v`. Internally, the ROM memory is configured as RAM and initialized via the `initial` statement. Its content can also be modified externally using designated signals. Table 5.2 presents the input/output signals of the `MCS4_ROM` module.

MCS4_ROM (mcs4_rom.v)				
Group	Signal Name	In/Out	Description	Note
System	CLK	Input	Clock Input	
	RES_N	Input	Reset Input	
CPU	SYNC_N	Output	Synchronization Signal Output	
	DATA_I[3:0]	Input	Data Inputs	
	DATA_O[3:0]	Output	Data Outputs	
	DATA_OE	Output	Data Output Enable	
	CM_N	Output	ROM Command Control Output	
	CL_N	Output	Clear Port Outputs	
ROM Ports	PORT_IN_ROM_CHIP7_CHIP0[31:0]	Input	ROM Input Ports, Chip7 - Chip0, each with 4bits	
	PORT_IN_ROM_CHIPF_CHIP8[31:0]	Input	ROM Input Ports, ChipF – Chip8, each with 4bits	
	PORT_OUT_ROM_CHIP7_CHIP0[31:0]	Output	ROM Output Ports, Chip7 - Chip0, each with 4bits	
	PORT_OUT_ROM_CHIPF_CHIP8[31:0]	Output	ROM Output Ports, ChipF – Chip8, each with 4bits	
	ROM_INIT_ENB	Input	Initialization Mode of MCS4 ROM	
	ROM_INIT_ADDR[11:0]	Input	Address for ROM Initialization	
	ROM_INIT_RE	Input	Read Enable for ROM Initialization	
	ROM_INIT_WE	Input	Write Enable for ROM Initialization	
	ROM_INIT_WDATA[7:0]	Input	Write Data for ROM Initialization	
	ROM_INIT_RDATA[7:0]	Output	Read Data for ROM Initialization	

Table 5.2: I/O Signals of `MCS4_ROM`

5.4.2 ROM Initialization

After system power-up, ROM contents are initialized using `$readmemh()` with the file `4001.code` in the `initial` statement, which contains the program code of Busicom's 141-PF calculator using MCS-4 chips.

The file 4001.code can be downloaded from:
http://www.4004.com/assets/busicom-141pf-simulator-w-flowchart-071113.zip
The license of the file is based on:
https://creativecommons.org/licenses/by-nc-sa/2.5/legalcode

For detailed analysis, refer to the disassembled listing:
Busicom-141PF-Calculator_asm_rel-1-0-1.txt
Downloadable from:
http://www.4004.com/2009/Busicom-141PF-Calculator_asm_rel-1-0-1.txt
The license of the file is based on:
https://creativecommons.org/licenses/by-nc-sa/2.5/legalcode

5.4.3 ROM Update Procedure

ROM content can be modified externally by following manners.

(1) Method to Write to ROM

1. Reset the entire MCS-4 system and halt CPU instruction execution.
2. Assert the signal ROM_INIT_ENB.
3. Set target address via ROM_INIT_ADDR[11:0] and write data via ROM_INIT_WDATA[7:0], then assert ROM_INIT_WE for a few clock cycles and deassert.
4. Repeat for other addresses and data sequentially.
5. Deassert ROM_INIT_ENB after writing.
6. Release system reset and resume CPU instruction execution.

(2) Method to Read from ROM

1. Reset the entire MCS-4 system and halt CPU instruction execution.
2. Assert the signal ROM_INIT_ENB.
3. Set the read address using ROM_INIT_ADDR[11:0] and assert ROM_INIT_RE for a few clock cycles.
4. Read data appears on ROM_INIT_RDATA[7:0]; then deassert ROM_INIT_RE.
5. Repeat for additional read operations.
6. Deassert ROM_INIT_ENB after completion.
7. Release system reset and resume CPU instruction execution.

Note: Read operations may immediately follow write operations while ROM_INIT_ENB remains asserted.

5.4.4 Implementation of 4001 I/O Ports

On actual 4001 chips, the directionality of I/O ports is fixed via metal options. In this system, both input and output ports are usable simultaneously. Signals are separated into independent input and output paths. Write operations affect the output port, while read operations retrieve the logic level of the input port. Figure 5.3 illustrates the assumed metal option settings.

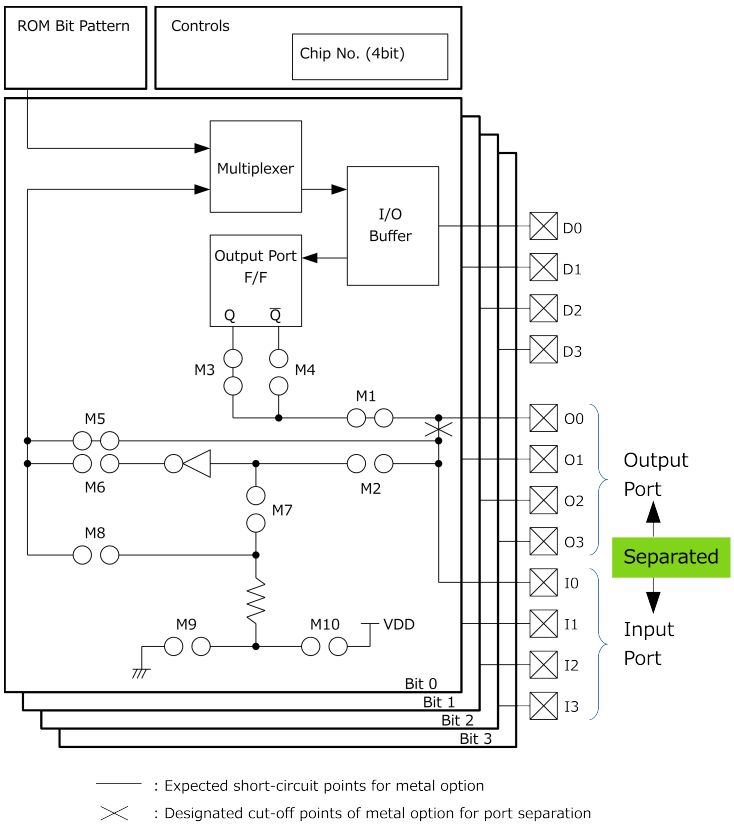


Table 5.3: Assumed Metal Option Settings in MCS4_ROM

The I/O port signals for the MCS4_ROM module are formed by concatenating those from multiple 4001 chips:

- PORT_IN_ROM_CHIP7_CHIP0[31:0] — Input ports from chips 7 through 0
- PORT_IN_ROM_CHIPF_CHIP8[31:0] — Input ports from chips 15 through 8
- PORT_OUT_ROM_CHIP7_CHIP0[31:0] — Output ports from chips 7 through 0
- PORT_OUT_ROM_CHIPF_CHIP8[31:0] — Output ports from chips 15 through 8

5.4.5 RTL Contents of the MCS4_ROM Module

The RTL code mcs4_rom.v includes the ROM memory, input ports, and output ports for all 16 4001 chips.

The module operates in synchronization with the CPU.

During instruction fetch and the second cycle of the FIN instruction, address is captured in states A1–A3 and read data is output during states M1–M2.

When executing the SRC instruction, the signal CM_ROM_N is asserted in state X2, prompting retrieval of the SRC address.

If the upper 4 bits of the instruction code (OPR) fetched in state M2 equal 4'b1110 (indicating data access operations), then CM_ROM_N is asserted again in the subsequent M2 state. The lower 4 bits (OPA) on the data bus are decoded to determine the target and direction (read/write), and the specified read or write operation is executed in state X2.

5.5 Logic of the MCS4_RAM Module (4002)

5.5.1 Module MCS4_RAM Incorporates 32 4002 (RAM) Chips

The module MCS4_RAM consists of a block comprising 32 chips of 4002 (RAM); 8 banks x 4chips; totaling 2560 nibbles (1280 bytes). The RTL implementation is defined in mcs4_ram.v. Table 5.4 presents the input/output signals of the MCS4_RAM module.

MCS4_RAM (mcs4_ram.v)				
Group	Signal Name	In/Out	Description	Note
System	CLK	Input	Clock Input	
	RES_N	Input	Reset Input	
CPU	SYNC_N	Output	Synchronization Signal Output	
	DATA_I[3:0]	Input	Data Inputs	
	DATA_O[3:0]	Output	Data Outputs	
	DATA_OE	Output	Data Output Enable	
	CM_N	Output	RAM Command Control Output	
RAM Ports	PORT_OUT_RAM_BANK1_BANK0	Output	ROM Output Ports, BANK1 – BANK0, Chip3 – Chip0 each with 4bits	
	PORT_OUT_RAM_BANK3_BANK2	Output	ROM Output Ports, BANK3 – BANK2, Chip3 – Chip0 each with 4bits	
	PORT_OUT_RAM_BANK5_BANK4	Output	ROM Output Ports, BANK5 – BANK4, Chip3 – Chip0 each with 4bits	
	PORT_OUT_RAM_BANK7_BANK6	Output	ROM Output Ports, BANK7 – BANK6, Chip3 – Chip0 each with 4bits	

Table 5.4: I/O Signals of MCS4_RAM

5.5.2 Expansion of RAM Banks

The 4004 CPU outputs four signals: CM_RAM_N[3:0]. If these are used as one-hot signals directly without external circuitry, the system can support up to four RAM banks. However, by encoding CM_RAM_N[3:0] and decoding them inside the module, the number of RAM banks can be expanded to eight. In the MCS4_RAM module, CM_RAM_N[3:0] from the CPU is decoded to generate eight bank-select signals as shown in Table 5.5.

5.5.3 Implementation of 4002 Output Ports

The output port signals for the MCS4_RAM module are formed by concatenating those from multiple 4002 chips:

- PORT_OUT_RAM_BANK1_BANK0[31:0] — Output ports BANK1/Chip3-Chip0, and BANK0/Chip3-Chip0
- PORT_OUT_RAM_BANK3_BANK2[31:0] — Output ports BANK3/Chip3-Chip0, and BANK2/Chip3-Chip0
- PORT_OUT_RAM_BANK5_BANK4[31:0] — Output ports BANK5/Chip3-Chip0, and BANK4/Chip3-Chip0
- PORT_OUT_RAM_BANK7_BANK6[31:0] — Output ports BANK7/Chip3-Chip0, and BANK6/Chip3-Chip0

cm_ram_n_encoded				cm_ram_n								
3	2	1	0	7	6	5	4	3	2	1	0	
0	0	0	0	1	1	1	1	1	1	1	1	n/a
0	0	0	1	0	1	1	1	1	1	1	1	Bank 7
0	0	1	0	1	1	1	1	1	1	1	1	n/a
0	0	1	1	1	0	1	1	1	1	1	1	Bank 6
0	1	0	0	1	1	1	1	1	1	1	1	n/a
0	1	0	1	1	1	0	1	1	1	1	1	Bank 5
0	1	1	0	1	1	1	1	1	1	1	1	n/a
0	1	1	1	1	1	1	0	1	1	1	1	Bank 4
1	0	0	0	1	1	1	1	1	1	1	1	n/a
1	0	0	1	1	1	1	1	0	1	1	1	Bank 3
1	0	1	0	1	1	1	1	1	1	1	1	n/a
1	0	1	1	1	1	1	1	1	0	1	1	Bank 2
1	1	0	0	1	1	1	1	1	1	1	1	n/a
1	1	0	1	1	1	1	1	1	1	0	1	Bank 1
1	1	1	0	1	1	1	1	1	1	1	0	Bank 0
1	1	1	1	1	1	1	1	1	1	1	1	n/a

Table 5.5: Decode of CM_RAM_N[3:0]

5.5.4 RTL Contents of the `MCS4_RAM` Module

The RTL code in `mcs4_ram.v` describes the memory and output ports for 8 banks × 4 chips of `4002` RAM.

The module operates in synchronization with the CPU.

When the CPU executes the `SRC` instruction, `CM_RAM_N[3:0]` is asserted during state X2, which triggers the loading of the SRC address.

During state M2, the CPU fetches the upper 4 bits (`OPR`) of the instruction code. If `OPR` matches the data-access opcode (`4'b1110`), then `CM_RAM_N[3:0]` is asserted in the following M2 cycle. The lower 4 bits (`OPA`) on the data bus are decoded to determine the access target and operation direction (read/write), and the corresponding read or write operation is performed during state X2.

5.5.5 Module Structure of `MCS4_RAM`

Within the `MCS4_RAM` module, a total of 32 instances of the `MCS4_RAM_CHIP` module—each representing one `4002` RAM chip—are instantiated. The `MCS4_RAM_CHIP` module itself is also defined within the RTL code `mcs4_ram.v`. Each RAM bank consists of four `MCS4_RAM_CHIP` instances, and these must be distinguished by their chip number.

To identify each chip, two input signals—`P0` and `P1`—are provided to the `MCS4_RAM_CHIP` module:

- `P0`: A chip select signal found in actual `4002` devices.
- `P1`: A signal introduced to distinguish between chip types (e.g., `4002-1` and `4002-2`) at the input level.

When instantiating `MCS4_RAM_CHIP` modules within `MCS4_RAM`, the logic level of `P0` and `P1` is explicitly assigned.

5.5.6 Reset Behavior and RAM Initialization

Real `4002` chips feature an internal mechanism to initialize RAM contents to zero during reset. In actual devices, asserting the reset signal for 32 instruction cycles (equivalent to

256 system clock cycles) triggers an internal counter to clear the RAM contents.

This behavior is replicated in the current system. The MCS4_RAM_CHIP module incorporates similar logic such that asserting the reset signal RES_N for 64 or more system clock cycles initializes RAM contents to zero.

Unlike modern RAM devices—which generally lack hardware-level zero-clear functionality—the 4002 RAM's automatic initialization eliminates the need for memory-clear routines in software, contributing to reduced program size.

5.6 Logic of the MCS4_SHIFTER Module (4003)

5.6.1 Simple Shift Register 4003 (Shift Register) Chip

The MCS4_SHIFTER module implements the functionality of the 4003 chip, which serves as a serial-in/parallel-out shift register. Its RTL description is defined in the file mcs4_shifter.v. Table 5.6 presents the input/output signals of the MCS4_SHIFTER module.

SHIFT REGISTER (mcs4_shifter.v)				
Group	Signal Name	In/Out	Description	Note
System	CLK	Input	Clock Input	
	RES_N	Input	Reset Input	
	SCK	Input	Shift Clock	
	SDI	Input	Shift Data Input	
	SDO	Output	Shift Data Output	
	OE	Input	Output Enable	
	Q[9:0]	Output	Parallel Output	

Table 5.6: I/O Signals of MCS4_SHIFTER

5.6.2 RTL Contents of the MCS4_SHIFTER Module

The RTL module mcs4_shifter.v implements a 10-bit shift register. Data is shifted into the register on the rising edge of the serial clock signal SCK from the serial data input SDI, and shifted out via the serial data output SDO.

The module features an output enable signal OE, which determines the behavior of the parallel output Q[9:0]:

- If OE is High, the contents of the 10-bit shift register are presented at Q[9:0].
- If OE is Low, the output Q[9:0] is forced to zero.

Chapter 6

MCS-4 System and Busicom 141-PF

In the previous chapter, I explained the logical design of the MCS-4 chipset, comprising the `4004` (CPU), `4001` (ROM), `4002` (RAM), and `4003` (shift register). In this chapter, we construct a complete MCS-4 system by integrating these chips, and further reproduce the `141-PF` calculator developed by Busicom Corporation, implementing it on an FPGA.

The user interfaces such as the calculator's keyboard and printer, shown in Figure 6.1, are realized using a modern microcontroller based on the RISC-V architecture, combined with a UART terminal or a Touch LCD panel. Our goal is to thoroughly appreciate the functionality and internal architecture of the historically significant `141-PF` calculator, which remains practically usable to this day and is exhibited in museums.

The RTL descriptions for the system are stored in the directory `RTL/`. The logic simulation stuffs can be found in the directory `SIM_iverilog/` and `SIM_questa/`. The FPGA implementation are stored in the directory `FPGA/`.

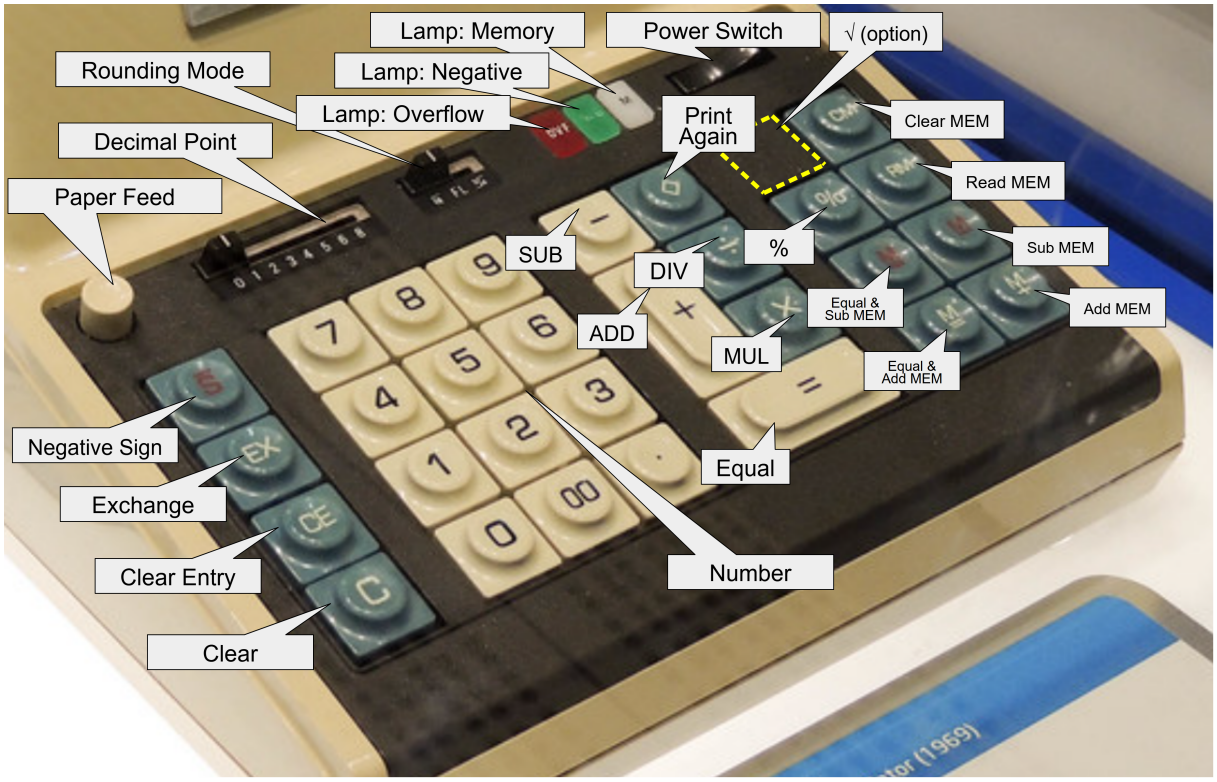


Figure 6.1: Key Board and Switches on Calculator 141-PF
https://www.flickriver.com/photos/gorekun/34947650073/

6.1 Overview of the designed MCS-4 System

This MCS-4 system is designed to emulate the configuration of the `141-PF` calculator, with the ultimate goal of implementation on an FPGA platform. The original `141-PF` calculator comprises five `4001` (ROM) chips—one of which is an optional chip for square root calculation routines—two `4002` (RAM) chips, three `4003` (shift register) chips, and a single `4004` (CPU) chip.

However, to accommodate pi calculation, which is described in another chapter, the memory architecture of the designed MCS-4 system has been expanded to its maximum configuration. Specifically, the `4004` (CPU) is connected to sixteen `4001` (ROM) chips and thirty-two `4002` (RAM) chips, organized into eight banks of four chips each.

A block diagram of the designed MCS-4 system is shown in Figure 6.2, and a specification summary is presented in Table 6.1.

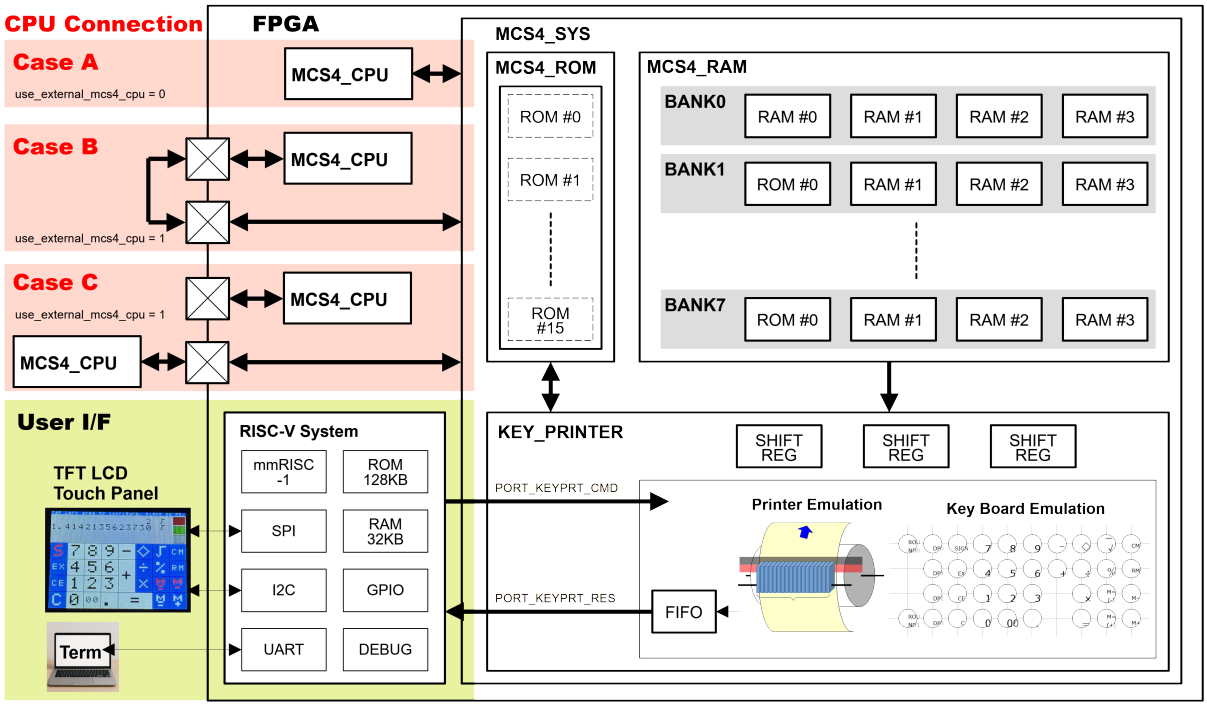


Figure 6.2: Block Diagram of MCS-4 System

Peripheral circuitry of the MCS-4 system aligns with the hardware configuration of the `141-PF` calculator, utilizing three `4003` (shift register) chips. The behavior of the calculator's keyboard and printer interfaces is emulated using a modern RISC-V microcontroller system. The RISC-V subsystem adopts the existing `mmRISC-1` architecture, with peripheral functions retained and only the GPIO set extended[4].

Communication between the RISC-V subsystem and the MCS-4 system is handled via GPIO signals: command signals are transmitted through `PORT_KEYPRT_CMD[31:0]`, and responses are received through `PORT_KEYPRT_RES[31:0]`. The control logic for these command/response operations is implemented in the `KEY_PRINTER` module, which also integrates three `4003` (shift register) chips.

User interaction via the RISC-V subsystem is primarily achieved through UART terminal operations. Optionally, a graphical interface is also available via a color LCD panel with touch functionality.

The `4004` (CPU) connects internally to the MCS-4 memory system (ROM and RAM) within the FPGA. Additionally, to accommodate the use of standalone `4004` chips, external connectivity is supported. Since the real `4004` chip is fabricated in a 10µm PMOS

Category				Description
Hardware Logic in Altera MAX 10 FPGA (DE10-Lite Board)	MCS4_CPU	MCS4_CPU	4004 (CPU)	- Selectable from Internal or External - Operates at the same frequency as the actual hardware (750 kHz).
	MCS4_SYS	MCS4_ROM	4001 (ROM) x 16	- Implements support for 16 chips. - ROM capacity: 4,096 bytes - 64 input ports - 64 output ports (Input and output ports are separated and are independent.)
		MCS4_RAM	4002 (RAM) x 32	- Implements 8 banks, supporting up to 32 chips. - RAM capacity: 2,560 nibbles (1,280 bytes) - 128 output ports
		KEY_PRINTER	4003 (Shift Register) x 3	- 4003 x 1 unit for keyboard scanning - 4003 x 2 units for printer output (cascade connection) - Control logic for Key Board and Printer
	RISCV_TOP (MCU for User I/F)	RISC-V CPU	mmRISC-1	- RV32IMAC @ 20MHz - 4-wire JTAG Debug Interface
		Memory	ROM	- 128KB
			RAM	- 32KB
			SDRAM	- 64KB (32MW x 16bits) on DE10-Lite board
		Peripherals	GPIO	- 32bit I/O Ports x 6 (expanded from original mmRISC-1)
			UART	- x1
			SPI	- Master x 1 (for control of Touch TFT Shield)
			I2C	- x2 (for control of Touch TFT Shield and 3-axis accelerometer sensor on DE10-Lite board)
User Interface				- PC Terminal via UART controller by RISC-V (115200bps, 8N1)
				- 2.8inch TFT Touch Shield for Arduino controlled by RISC-V Adafruit Capacitive Touch (Product ID: 1947) or Adafruit Resistive Touch (Product ID: 1651)
Software		MCS4 Program	141PF	- Program for Calculator of 141-PF
			PI4004	- Program for Calculating Pi to 500 Digits
		RISC-V Program	MCS4_SYSTEM	- User Interface of MCS-4 System
Note				While the MCS4_ROM is writable from the RISC-V subsystem, this implementation does not support the native capability of the MCS4_CPU to perform self-modification when part of the program memory—typically composed of 4001 devices—is substituted with RAM.

Table 6.1: MCS-4 System Specification

process and is electrically incompatible with the FPGA (due to voltage and DC level mismatches), external level-shifting circuitry might be required.

Nevertheless, recent advances in open-source PDKs and EDA tools have enabled free LSI designs, and users can fabricate a custom 4004 chip via low-cost foundry shuttle services. These chips can be connected to the FPGA and used for practical experimentation.

To support these scenarios, the FPGA design accommodates the following three configurations for CPU connection, selectable via onboard switches:

- **Case A**: 4004 (CPU) connected internally to the FPGA's MCS-4 system (ROM and RAM)
- **Case B**: 4004 logic implemented in the FPGA, interfaced to the internal MCS-4 system via external pins
- **Case C**: Standalone 4004 chip connected via external pins to the FPGA's internal MCS-4 system (ROM and RAM)

Among these, Case B and Case C share an identical logical structure, differing only in their external connections. Case B is provided as a debugging option for experimental use of Case C.

6.2 FPGA Board and Optional Components Used

The MCS-4 system designed in this project is implemented on the DE10-Lite FPGA board from Teasic, as shown in Figure 6.3. The DE10-Lite is a compact board equipped with an Altera MAX 10 FPGA. The user interface for the calculator application within this system is controlled by firmware running on a RISC-V subsystem embedded in the

FPGA. This interface communicates with a terminal application on the PC via UART.

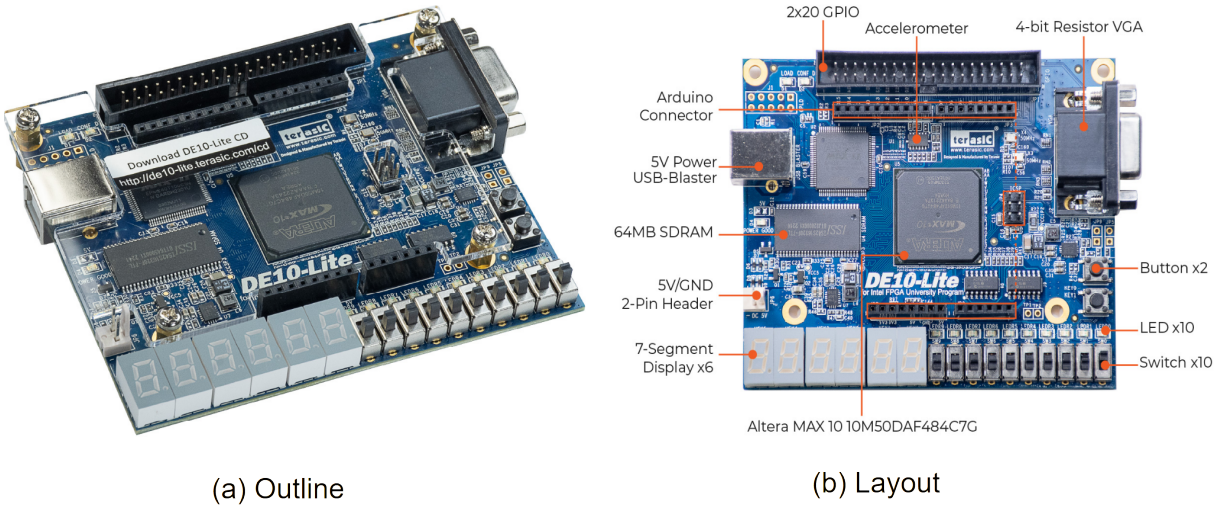


Figure 6.3: Terasic DE10-Lite Board

https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=1021

For users who prefer to operate the calculator via a graphical user interface (GUI), the system can be expanded by mounting an Arduino-compatible LCD panel with a capacitive touch screen onto the DE10-Lite board. The touch-enabled LCD panel used is the Adafruit 2.8" TFT Touch Shield for Arduino with Capacitive Touch (Product ID: 1947), as shown in Figure 6.4.

The author also possesses a resistive touch panel variant of the LCD, shown in Figure 6.4. This panel, the Adafruit 2.8" TFT Touch Shield for Arduino with Resistive Touch (Product ID: 1651), is also compatible with the system.

The firmware on the RISC-V subsystem automatically detects the presence of an LCD panel on the DE10-Lite board and the type of touch panel installed, switching the user interface from a UART-based terminal application to the touch-enabled LCD panel as appropriate.

However, the resistive touch panel version shown in Figure 6.4 has undergone a revision, and its integrated touch controller IC has been changed. Consequently, the latest model is not compatible with the firmware developed by the author.

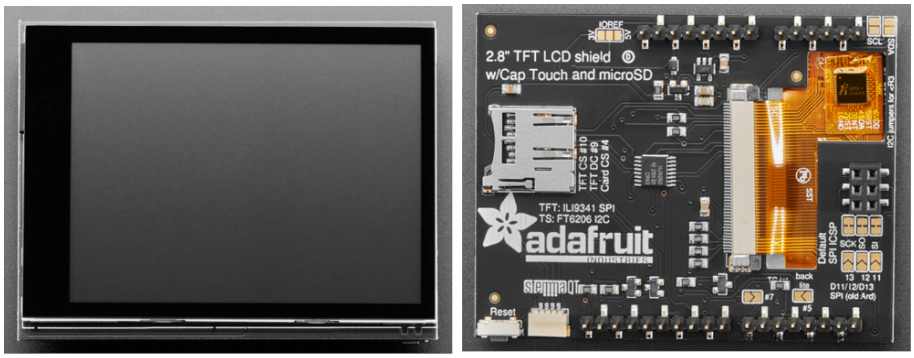
6.3 Overview of the FPGA Top-Level Logic

The structure of the FPGA top-level logic is illustrated in Figure 6.2, and will be further explained in detail below. The top-level RTL description of the FPGA is located in `RTL/FPGA_TOP/fpga_top.v`, with the RTL description related to the MCS-4 system in `RTL/MCS4`, and the RISC-V subsystem in `RTL/RISCV`.

6.3.1 fpga_top.v: FPGA Top-Level RTL Description

Regarding the clock configuration, the 50 MHz oscillator on the DE10-Lite board provides input to the FPGA, and the internal PLL generates a 20 MHz clock for the RISC-V subsystem and a 750 kHz clock for the MCS-4 system.

For the reset mechanism, a power-on reset generation circuit has been implemented within the FPGA. This consists of a counter that produces a reset signal with a fixed

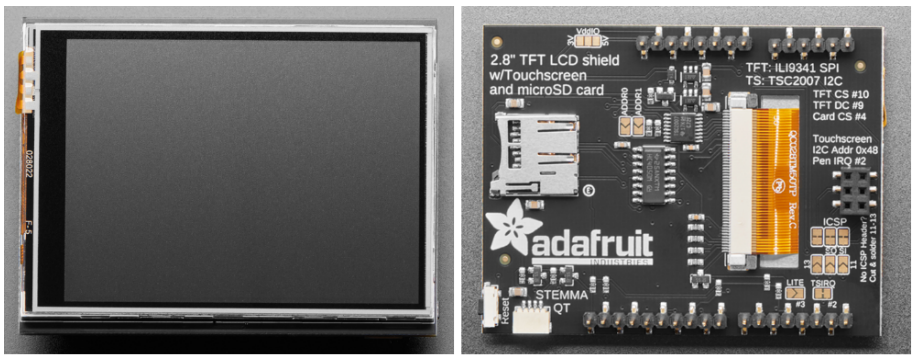


(a) Front Side (b) Back Side

Figure 6.4: Adafruit 2.8" TFT Touch Shield for Arduino with Capacitive Touch (Product ID: 1947)

This is the recommended one. LCD Controller=ILI9341 (SPI I/F), Capacitive Touch Controller=FT6206 (I2C I/F).

https://www.adafruit.com/product/1947



(a) Front Side (b) Back Side

Figure 6.5: Adafruit 2.8" TFT Touch Shield for Arduino with Resistive Touch Screen (Product ID: 1651)

LCD Controller=ILI9341 (SPI I/F), Resistive Touch Controller=STMPE610 (SPI I/F).

https://www.adafruit.com/product/1651

width. By setting synthesis constraints such that the counter is cleared to zero upon initial power-on, the reset behavior is reliably established. The final reset signal distributed inside the FPGA is synchronized by clock and constructed from an OR combination of the power-on reset signal, an external reset input from a switch, and the PLL lock signal.

6.3.2 Modules Related to MCS-4

Within the top-level hierarchy of the FPGA, the `MCS4_CPU` and `MCS4_SYS` modules are instantiated. The `MCS4_CPU` module represents the 4004 CPU, while the `MCS4_SYS` module includes the `MCS4_ROM`, `MCS4_RAM`, and `KEY_PRINTER` modules.

As described in the previous section, three configurations—Case A, Case B, and Case C—are available for connecting the `MCS4_CPU`, and the selection logic is also implemented within the top-level hierarchy. Specifically, the external SW9 switch input is connected via `GPIO2[9]` and captured as the signal `use_external_mcs4_cpu`. When this signal level is Low, Case A is selected; when High, either Case B or Case C is enabled. This switching mechanism is the reason why `MCS4_CPU` and `MCS4_SYS` modules are separated.

Table 6.2 lists the input/output signals of the `MCS4_SYS` module, while Table 6.3 summarizes the input/output signals of the `KEY_PRINTER` module.

MCS4_SYS (mcs4_sys.v)				
Group	Signal Name	In/Out	Description	Note
System	CLK	Input	Clock Input	
	RES_N	Input	Reset Input	
CPU	SYNC_N	Output	Synchronization Signal Output	
	DATA_I[3:0]	Input	Data Inputs	
	DATA_O[3:0]	Output	Data Outputs	
	DATA_OE	Output	Data Output Enable	
	CM_ROM_N	Output	ROM Command Control Output	
	CM_RAM_N[3:0]	Output	RAM Command Control Outputs	
	TEST	Input	Conditional Branch Test Condition Input	
Host I/F for Calculator	PORT_KEYPRT_CMD[31:0]	Input	Calculator Control Command from Host CPU	
	PORT_KEYPRT_RES[31:0]	Output	Calculator Control Response to Host CPU	
Host I/F for ROM Initialization	ROM_INIT_ENB	Input	Initialization Mode of MCS4 ROM	
	ROM_INIT_ADDR[11:0]	Input	Address for ROM Initialization	
	ROM_INIT_RE	Input	Read Enable for ROM Initialization	
	ROM_INIT_WE	Input	Write Enable for ROM Initialization	
	ROM_INIT_WDATA[7:0]	Input	Write Data for ROM Initialization	
	ROM_INIT_RDATA[7:0]	Output	Read Data for ROM Initialization	

Table 6.2: I/O Signals of `MCS4_SYS`

KEY_PRINTER (key_printer.v)				
Group	Signal Name	In/Out	Description	Note
System	CLK	Input	Clock Input	
	RES_N	Input	Reset Input	
Common Control	ENABLE	Input	Enable Key and Printer	
	TEST	Output	TEST Signal of CPU (Timing of each row position of printer drum)	
ROM Ports	PORT_IN_ROM_CHIP7_CHIP0[31:0]	Output	ROM Input Ports, Chip7 - Chip0, each with 4bits	
	PORT_IN_ROM_CHIPF_CHIP8[31:0]	Output	ROM Input Ports, ChipF – Chip8, each with 4bits	
	PORT_OUT_ROM_CHIP7_CHIP0[31:0]	Input	ROM Output Ports, Chip7 - Chip0, each with 4bits	
	PORT_OUT_ROM_CHIPF_CHIP8[31:0]	Input	ROM Output Ports, ChipF – Chip8, each with 4bits	
RAM Ports	PORT_OUT_RAM_BANK1_BANK0	Input	ROM Output Ports, BANK1 – BANK0, Chip3 – Chip0 each with 4bits	
	PORT_OUT_RAM_BANK3_BANK2	Input	ROM Output Ports, BANK3 – BANK2, Chip3 – Chip0 each with 4bits	
	PORT_OUT_RAM_BANK5_BANK4	Input	ROM Output Ports, BANK5 – BANK4, Chip3 – Chip0 each with 4bits	
	PORT_OUT_RAM_BANK7_BANK6	Input	ROM Output Ports, BANK7 – BANK6, Chip3 – Chip0 each with 4bits	
Host I/F for Calculator	PORT_KEYPRT_CMD[31:0]	Input	Calculator Control Command from Host CPU	
	PORT_KEYPRT_RES[31:0]	Output	Calculator Control Response to Host CPU	

Table 6.3: I/O Signals of `KEY_PRINTER`

6.3.3 RISC-V Subsystem

The RISC-V subsystem utilizes the mmRISC-1[4] design as-is, including its peripheral functions. However, the original configuration of three 32-bit GPIO ports has been ex-

panded to six ports in this system.

6.4 Logic Description of MCS-4 Related Modules

The `MCS4_CPU` module is a faithful implementation of the Intel 4004 CPU, which is described in detail in a separate chapter. The `MCS4_SYS` module comprises the `MCS4_ROM`, `MCS4_RAM`, and `KEY_PRINTER` modules, with each ROM and RAM module also explained in a separate section. This chapter focuses on the `KEY_PRINTER` module.

6.4.1 KEY_PRINTER Module

The `KEY_PRINTER` module integrates three 4003 shift registers, forming a circuit that emulates the keyboard, printer, and status indicator lamps of the calculator. It connects to the input/output ports of the `MCS4_ROM` module and to the output ports of the `MCS4_RAM` module, thereby simulating the behavior of the original 141-PF calculator.

This module also interfaces with a RISC-V subsystem responsible for user interaction. It exchanges command and response signals through `PORT_KEYPRT_CMD[31:0]` and `PORT_KEYPRT_RES[31:0]`, respectively. Operating at 750 kHz, the `KEY_PRINTER` faithfully reproduces the speed and performance of the original 141-PF calculator hardware.

6.4.2 I/O Circuitry of Busicom 141-PF

The `KEY_PRINTER` module emulates the I/O circuits located outside the ROM (4001) and RAM (4002) ports of the 141-PF calculator. This emulation allows the original Busicom binary code, available online, to execute correctly. Figure 6.6 illustrates the external I/O circuitry assumed by the 141-PF program.

To expand the number of output ports, this circuit uses multiple 4003 shift registers. Table 6.4 lists the connection targets for the ROM/RAM I/O ports, while Table 6.5 shows the configuration of 4003 shift registers within the calculator ' s I/O circuitry.

Device	Pin	Direction	Signal Descriptions
CPU	TEST	IN	Printer drum sector signal DRUM_SECTOR (period = 28ms / frequency = 35.7Hz)
4001 ROM#0	IO0	OUT	Shift clock for keyboard column signals KBC9–KBC0 input (i4003#0)
	IO1	OUT	Shared shift input data for keyboard and printer (i4003#0, i4003#1)
	IO2	OUT	Shift clock for printer column data input (i4003#1, i4003#2)
	IO3	OUT	Unused
4001 ROM#1	IO0	IN	Keyboard row signal KBR0 input
	IO1	IN	Keyboard row signal KBR1 input
	IO2	IN	Keyboard row signal KBR2 input
	IO3	IN	Keyboard row signal KBR3 input
4001 ROM#2	IO0	IN	Printer drum index signal DRUM_INDEX (period = 13×28 = 364ms / frequency = 2.74Hz)
	IO1	IN	Unused
	IO2	IN	Unused
	IO3	IN	Printer paper feed button input (1 when pressed)
4002 RAM#0	O0	OUT	Printer color selection PRINTING_COLOR (0: black, 1: red)
	O1	OUT	Printer hammer firing command FIRE_HUMMERS
	O2	OUT	Unused
	O3	OUT	Printer paper feed command ADVANCE_PAPER
4002 RAM#1	O0	OUT	Status lamp (Memory) ON (1 = ON)
	O1	OUT	Status lamp (Overflow) ON (1 = ON)
	O2	OUT	Status lamp (Negative) ON (1 = ON)
	O3	OUT	Unused

Table 6.4: Connections of ROM/RAM I/O Ports in the Calculator I/O Circuit

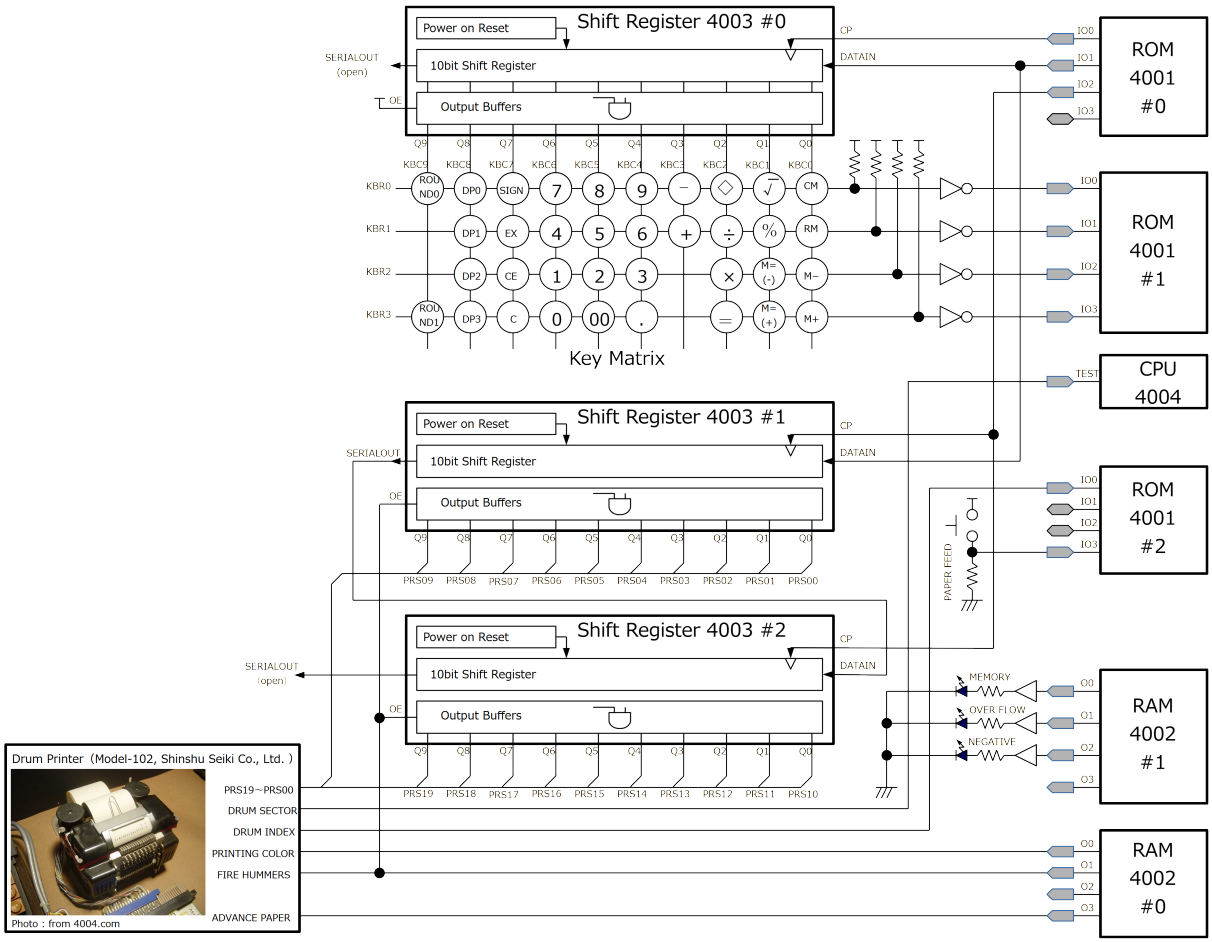


Figure 6.6: I/O Circuit of the Busicom 141-PF Calculator via MCS-4 System

Usage	Device	Pin	Direction	Signal Connections
Key Board Column (KBC)	i4003#0	OE	IN	VDD
		CP	IN	ROM#0 IO0 (OUT)
		DATA IN	IN	ROM#0 IO1 (OUT), shared between keyboard and printer
		SERIAL OUT	OUT	Open
		Q9∼Q0	OUT	Keyboard column signals KBC9–KBC0
Printer Hammer Control Lower Side	i4003#1	OE	IN	RAM#0 O1 (Printer hammer firing command)
		CP	IN	ROM#0 IO2 (OUT)
		DATA IN	IN	ROM#0 IO1 (OUT), shared between keyboard and printer
		SERIAL OUT	OUT	Connected to DATA IN of i4003#2 for cascading shift registers
		Q9∼Q0	OUT	Printer hammer control PRS9–PRS0
Printer Hammer Control Higher Side	i4003#2	OE	IN	RAM#0 O1 (Printer hammer firing command)
		CP	IN	ROM#0 IO2 (OUT)
		DATA IN	IN	Connected to SERIAL OUT of i4003#1 for cascading shift registers
		SERIAL OUT	OUT	Open
		Q9∼Q0	OUT	Printer hammer control PRS19–PRS10

Table 6.5: Connection Method of 4003 Registers Within the Calculator I/O Circuit

6.4.3 Keyboard Matrix of the Calculator

The keyboard of the Busicom 141-PF calculator, along with the input switches (rounding mode selector and decimal digit selector), is read via a matrix circuit. When a key or switch is activated, its intersection connects a column signal (KBC0 to KBC9) with a row signal (KBR0 to KBR3).

As shown at the top of Figure 6.6, the 4003 #0 shift register generates the KBC0–KBC9 column signals. The clock and shift input data for 4003 #0 are supplied from IO0 and IO1 of 4001 (ROM) #0, respectively.

The four row signals KBR0–KBR3 are read via pull-up resistors and inverters through the input ports of 4001 (ROM) #1. When no keys are pressed, all four signals read LOW by 4001 (ROM) #1. The column signals KBC0–KBC9 are initially set to HIGH, and each is momentarily pulled LOW in sequence at high speed (key scanning operation). If a key or switch is ON when a column signal is LOW, the corresponding row signal reads HIGH, allowing the program to detect the key or switch state.

Table 6.6 illustrates the key and switch mappings of the calculator's keyboard matrix.

Table 6.6(a) shows the full layout of the keyboard matrix. The range of columns KBC0 to KBC7 corresponds to the main keyboard. Pressing a key in this region results in its conversion to a Key Code, as indicated in parentheses, within the 141-PF program.

Column KBC8 is for switches that specify the number of decimal digits. The switch states and their meanings interpreted by the 141-PF program are listed in Table 6.6(b).

Column KBC9 is for switches that specify the rounding mode. The switch states and corresponding interpretations by the 141-PF program are presented in Table 6.6(c).

		Key Board Column Selection									
		KBC9	KBC8	KBC7	KBC6	KBC5	KBC4	KBC3	KBC2	KBC1	KBC0
Key Board Row Signals	KBR0	ROUND0	DP0	SIGN (0x9D)	7 (0x99)	8 (0x95)	9 (0x91)	− (0x8D)	◇ (0x89)	√ (0x85)	CM (0x81)
	KBR1		DP1	EX (0x9E)	4 (0x9A)	5 (0x96)	6 (0x92)	+ (0x8E)	÷ (0x8A)	% (0x86)	RM (0x82)
	KBR2		DP2	CE (0x9F)	1 (0x9B)	2 (0x97)	3 (0x93)	◇2 (0x8F)	× (0x8B)	M=(−) (0x87)	M− (0x83)
	KBR3	ROUND1	DP3	C (0xA0)	0 (0x9C)	00 (0x98)	. (0x94)	000 (0x90)	= (0x8c)	M=(+) (0x88)	M+ (0x84)

: Unimplemented Key ( ) : Key Code

(a) Key Board matrix

DP3	DP2	DP1	DP0	Number of Decimal Places
OFF	OFF	OFF	OFF	0
OFF	OFF	OFF	ON	1
OFF	OFF	ON	OFF	2
OFF	OFF	ON	ON	3
OFF	ON	OFF	OFF	4
OFF	ON	OFF	ON	5
OFF	ON	ON	OFF	6
OFF	ON	ON	ON	n/a
ON	OFF	OFF	OFF	8
Others				n/a

(b) Decimal Point Switch

ROUND1	ROUND0	Rounding Mode
OFF	OFF	FLOATING
OFF	ON	ROUND
ON	OFF	TRUNCATE
ON	ON	n/a

(c) Rounding Switch

Table 6.6: Keyboard Matrix States and Meanings of the Calculator

6.4.4 Printing Mechanism of the Calculator

The calculation results of the 141-PF calculator are printed out via a built-in printer. As previously mentioned, the adopted printer model is the Model 102 from Shinshu Seiki Co. (now Seiko Epson). The internal printing mechanism is illustrated in Figure 6.7.

A rotating drum is embedded with convex typesetting characters. Above the drum are layered, in order: paper, ink ribbon, and a hammer. When a desired character on the drum rotates into position beneath the paper, the hammer strikes in precise timing, transferring the ink to the paper via impact. This mechanism is synchronized to ensure accurate printing at each target position.

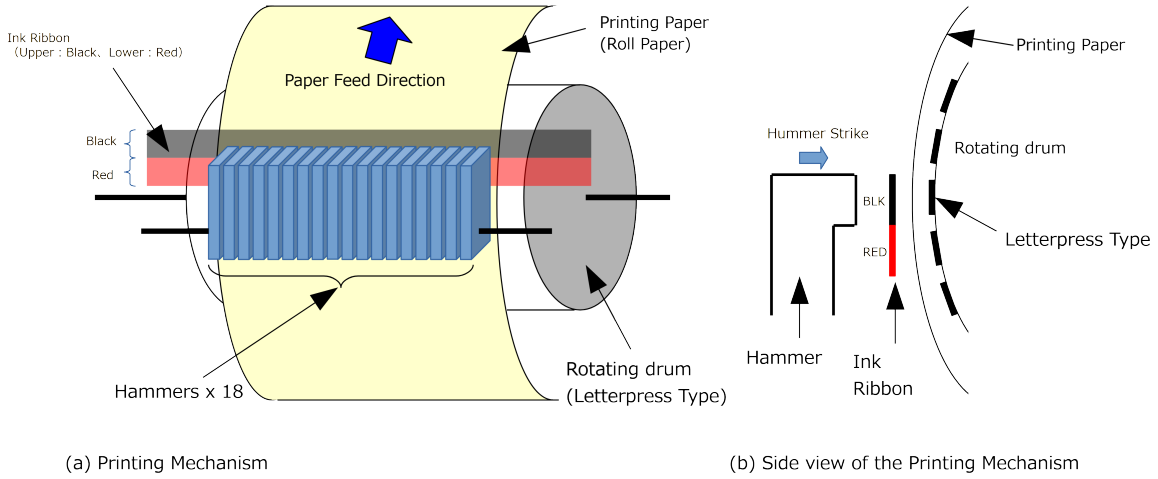


Figure 6.7: Printing Mechanism of the Calculator

6.4.5 Rotating Drum and Status Signals of the Printer

The rotating drum of the Busicom 141-PF calculator's printer contains embedded typesetting characters, as illustrated in Table 6.7. Conceptually, the drum forms a cylindrical shape where the top and bottom edges of Table 6.7 connect in a loop.

Horizontally, the drum holds 18 columns (digits), including control characters. However, no type is embedded in Column 16, which always results in a blank space on the printed paper. Vertically, the drum is divided into 13 sectors (rows), each containing a full set of characters.

As the drum rotates, it outputs a status signal `DRUM_SECTOR` to indicate that a sector's characters have reached the print position, where the hammer can strike. The 4004 CPU detects this signal via its `TEST` pin to activate the hammer in precise synchronization.

To determine the rotational position of the drum, a separate signal `DRUM_INDEX` is generated when Sector 0's characters align with the print position. This signal is read via IO0 on 4001 (ROM) #2.

The `DRUM_SECTOR` signal is output at approximately 28 ms intervals (frequency = 35.7 Hz), while the `DRUM_INDEX` signal occurs once every full rotation of 13 sectors, i.e., every 364 ms (frequency = 2.74 Hz).

6.4.6 Hammer Control of the Printer

The 141-PF calculator program reads the rotational position of the drum and sequentially issues hammer strike commands for the characters that appear earliest on the drum. There are 18 hammers corresponding to the 18 character columns on the drum.

	Column 1	Column 2	Column 3	Column 4	Column 5	Column 6	Column 7	Column 8	Column 9	Column 10	Column 11	Column 12	Column 13	Column 14	Column 15	Column 16	Column 17	Column 18	Sector Signal to TEST Pin	Index Signal to ROM#2 IO0
	PRS3	PRS4	PRS5	PRS6	PRS7	PRS8	PRS9	PRS10	PRS11	PRS12	PRS13	PRS14	PRS15	PRS16	PRS17		PRS0	PRS1		
Sector 0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		◇	#	✔	✔
Sector 1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1		+	*	✔	
Sector 2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2		−	I	✔	
Sector 3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3		×	II	✔	
Sector 4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4		/	III	✔	
Sector 5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5		M+	M+	✔	
Sector 6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6		M−	M−	✔	
Sector 7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7		.	T	✔	
Sector 8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8		=	K	✔	
Sector 9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9		√	E	✔	
Sector 10	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		%	Ex	✔	
Sector 11	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		C	C	✔	
Sector 12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		R	M	✔	

Table 6.7: Rotating Drum of the Printer

The selection signals for these hammers are labeled `PRS00` to `PRS19`, as shown in Figure 6.6, and are output from 4003 #1 and 4003 #2. These shift registers are connected in a cascade configuration. The clock signal is supplied via IO2 of 4001 (ROM) #0, and shift input data is provided via IO1 of the same ROM. Although the shift input data line is shared with the 4003 #0 used for the keyboard matrix, the shift clocks are separate, allowing independent operation without conflict.

The correspondence between `PRS00`–`PRS19` and actual drum print positions is listed in Table 6.7. Note that some signals are unused.

To initiate the hammer strike, the signal `FIRE_HAMMERS` is asserted. This signal is output from O1 of 4002 (RAM) #0. As a result, the hammers selected by `PRS00`–`PRS19` are activated simultaneously at the timing of `FIRE_HAMMERS`. Multiple hammers may be fired at the same time.

6.4.7 Printing Sequence of the Calculator's Printer

An example of the printing sequence is shown in Table 6.8. Here, we consider the case where the string “1.4142135623730 SQ” is printed. Since the 141-PF program continuously monitors the rotational position of the drum, printing can begin from the sector that is approaching the print position first. In this example, Sector 0 is assumed to be next in line for printing.

First, the hammer selection signals (`PRS00`–`PRS19`) corresponding to the characters in Sector 0 are configured, and the hammer is triggered. In this case, only Column 15 (character “0”) is activated, since the digit “0” appears only once in the numeric result.

As the drum rotates, Sector 1 approaches the print position. Again, hammer selection signals are configured for the printable characters, and the hammer is triggered. Here, the digit “1” appears three times in the numeric result, so the corresponding hammer signals for those columns are asserted.

This process continues up to Sector 12. As a result, the printed output is not produced sequentially from left to right. Instead, characters across various columns are struck sector-by-sector in a scattered order, and the full result gradually appears once all necessary sectors have passed.

Time	Sector	PRS[19:0]		Columns to Hammer																	Sector Signal to TEST Pin	Index Signal to ROM#2 IO0
		Hex	Binary	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	
	0	20'h20000	20'b0010_0000_0000_0000_0000															0				✔	✔
	1	20'h00248	20'b0000_0000_0010_0100_1000	1			1			1												✔	
	2	20'h02100	20'b0000_0010_0001_0000_0000						2					2								✔	
	3	20'h14400	20'b0001_0100_0100_0000_0000								3					3		3				✔	
	4	20'h000A0	20'b0000_0000_0000_1010_0000			4		4														✔	
	5	20'h00800	20'b0000_0000_1000_0000_0000									5										✔	
	6	20'h01000	20'b0000_0001_0000_0000_0000										6									✔	
	7	20'h08000	20'b0000_1000_0000_0000_0000													7						✔	
	8	20'h00000	20'b0000_0000_0000_0000_0000																			✔	
	9	20'h00001	20'b0000_0000_0000_0000_0001																	√		✔	
	10	20'h00010	20'b0000_0000_0000_0001_0000		.																	✔	
	11	20'h00000	20'b0000_0000_0000_0000_0000																			✔	
	12	20'h00000	20'b0000_0000_0000_0000_0000																			✔	
			Final Result of Printing	1	.	4	1	4	2	1	3	5	6	2	3	7	3	0		√			

Table 6.8: Printing Sequence Example of the Calculator's Printer

6.4.8 Ink Ribbon and Paper Feed Control

Before a character is struck by the hammer, the printing color is specified via the `PRINTING_COLOR` signal, which is provided from `4002(RAM) #0`, output pin `O1`. A value of 0 indicates black ink, while a value of 1 selects red ink.

After a full line has been printed, the paper is advanced by asserting the `ADVANCE_PAPER` signal from `4002(RAM) #0`, output pin `O3`. This signal is also triggered when the paper feed button is manually pressed.

6.4.9 Paper Feed Button

The paper feed button is not part of the keyboard matrix. As shown in Figure 6.6, it is connected to `4001(ROM) #2` at input/output pin `IO3`.

6.4.10 Status Indicator Lamps

Three status indicator lamps are provided to reflect internal conditions:

- **V**: Overflow occurred
- **N**: Result is negative
- **M**: Memory data valid

These indicators are controlled via `4002(RAM) #1`, output pins `O0` to `O2`, respectively.

6.4.11 `KEY_PRINTER` Controlled by the RISC-V Subsystem

The module `KEY_PRINTER` implements the logic that emulates the I/O circuitry of the previously described Busicom 141-PF calculator. The RTL description is provided in `key_printer.v`. `KEY_PRINTER` is controlled via the input/output ports of the RISC-V subsystem. Through coordinated operation between `KEY_PRINTER` and the RISC-V subsystem, the I/O behavior of the 141-PF calculator is faithfully reproduced.

Table 6.9 lists the command signals `PORT_KEYPRT_CMD[31:0]` sent from the RISC-V subsystem to `KEY_PRINTER`. Table 6.10 shows the response signals `PORT_KEYPRT_RES[31:0]` returned from `KEY_PRINTER` to the RISC-V subsystem.

PORT_KEYPRT_CMD[31:0]			
Bit	Field Name		Description
31:16	Unused		
15	PRT_FIFO_POP_REQ		Printer FIFO Read Request (Rising Edge Trigger)
14	PAPER_FEED_REQ		Printer Paper Feed Request
13	ROUND_SWITCH	ROUND1	Rounding Mode Switch Input 00: FLOATING (Display all significant digits) 01: ROUND (Round-off) 10: TRUNCATE (Truncate) 11: inhibitted
12		ROUND0	
11	DECIMAL_POINT_SWITCH	DP3	Decimal Places Switch Input 0000: 0 digits after decimal point 0001 – 0110: 1 to 6 digits after decimal point Others: 8 digits after decimal point
10		DP2	
9		DP1	
8		DP0	
7:0	KEY_CODE		Key Code

Table 6.9: Command Signals Sent to `KEY_PRINTER` (Connected to RISC-V Subsystem Ports)

PORT_KEYPRT_RES[31:0]		
Bit	Field_name	Description
31	PRT_FIFO_DATA_RDY	Indicates Data Availability in Printer FIFO
30:16	PRT_COLUMN_01_15	Printer FIFO Data: Column Print Position (01–15)
15:14	PRT_COLUMN_17_18	Printer FIFO Data: Column Print Position (17–18)
13:10	PRT_DRUM_COUNT	Printer FIFO Data: Drum Print Position (0x0–0xC)
9	PRT_RED_RIBON	Printer FIFO Data: Print Color (0: Black, 1: Red)
8	PRT_PAPER_FEED	Printer FIFO Data: Paper Feed Command
7	LAMP_MINUS_SIGN	Status Indicator Lamp: Minus Sign (Negative Value)
6	LAMP_OVERFLOW	Status Indicator Lamp: Overflow
5	LAMP_MEMORY	Status Indicator Lamp: Memory Data Validity
4:1	Unused	
0	PRT_FIFO_POP_ACK	Completion of Printer FIFO Data Retrieval

Table 6.10: Response Signals from KEY_PRINTER (Connected to RISC-V Subsystem Ports)

6.4.12 Keyboard and Switch Emulation

All control signals related to the keyboard and switches are encapsulated in the command signal described in Table 6.9, and are simply passed through to the KEY_PRINTER module. Upon receiving user interface instructions for key input, the RISC-V subsystem encodes the pressed key type and switch status into specific fields within the lower 15 bits of PORT_KEYPRT_CMD[31:0], which it then outputs. If no key is pressed, the 8-bit KEY_CODE field is set to 8'h00.

In response, KEY_PRINTER generates the row signals KBR0–KBR3 in synchronization with the column scan signals KBC0–KBC9. These signals are then output from KEY_PRINTER. Additionally, the state of the paper feed button is also output.

6.4.13 Printer Emulation

When the calculator program for the 141-PF initiates printer operations—such as striking the hammer, advancing the ink ribbon, or feeding paper—the action must be conveyed to the RISC-V subsystem for output to the serial terminal or display on a touch LCD panel.

Due to the complete asynchrony between the 141-PF program and the RISC-V subsystem software, a FIFO buffer with a depth of 256 stages is inserted to store printer operation information. When the 141-PF program performs an action—hammer strike, ribbon movement, or paper feed—the corresponding printer control signals (including drum column position, hammer column selection signals PRS00–PRS19, ink ribbon color selection PRINTING_COLOR, and paper feed signal ADVANCE_PAPER) are written into the FIFO.

This information is transmitted to the RISC-V subsystem via the response signal PORT_KEYPRT_RES[31:0] (see Table 6.10). When data is present in the FIFO, bit 15 PRT_FIFO_DATA_RDY is set to 1, indicating that the RISC-V subsystem should read the printer data from bits 8 to 30.

After the RISC-V subsystem reads the data, it asserts bit 15 PRT_FIFO_POP_REQ of PORT_KEYPRT_CMD[31:0] to inform KEY_PRINTER that the FIFO entry was consumed. Upon acknowledgment, KEY_PRINTER asserts bit 0 PRT_FIFO_POP_ACK of PORT_KEYPRT_RES[31:0], allowing the RISC-V subsystem to clear PRT_FIFO_POP_REQ.

The access procedure for the printer FIFO from the RISC-V subsystem is summarized as follows:

1. Periodically monitor PORT_KEYPRT_RES[31:0] for PRT_FIFO_DATA_RDY being set.
2. If set, read printer information from bits 8–30.

3. Assert `PRT_FIFO_POP_REQ`.
4. Wait until `PRT_FIFO_POP_ACK` is asserted.
5. Clear `PRT_FIFO_POP_REQ`.

6.4.14 Emulation of Calculator Status Lamps

The ON/OFF states of the three calculator status lamps are stored in bits 5 through 7 of the response signal `PORT_KEYPRT_RES[31:0]` from `KEY_PRINTER` to the RISC-V subsystem. By periodically monitoring these bits, the RISC-V subsystem reflects the lamp status to the user interface.

6.4.15 Synchronization Across Clock Domains

The RISC-V subsystem (operating at 20 MHz) and `KEY_PRINTER` (operating at 750 kHz) function in an asynchronous relationship. Among the command signals sent from the RISC-V subsystem and received by `KEY_PRINTER`, key input signals do not undergo synchronization processing, as debounce handling is performed by the 4004 CPU-side software.

However, the `PRT_FIFO_POP_REQ` signal—used to retrieve printer output data from the FIFO—is synchronized to prevent metastability-induced malfunctions. Response signals from `KEY_PRINTER` to the RISC-V subsystem are not synchronized. Nevertheless, during printer FIFO data retrieval, a handshake protocol is followed using `REQ` (request) and `ACK` (acknowledge) signals, ensuring stable operation without explicit synchronization logic.

6.5 Program for the 4004 Calculator Model 141-PF

6.5.1 Program Stored in MCS4_ROM

The ROM module of the MCS4_ROM, which stores programs for the MCS-4 system, is described in RTL reminiscent of SRAM. When synthesized for FPGA, it automatically uses the Block RAM of the FPGA. Since the actual implementation is RAM, the contents are initialized with `$readmemh()` when power is applied to the FPGA at startup. This initialization code is the file `4001.code`, which is identical to the genuine program code for the Busicom calculator Model 141-PF using MCS-4 chips. The file can be downloaded from `http://www.4004.com/assets/busicom-141pf-simulator-w-flowchart-071113.zip`. The system is constructed, including peripheral circuits, to operate this `4001.code` without any changes.

6.5.2 Artistic Program for the 4004 Calculator Model 141-PF

A detailed explanation of this program's contents, including its disassembly list, is available in the file `Busicom-141PF-Calculator_asm_rel-1-0-1.txt`, which can be downloaded from `http://www.4004.com/2009/Busicom-141PF-Calculator_asm_rel-1-0-1.txt`. Reading this file reveals how exceptionally well-designed this program is, making it feel like solving a puzzle. For keyboard matrix input scanning, the program implements features that eliminate chattering while detecting simultaneous presses of keys in the same column, thereby excluding such inputs. Additionally, even while printing results after an operation, the program allows up to 8 stroke inputs for the next keys, enhancing usability.

6.5.3 Optional Square Root Calculations

Square root calculations are implemented in the 141-PF calculator and also operate in the newly created system. The performance of these calculations is surprisingly fast. The file `Busicom-141PF-Calculator_asm_rel-1-0-1.txt` contains a detailed explanation of the algorithm, which is simply marvelous. Incidentally, this square root calculation was treated as an option for the 141-PF, and most of the actual machines produced at the time lacked square root keys. To add square root calculations, an additional 4001 (ROM) containing the square root program needs to be integrated.

Apart from square root calculations, the basic calculator program is 1024 bytes (addresses 0x000–0x3FF). The square root program occupies 256 bytes (addresses 0x400–0x4FF). Both are highly compact, with no need for kilobytes or megabytes.

The program for the 4004 Calculator Model 141-PF is an intricately thought-out masterpiece. It is truly astonishing how all these functions are contained in less than 1280 bytes of code.

6.6 RISC-V Subsystem

6.6.1 Hardware of the RISC-V Subsystem

The RISC-V subsystem leverages the SoC system design based on the `mmRISC-1` core [4]. Detailed information about the design of `mmRISC-1` and its SoC system can be found at `https://github.com/munetomo-maruyama/mmRISC-1`.

The RTL of the built-in RISC-V subsystem is stored under the directory `RTL/RISCV/`. The top-level description of the RISC-V subsystem is `RTL/RISCV/riscv_top/riscv_top.v`. Debug interface supports only the 4-wire JTAG, with the 2-wire cJTAG support being removed.

The original configuration for GPIO was 32 bits × 3 (GPIO0[31:0]–GPIO2[31:0]). However, this system extends the GPIO to 32 bits × 6 (GPIO0[31:0]–GPIO5[31:0]). Consequently, the `PORT` module description at `RTL/RISCV/port/port.v` has been modified. The address mapping for registers in the extended `PORT` module is shown in Table 6.11. For GPIO3 to GPIO5, additional registers (`PDR3`–`PDR5`, `PDD3`–`PDD5`) have been implemented with the same functionalities as those of `PDR0`–`PDR2` and `PDD0`–`PDD2`.

Offset	Name	Description
0x00	PDR0	Port Data Register 0
0x04	PDR1	Port Data Register 1
0x08	PDR2	Port Data Register 2
0x0c	PDR3	Port Data Register 3
0x10	PDR4	Port Data Register 4
0x14	PDR5	Port Data Register 5
0x20	PDD0	Port Data Direction 0
0x24	PDD1	Port Data Direction 1
0x28	PDD2	Port Data Direction 2
0x2c	PDD3	Port Data Direction 3
0x30	PDD4	Port Data Direction 4
0x34	PDD5	Port Data Direction 5

Table 6.11: Registers in the PORT Module

Among the ports, GPIO0 to GPIO2 are routed externally from the FPGA, as shown

RISC-V GPIO	FPGA Pin	Connection on DE10-Lite Board			Note
GPIO0[0]	C14	7-seg LED 0	HEX00	segA	
GPIO0[1]	E15		HEX01	segB	
GPIO0[2]	C15		HEX02	segC	
GPIO0[3]	C16		HEX03	segD	
GPIO0[4]	E16		HEX04	segE	
GPIO0[5]	D17		HEX05	segF	
GPIO0[6]	C17		HEX06	segG	
GPIO0[7]	D15		HEX07	segDP	
GPIO0[8]	C18	7-seg LED 1	HEX10	segA	
GPIO0[9]	D18		HEX11	segB	
GPIO0[10]	E18		HEX12	segC	
GPIO0[11]	B16		HEX13	segD	
GPIO0[12]	A17		HEX14	segE	
GPIO0[13]	A18		HEX15	segF	
GPIO0[14]	B17		HEX16	segG	
GPIO0[15]	A16		HEX17	segDP	Turns on at a Low level.
GPIO0[16]	B20	7-seg LED 2	HEX20	segA	
GPIO0[17]	A20		HEX21	segB	
GPIO0[18]	B19		HEX22	segC	
GPIO0[19]	A21		HEX23	segD	
GPIO0[20]	B21		HEX24	segE	
GPIO0[21]	C22		HEX25	segF	
GPIO0[22]	B22		HEX26	segG	
GPIO0[23]	A19		HEX27	segDP	
GPIO0[24]	F21	7-seg LED 3	HEX30	segA	
GPIO0[25]	E22		HEX31	segB	
GPIO0[26]	E21		HEX32	segC	
GPIO0[27]	C19		HEX33	segD	
GPIO0[28]	C20		HEX34	segE	
GPIO0[29]	D19		HEX35	segF	
GPIO0[30]	E17		HEX36	segG	
GPIO0[31]	D22		HEX37	segDP	

Table 6.12: GPIO0 connected to External Pins

in Tables 6.12, 6.13, and 6.14. Meanwhile, GPIO3 to GPIO5 are connected internally within the FPGA, as detailed in Tables 6.15, 6.16, and 6.17.

6.6.2 Software of the RISC-V Subsystem

The software for the RISC-V subsystem was developed using a GCC-based environment within the integrated development environment `Eclipse Embedded CDT`. The workspace and source code are stored in the directory `SOFTWARE/workspace/MCS4_141PF`.

After startup, the main routine (`main.c`) determines the presence and type of the touch LCD panel (capacitive or resistive). If the touch LCD panel is not connected, subsequent operations are conducted through the serial terminal. The serial terminal should be configured to `115200bps`, 8N1 format. When the serial terminal is selected as the user interface, control is handed over to the routines in `src_app/src/cui_141pf.c`.

If the touch LCD panel is connected, control is passed to the routines in `src_app/src/gui_141pf.c`, enabling graphical calculator operations through touch interaction.

On the DE10-Lite board, resetting with `SW8 (GPIO2[8])` in the OFF (Low level) state runs the calculator program for the Model 141-PF. When `SW8` is ON (High level) during reset, the program for 500-digit pi calculation, which will be explained later, is loaded and executed in the `MCS4_ROM` module. Switching `SW8` back to OFF and resetting will reload the 141-PF calculator program into the `MCS4_ROM` module for execution.

The 500-digit pi calculation program solely prints the results. Even if the touch LCD panel is connected, the calculation results are displayed on the serial terminal.

The software for the RISC-V subsystem is constructed to run on the FreeRTOS for

RISC-V GPIO	FPGA Pin	Connection on DE10-Lite Board			Note
GPIO1[0]	F18	7-seg LED 4	HEX40	segA	Turns on at a Low level.
GPIO1[1]	E20		HEX41	segB	
GPIO1[2]	E19		HEX42	segC	
GPIO1[3]	J18		HEX43	segD	
GPIO1[4]	H19		HEX44	segE	
GPIO1[5]	F19		HEX45	segF	
GPIO1[6]	F20		HEX46	segG	
GPIO1[7]	F17		HEX47	segDP	
GPIO1[8]	J20	7-seg LED 5	HEX50	segA	
GPIO1[9]	K20		HEX51	segB	
GPIO1[10]	L18		HEX52	segC	
GPIO1[11]	N18		HEX53	segD	
GPIO1[12]	M20		HEX54	segE	
GPIO1[13]	N19		HEX55	segF	
GPIO1[14]	N20		HEX56	segG	
GPIO1[15]	L19		HEX57	segDP	
GPIO1[16]	A8	LED	LEDR0		Turns on at a High level.
GPIO1[17]	A9		LEDR1		
GPIO1[18]	A10		LEDR2		
GPIO1[19]	B10		LEDR3		
GPIO1[20]	D13		LEDR4		
GPIO1[21]	C13		LEDR5		
GPIO1[22]	E14		LEDR6		
GPIO1[23]	D14		LEDR7		
GPIO1[24]	A11		LEDR8		
GPIO1[25]	B11		LEDR9		
GPIO1[26]	AA5	General Purpose I/O Port	GPIO1[26]	TT_SEL_INC	Control Signals for TinyTapeout Chip
GPIO1[27]	AA6		GPIO1[27]	TT_SEL_RST_N	
GPIO1[28]	AA7		GPIO1[28]	TT_ENA	

Table 6.13: GPIO1 connected to External Pins

RISC-V GPIO	FPGA Pin	Connection on DE10-Lite Board			Note
GPIO2[0]	C10	Slide Switch	SW0		Slide up for High Level.
GPIO2[1]	C11		SW1		
GPIO2[2]	D12		SW2		
GPIO2[3]	C12		SW3		
GPIO2[4]	A12		SW4		
GPIO2[5]	B12		SW5		
GPIO2[6]	A13		SW6		
GPIO2[7]	A14		SW7		
GPIO2[8]	B14		SW8		
GPIO2[9]	F15		SW9	Connect external MCS4 CPU	
GPIO2[10]	A7	Push Switch	KEY1	Reset Halt	Push to Low level.

Table 6.14: GPIO2 connected to External Pins

RISC-V GPIO	KEYPRT Command Signal	Bit Field	Note
GPIO3[0]	PORT_KEYPRT_CMD[0]	KEY_CODE[0]	Key Board Input Code
GPIO3[1]	PORT_KEYPRT_CMD[1]	KEY_CODE[1]	
GPIO3[2]	PORT_KEYPRT_CMD[2]	KEY_CODE[2]	
GPIO3[3]	PORT_KEYPRT_CMD[3]	KEY_CODE[3]	
GPIO3[4]	PORT_KEYPRT_CMD[4]	KEY_CODE[4]	
GPIO3[5]	PORT_KEYPRT_CMD[5]	KEY_CODE[5]	
GPIO3[6]	PORT_KEYPRT_CMD[6]	KEY_CODE[6]	
GPIO3[7]	PORT_KEYPRT_CMD[7]	KEY_CODE[7]	
GPIO3[8]	PORT_KEYPRT_CMD[8]	DP0	Decimal Point SW
GPIO3[9]	PORT_KEYPRT_CMD[9]	DP1	
GPIO3[10]	PORT_KEYPRT_CMD[10]	DP2	
GPIO3[11]	PORT_KEYPRT_CMD[11]	DP3	
GPIO3[12]	PORT_KEYPRT_CMD[12]	ROUND0	Round Mode SW
GPIO3[13]	PORT_KEYPRT_CMD[13]	ROUND1	
GPIO3[14]	PORT_KEYPRT_CMD[14]	PAPER_FEED_REQ	
GPIO3[15]	PORT_KEYPRT_CMD[15]	PRT_FIFO_POP_REQ	Printer FIFO Read Request (Rising Edge Trigger)
GPIO3[16]	PORT_KEYPRT_CMD[16]	unused	
GPIO3[17]	PORT_KEYPRT_CMD[17]	unused	
GPIO3[18]	PORT_KEYPRT_CMD[18]	unused	
GPIO3[19]	PORT_KEYPRT_CMD[19]	unused	
GPIO3[20]	PORT_KEYPRT_CMD[20]	unused	
GPIO3[21]	PORT_KEYPRT_CMD[21]	unused	
GPIO3[22]	PORT_KEYPRT_CMD[22]	unused	
GPIO3[23]	PORT_KEYPRT_CMD[23]	unused	
GPIO3[24]	PORT_KEYPRT_CMD[24]	unused	
GPIO3[25]	PORT_KEYPRT_CMD[25]	unused	
GPIO3[26]	PORT_KEYPRT_CMD[26]	unused	
GPIO3[27]	PORT_KEYPRT_CMD[27]	unused	
GPIO3[28]	PORT_KEYPRT_CMD[28]	unused	
GPIO3[29]	PORT_KEYPRT_CMD[29]	unused	
GPIO3[30]	PORT_KEYPRT_CMD[30]	unused	
GPIO3[31]	PORT_KEYPRT_CMD[31]	unused	

Table 6.15: GPIO3 connected to Internal Signals

RISC-V GPIO	KEYPRT Response Signal	Bit Field	Note
GPIO4[0]	PORT_KEYPRT_RES[0]	PRT_FIFO_POP_ACK	Completion of Printer FIFO Data Retrieval
GPIO4[1]	PORT_KEYPRT_RES[1]	unused	
GPIO4[2]	PORT_KEYPRT_RES[2]	unused	
GPIO4[3]	PORT_KEYPRT_RES[3]	unused	
GPIO4[4]	PORT_KEYPRT_RES[4]	unused	
GPIO4[5]	PORT_KEYPRT_RES[5]	LAMP_MEMORY	Status Indicator Lamp: Memory Data Validity
GPIO4[6]	PORT_KEYPRT_RES[6]	LAMP_OVERFLOW	Status Indicator Lamp: Overflow
GPIO4[7]	PORT_KEYPRT_RES[7]	LAMP_MINUS_SIGN	Status Indicator Lamp: Minus Sign (Negative Value)
GPIO4[8]	PORT_KEYPRT_RES[8]	PRT_PAPER_FEED	Printer FIFO Data: Paper Feed Command
GPIO4[9]	PORT_KEYPRT_RES[9]	PRT_RED_RIBON	Printer FIFO Data: Print Color (0: Black, 1: Red)
GPIO4[10]	PORT_KEYPRT_RES[10]	PRT_DRUM_COUNT[0]	Printer FIFO Data: Drum Print Position (0x0–0xC)
GPIO4[11]	PORT_KEYPRT_RES[11]	PRT_DRUM_COUNT[1]	
GPIO4[12]	PORT_KEYPRT_RES[12]	PRT_DRUM_COUNT[2]	
GPIO4[13]	PORT_KEYPRT_RES[13]	PRT_DRUM_COUNT[3]	
GPIO4[14]	PORT_KEYPRT_RES[14]	PRT_COLUMN_17_18[18]	Printer FIFO Data: Column Print Position (17–18)
GPIO4[15]	PORT_KEYPRT_RES[15]	PRT_COLUMN_17_18[17]	
GPIO4[16]	PORT_KEYPRT_RES[16]	PRT_COLUMN_01_15[15]	Printer FIFO Data: Column Print Position (01–15)
GPIO4[17]	PORT_KEYPRT_RES[17]	PRT_COLUMN_01_15[14]	
GPIO4[18]	PORT_KEYPRT_RES[18]	PRT_COLUMN_01_15[13]	
GPIO4[19]	PORT_KEYPRT_RES[19]	PRT_COLUMN_01_15[12]	
GPIO4[20]	PORT_KEYPRT_RES[20]	PRT_COLUMN_01_15[11]	
GPIO4[21]	PORT_KEYPRT_RES[21]	PRT_COLUMN_01_15[10]	
GPIO4[22]	PORT_KEYPRT_RES[22]	PRT_COLUMN_01_15[9]	
GPIO4[23]	PORT_KEYPRT_RES[23]	PRT_COLUMN_01_15[8]	
GPIO4[24]	PORT_KEYPRT_RES[24]	PRT_COLUMN_01_15[7]	
GPIO4[25]	PORT_KEYPRT_RES[25]	PRT_COLUMN_01_15[6]	
GPIO4[26]	PORT_KEYPRT_RES[26]	PRT_COLUMN_01_15[5]	
GPIO4[27]	PORT_KEYPRT_RES[27]	PRT_COLUMN_01_15[4]	
GPIO4[28]	PORT_KEYPRT_RES[28]	PRT_COLUMN_01_15[3]	
GPIO4[29]	PORT_KEYPRT_RES[29]	PRT_COLUMN_01_15[2]	
GPIO4[30]	PORT_KEYPRT_RES[30]	PRT_COLUMN_01_15[1]	
GPIO4[31]	PORT_KEYPRT_RES[31]	PRT_FIFO_DATA_RDY	Indicates Data Availability In Printer FIFO

Table 6.16: GPIO4 connected to Internal Signals

RISC-V GPIO	MCS4_ROM Initialization	Bit Field	Note
GPIO5[0]	ROM_INIT_RDATA[0]	ROM Read Data for Initialization	input
GPIO5[1]	ROM_INIT_RDATA[1]		input
GPIO5[2]	ROM_INIT_RDATA[2]		input
GPIO5[3]	ROM_INIT_RDATA[3]		input
GPIO5[4]	ROM_INIT_RDATA[4]		input
GPIO5[5]	ROM_INIT_RDATA[5]		input
GPIO5[6]	ROM_INIT_RDATA[6]		input
GPIO5[7]	ROM_INIT_RDATA[7]		input
GPIO5[8]	ROM_INIT_WDATA[0]	ROM Write Data for Initialization	output
GPIO5[9]	ROM_INIT_WDATA[1]		output
GPIO5[10]	ROM_INIT_WDATA[2]		output
GPIO5[11]	ROM_INIT_WDATA[3]		output
GPIO5[12]	ROM_INIT_WDATA[4]		output
GPIO5[13]	ROM_INIT_WDATA[5]		output
GPIO5[14]	ROM_INIT_WDATA[6]		output
GPIO5[15]	ROM_INIT_WDATA[7]		output
GPIO5[16]	ROM_INIT_ADDR[0]	ROM Address for Initialization	output
GPIO5[17]	ROM_INIT_ADDR[1]		output
GPIO5[18]	ROM_INIT_ADDR[2]		output
GPIO5[19]	ROM_INIT_ADDR[3]		output
GPIO5[20]	ROM_INIT_ADDR[4]		output
GPIO5[21]	ROM_INIT_ADDR[5]		output
GPIO5[22]	ROM_INIT_ADDR[6]		output
GPIO5[23]	ROM_INIT_ADDR[7]		output
GPIO5[24]	ROM_INIT_ADDR[8]		output
GPIO5[25]	ROM_INIT_ADDR[9]		output
GPIO5[26]	ROM_INIT_ADDR[10]		output
GPIO5[27]	ROM_INIT_ADDR[11]		output
GPIO5[28]	ROM_INIT_RE	ROM Read Enable for initialization	output
GPIO5[29]	ROM_INIT_WE	ROM Write Enable for initialization	output
GPIO5[30]	ROM_INIT_ENB	Initialization Mode of MCS4 ROM	output
GPIO5[31]	MCS4_RES_N	Reset MCS4	output

Table 6.17: GPIO5 connected to Internal Signals

RISC-V ported by the author for `mmRISC-1`. The RTOS kernel is stored in `src/src_kernel`, while the applications are stored in `src/src_app`. In the calculator control routines (`src_app/src/cui_141pf.c` and `src_app/src/gui_141pf.c`), tasks for keyboard control and printer control operate independently. Furthermore, in `src_app/src/cui_141pf.c`, using the serial terminal, a semaphore notifies the keyboard control task whenever the UART receives a character, triggered by its interrupt service routine.

6.6.3 When Modifying Software of the RISC-V Subsystem

The binary software of the RISC-V subsystem initializes the memory components (`s_mem0` to `s_mem3`) within the instruction memory (`RISCV/ram/rami.v`) using the `$readmemh` command. Each memory component is byte-wide, and their respective initialization files are `MCS4_141PF.rom0` to `MCS4_141PF.rom3`. These files are created by converting the Intel Hex binary `SOFTWARE/workspace/MCS4_141PF/debug/MCS4_141PF.hex` of the program. The conversion tools are located in the `SOFTWARE/tools` directory: `hex2v32_lane0.exe` to `hex2v32_lane3.exe`. These tools are built using source files written in standard C (`hex2v32_lane0.c` to `hex2v32_lane3.c`).

Once the four conversion tools are prepared, navigate to the directory `RTL/RISCV/ram` and execute the shell script below to generate the initialization files for `$readmemh`:

```
$ ./gen_init
```

Therefore, whenever modifying the `SOFTWARE/workspace/MCS4_141PF`, ensure to update the initial values of the instruction memory by running the command above. Subsequently, synthesize the FPGA, and the updated program will operate starting from power-on. The changes will also reflect in the logic simulation described later.

6.7 Logical Simulation of the MCS-4 System

6.7.1 Logical Simulation of the MCS-4 System Using Icarus Verilog

The MCS-4 system was simulated separately using Icarus Verilog, while the entire setup including FPGA macro elements and the RISC-V subsystem was simulated using Questa Altera FPGA Starter Edition. This section details the simulation process for the MCS-4 system.

6.7.2 Resources for Simulation

Resources required for logical simulation with Icarus Verilog are located under the directory `SIM_iverilog`. Descriptions in this section assume this directory as the working directory.

6.7.3 Installing Icarus Verilog

Install Icarus Verilog according to your environment by referring to online information. Also, install the waveform viewer `gtkwave` for reviewing simulation results.

6.7.4 For Simulation Using 141-PF Calculator Program

The RTL description of the `MCS4_ROM` module initializes the ROM memory with `4001.code`. Without modification, you can perform the logical simulation using this setup.

6.7.5 For Simulation Using Custom 4004 Program Code

To simulate using your custom 4004 (CPU) program code:

1. Assemble the source program (`.src`) using the assembler tool ADS4004 to generate the Intel Hex format binary file (`.hex`).

2. Convert this file to a Verilog HDL initialization file using the `hex2v` tool, whose C source is located at `../SOFTWARE/tools/hex2v.c`.

3. For example:

```
$ ../SOFTWARE/tools/hex2v.exe ../SOFTWARE/ADS4004/test.src.hex > ../RTL/MCS4/
```

4. Modify the RTL description of `MCS4_ROM` (`../RTL/MCS4/mcs4_rom.v`) as follows:

```
$readmemh("../RTL/MCS4/test.src.hex.code", rom);
```

The preparation of the 4004 (CPU) program is now complete.

6.7.6 Testbench for 141-PF calculator

The top-level testbench for logical simulation is `tb_TOP.v`. The testbench is designed to verify the operation of the 141-PF calculator by directly manipulating `PORT_KEYPRT_CMD[31:0]` and `PORT_KEYPRT_RES[31:0]` stimuli to simulate keyboard inputs and retrieve printer data. For logical verification with custom 4004 (CPU) program codes, please modify the testbench as necessary.

6.7.7 Execution of the Simulation

All RTL descriptions required for logical simulation are listed in the file `flist`. The execution steps for the logical simulation have been compiled into the shell script `run_iverilog`. You can execute the script as follows:

```
$ ./run_iverilog
```

The RTL descriptions will be compiled by the `iverilog` command, and if no errors occur, the executable file `tb.vvp` is produced. This file is then passed to the main logic simulator `vvp`, which executes the logical simulation and generates the waveform file `tb.vcd`.

6.7.8 Viewing Waveform Files

View the resulting waveform file (`tb.vcd`) using `gtkwave`:

```
$ gtkwave.exe tb.vcd tb.gtkw
```

The second argument specified as `tb.gtkw` is a configuration file for the waveform viewer. This file allows you to freely select display waveforms and specify their order as desired. After arranging the waveform display methods on the viewer and saving the configuration, you can view the waveforms with the same display settings in subsequent sessions as shown in Figure 6.8.

Figure 6.8: Waveform of 141-PF Simulation

6.8 Logical Simulation of the Entire FPGA System

6.8.1 Logical Simulation of the Entire FPGA System Using Questa

The entire FPGA system, including FPGA macro elements and the RISC-V subsystem, is simulated using Questa Altera FPGA Starter Edition. Related resources for Questa-based simulation are stored under the directory `SIM_questa`. The following explanation assumes this directory as the working directory.

6.8.2 Executing Logical Simulation Using Questa

To start Questa, execute the following command:

```
$ vsim.exe
```

Then, within the Transcript window of Questa, execute:

```
Questa> do sim_TOP.do
```

Here, the file `sim_TOP.do`, specified as an argument for the `do` command, is a TCL script that includes the RTL descriptions used for simulation, simulation commands, and waveform display settings. You can modify the waveform display settings and other details as desired. An example of Questa simulation is shown in Figure 6.9.

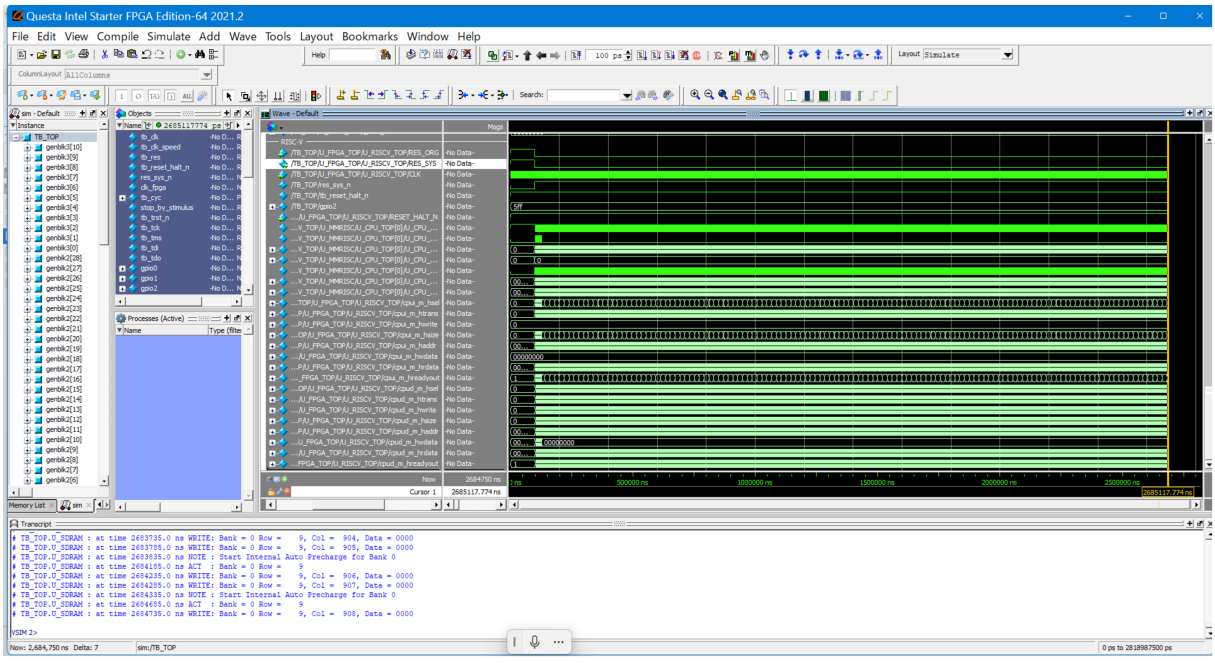


Figure 6.9: Waveform of FPGA Simulation

In this logical simulation, the RISC-V instruction memory `../RTL/RISCV/ram/rami.v` is initialized using `../SOFTWARE/workspace/MCS4_141PF/Debug/MCS4_141PF.hex`. Since the execution of this program takes time, the purpose of this simulation is primarily to verify the absence of errors in the RTL description, including the top-level FPGA hierarchy. For a detailed inspection of the RISC-V subsystem's operation, it is recommended to replace the RISC-V program with a simpler one and re-execute the simulation.

6.9 FPGA Implementation

The implementation of Altera FPGA MAX 10 is conducted using the Quartus Prime tool. FPGA-related resources are stored in the directory `FPGA/`.

Steps for Synthesis and Configuration

1. Open the Quartus Prime project file located at `FPGA/FPGA.qpf`.

2. Perform the synthesis process.

3. After a successful synthesis without any errors, generate the configuration file (`FPGA/output_files/FPGA.pof`).

4. Use the Quartus Prime tool "Programmer" to write the configuration file to the FPGA.

6.9.1 Signal Assignments for External FPGA Terminals

For the RISC-V subsystem's external terminal assignments related to GPIOs, please refer to Tables **??**–**??**. The assignments for other RISC-V-related external terminals are detailed in Table 6.18.

Within the MCS-4 system, the MCS4_CPU (4004 CPU) has a mode available for external connection. The signal assignments for this external connection can be found in Table 6.19.

6.9.2 MCS4_CPU Connection Methods and DE10-Lite Board Settings

Case A: When the MCS4_CPU (4004 CPU) is connected and operated within the FPGA, the DE10-Lite board settings are shown in Figure 6.10. There are no external connections related to the MCS4_CPU (4004 CPU). In this mode, set the slide switch SW9 to OFF.

Case B: When using the internal logic of the MCS4_CPU (4004 CPU) in the FPGA but connecting through external terminals, the DE10-Lite board settings are shown in Figure 6.11. Connect the MCS4_CPU (4004 CPU) with the MCS4_SYS via external connectors. As indicated by the red lines, connect the opposing terminals. In this mode, set the slide switch SW9 to ON.

Case C: When connecting an external device with equivalent functionality to the MCS4_CPU (4004 CPU) via external terminals, the DE10-Lite board settings are shown in Figure 6.12. Connect the external device's MCS4_CPU (4004 CPU) to the terminals on the MCS4_SYS side. In this mode, set the slide switch SW9 to ON.

For all cases above, the UART communication signals (TXD, RXD) should be connected to terminal software on a PC (115200bps, 8N1) via a USB-UART conversion board or the USB-JTAG interface board described later.

Signal	FPGA Pin	Connection on DE10-Lite Board	Note
RES_N	B8	KEY0	
CLK50	P11	Oscillator (50MHz)	
RESOUT_N	F16	RESET Output for Auduino Shield (negative)	
TRSTn	Y5	RISC-V JTAG Debugger Interface	
TCK	Y6		
TMS	AA2		
TDI	Y4		
TDO	Y3		
TXD	AB2	UART	Output
RXD	AB3		Input
I2C0_SCL	AB15	GSENSOR SCL	
I2C0_SDA	V11	GSENSOR SDA	
I2C0_ENA	AB16	GSENSOR CSn (Fixed to 1)	
I2C0_ADR	V12	GSENSOR ALTADDR (Fixed to 0)	
I2C0_INT1	Y14	GSENSOR INT1	
I2C0_INT2	Y13	GSENSOR INT2	
I2C1_SCL	AA20	Arduino IO15 CT_SCL (Capacitive Touch Controller)	
I2C1_SDA	AB21	Arduino IO14 CT_SDA (Capacitive Touch Controller)	
SPI_CSN[3]	AB9	Arduino IO04 CARD_CS (SD Card)	
SPI_CSN[2]	AB17	Arduino IO08 RT_CS (Resistive Touch Controller)	
SPI_CSN[1]	AA17	Arduino IO09 TFT_DC (LCD Controller)	
SPI_CSN[0]	AB19	Arduino IO10 TFT_CS (LCD Controller)	
SPI_MOSI	AA19	Arduino IO11	
SPI_MISO	Y19	Arduino IO12	
SPI_SCK	AB20	Arduino IO13	
SDRAM_CLK	L14	SDRAM	
SDRAM_CKE	N22		
SDRAM_CSn	U20		
SDRAM_DQM [0]	V22		
SDRAM_DQM [1]	J21		
SDRAM_RASn	U22		
SDRAM_CASn	U21		
SDRAM_WEn	V20		
SDRAM_BA [0]	T21		
SDRAM_BA [1]	T22		
SDRAM_ADDR[0]	U17		
SDRAM_ADDR[1]	W19		
SDRAM_ADDR[2]	V18		
SDRAM_ADDR[3]	U18		
SDRAM_ADDR[4]	U19		
SDRAM_ADDR[5]	T18		
SDRAM_ADDR[6]	T19		
SDRAM_ADDR[7]	R18		
SDRAM_ADDR[8]	P18		
SDRAM_ADDR[9]	P19		
SDRAM_ADDR[10]	T20		
SDRAM_ADDR[11]	P20		
SDRAM_ADDR[12]	R20		
SDRAM_DQ [0]	Y21		
SDRAM_DQ [1]	Y20		
SDRAM_DQ [2]	AA22		
SDRAM_DQ [3]	AA21		
SDRAM_DQ [4]	Y22		
SDRAM_DQ [5]	W22		
SDRAM_DQ [6]	W20		
SDRAM_DQ [7]	V21		
SDRAM_DQ [8]	P21		
SDRAM_DQ [9]	J22		
SDRAM_DQ [10]	H21		
SDRAM_DQ [11]	H22		
SDRAM_DQ [12]	G22		
SDRAM_DQ [13]	G20		
SDRAM_DQ [14]	G19		
SDRAM_DQ [15]	F22		

Table 6.18: RISC-V related External Signals except for GPIO

MCS4_SYS Signals	FPGA Pin	Description	Note
S_CLK	V10	Clock	Output
S_RES_N	V9	Reset	Output
S_SYNC_N	V8	Synchronize	Input
S_TEST	V7	Test Condition	Output
S_CM_ROM_N	W6	Command Control for ROM	Input
S_CM_RAM_N[0]	W5	Command Control for RAM	Input
S_CM_RAM_N[1]	AA14	Command Control for RAM	Input
S_CM_RAM_N[2]	W12	Command Control for RAM	Input
S_CM_RAM_N[3]	AB12	Command Control for RAM	Input
S_DATA[0]	AB11	Data Bus (Open Drain, Internal Weak Pull Up)	Input/Output
S_DATA[1]	AB10	Data Bus (Open Drain, Internal Weak Pull Up)	Input/Output
S_DATA[2]	AA9	Data Bus (Open Drain, Internal Weak Pull Up)	Input/Output
S_DATA[3]	AA8	Data Bus (Open Drain, Internal Weak Pull Up)	Input/Output

MCS4_CPU Signals	FPGA Pin	Description	Note
C_CLK	W10	Clock	Input
C_RES_N	W9	Reset	Input
C_SYNC_N	W8	Synchronize	Output
C_TEST	W7	Test Condition	Input
C_CM_ROM_N	V5	Command Control for ROM	Output
C_CM_RAM_N[0]	AA15	Command Control for RAM	Output
C_CM_RAM_N[1]	W13	Command Control for RAM	Output
C_CM_RAM_N[2]	AB13	Command Control for RAM	Output
C_CM_RAM_N[3]	Y11	Command Control for RAM	Output
C_DATA[0]	W11	Data Bus (Open Drain, Internal Weak Pull Up)	Input/Output
C_DATA[1]	AA10	Data Bus (Open Drain, Internal Weak Pull Up)	Input/Output
C_DATA[2]	Y8	Data Bus (Open Drain, Internal Weak Pull Up)	Input/Output
C_DATA[3]	Y7	Data Bus (Open Drain, Internal Weak Pull Up)	Input/Output

Table 6.19: MCS4__CPU (4004 CPU) related External Signals

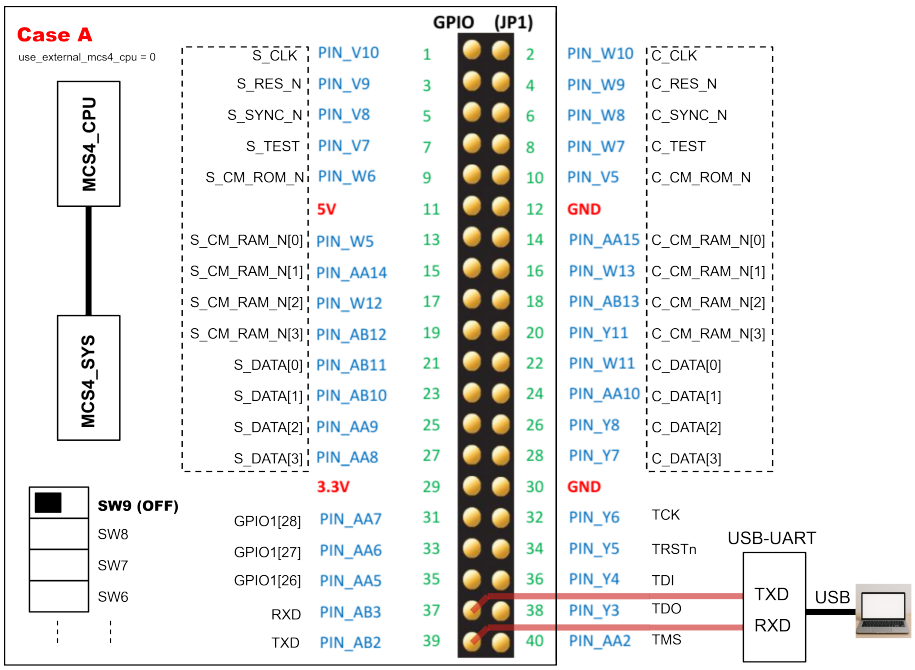


Figure 6.10: Case A: MCS4_CPU (4004 CPU) Connection

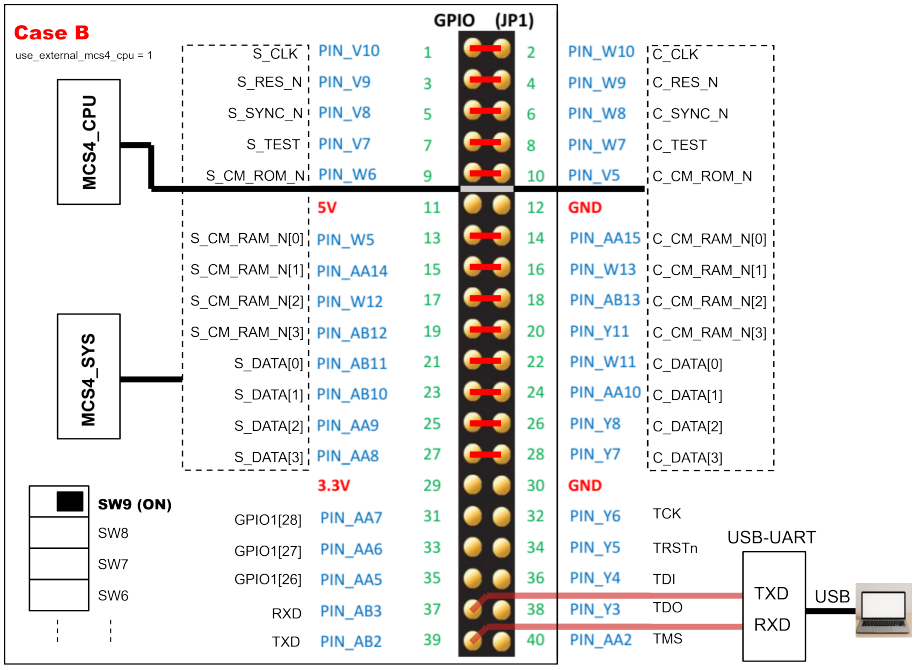


Figure 6.11: Case B: MCS4_CPU (4004 CPU) Connection

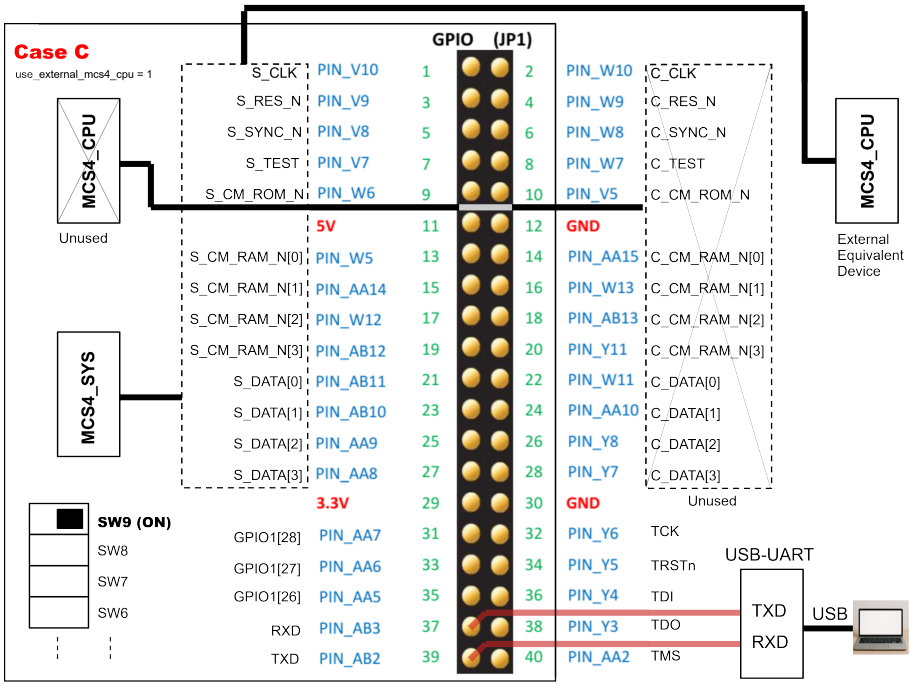


Figure 6.12: Case C: MCS4_CPU (4004 CPU) Connection

6.9.3 Connection for Debugging and Downloading RISC-V Programs

To debug RISC-V programs from the integrated development environment Eclipse, download them to instruction memory, and execute them, connect the PC via the USB-JTAG interface board as shown in Figure 6.13. A circuit example of the USB-JTAG interface board is depicted in Figure 6.14.

This interface board also provides USB-UART conversion functionality, enabling UART signals to be connected through it. For detailed methods of RISC-V debugging, please refer to the mmRISC-1 documentation [4].

6.10 Key Operation Method of the 141-PF Calculator

Examples of key operation methods for the actual 141-PF calculator are shown in Figure **??** and Figure **??**.

Addition and subtraction behave slightly differently from modern calculators. Press the "+" or "−" key *before* pressing the equal key. In other words, enter a value and then press "+" or "−". Please operate the keys as if saying aloud "add the number" or "subtract the number"—and do just that.

Multiplication and division follow standard procedures found in contemporary calculators. % Calculation, square root, and memory operations are handled similarly.

To input a negative number, press the "S" key before entering the numeric keys.

In the initial state, the number of digits after the decimal point is set to zero. Adjust the number of decimal places using the corresponding switches as needed. The rounding mode for the final digit can be selected via switch from the following options: Full significant digits display (FLOATING), round-off (ROUND), and truncation (TRUNCATE).

Additionally, three status indicator lamps are provided:

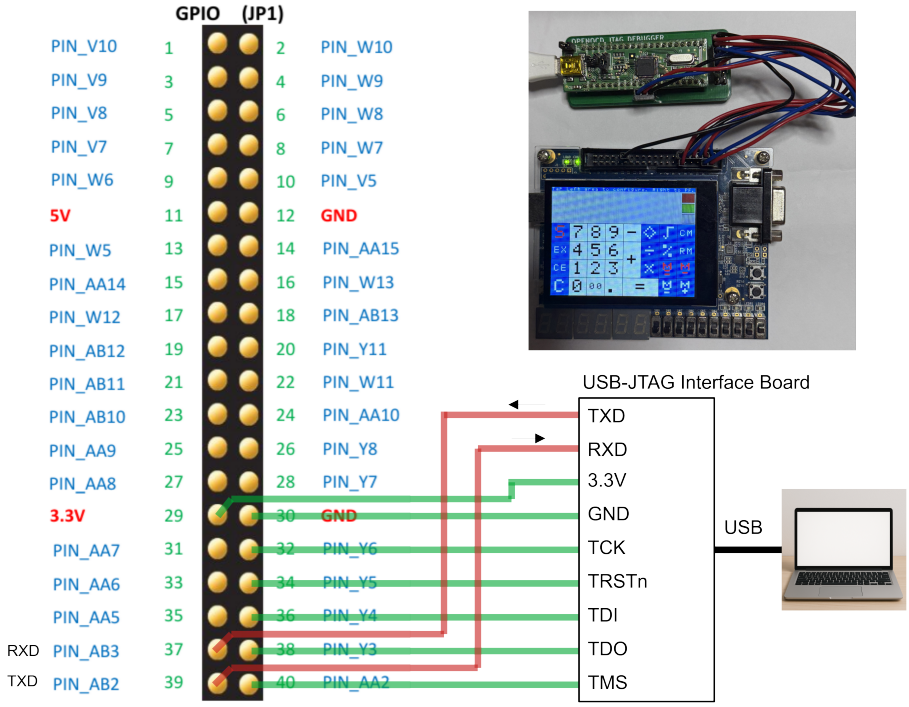


Figure 6.13: Connection of RISC-V USB-JTAG Interface Board

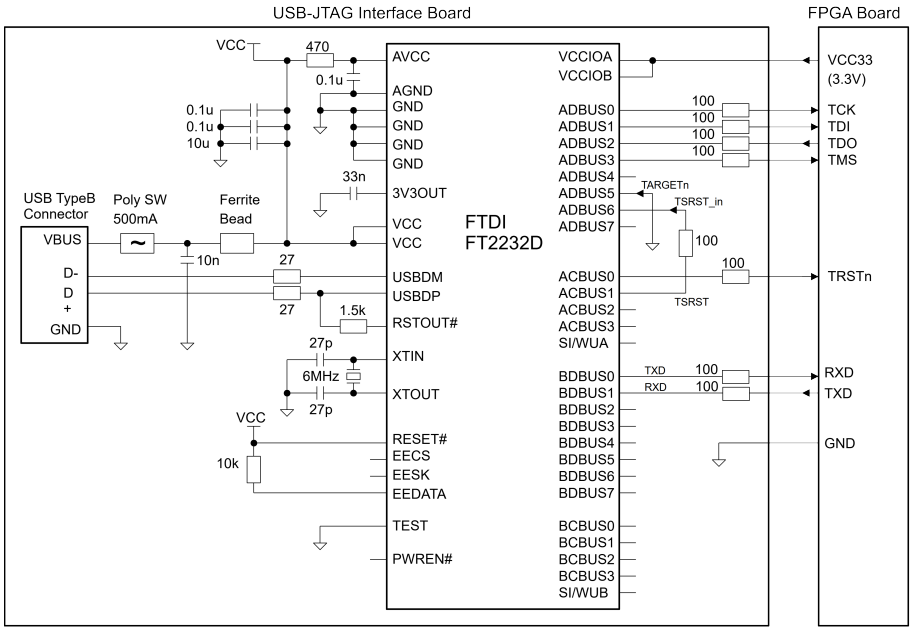


Figure 6.14: An example of schematics of RISC-V USB-JTAG Interface Board

- Overflow occurred (V)
- Negative result (N)
- Valid data stored in memory (M)

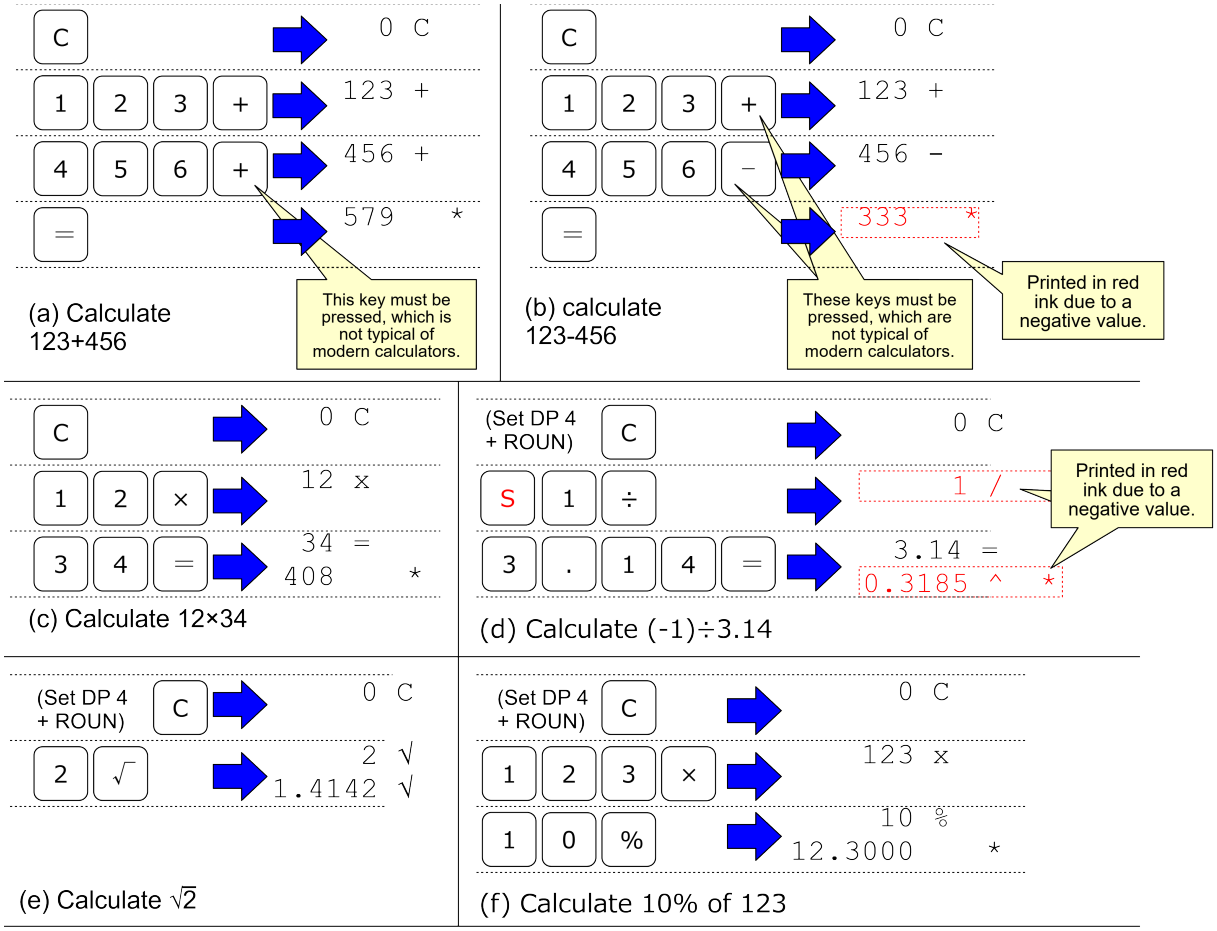


Figure 6.15: How to use the 141-PF (Normal Calculation)

6.11 The Charming Print Method of the 141-PF Calculator

Unlike modern devices that display results on LCDs, VFDs, or LEDs, the Busicom 141-PF calculator printed its output on paper using a printer. The printer employed was Shinshu Seiki ' s (now Seiko Epson) Model 102.

Inside the mechanism, a continuously rotating drum housed type characters embedded as reliefs. Above the drum were layers of paper, an ink ribbon, and a hammer. When the desired character on the drum rotated into alignment with the target print position, the hammer struck precisely, imprinting the character onto the paper.

On the calculator side, signals indicating the drum's rotational position were received, and for the characters most imminent on the drum, hammer signals were issued sequentially. As a result, digits in a print line did not appear left-to-right in order. Instead, the hammer struck various columns independently, and the full expression emerged as the print completed. This asynchronous yet elegant method delivers a truly unique and flavorful printing experience.

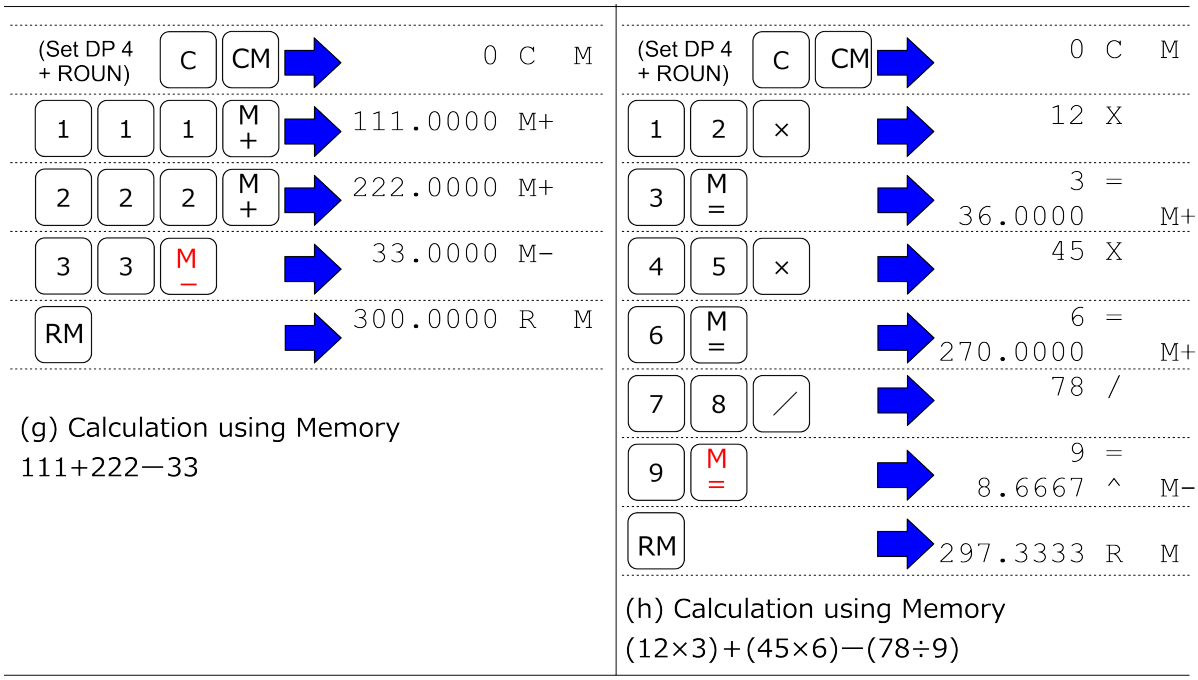


Figure 6.16: How to use the 141-PF (Memory Calculation)

From an efficiency standpoint, this technique enabled the fastest possible printing with drum-style mechanisms. One cannot help but feel the determination and ingenuity of the engineers behind it.

The ink ribbon used a nostalgic two-color scheme of black and red, reminiscent of old mechanical typewriters. For each character, the ribbon's vertical position shifted to match the desired color. On the 141-PF calculator, negative values were printed in red.

6.12 Operation Method Using a Serial Terminal in This FPGA System

This FPGA system was designed to replicate the operation of the Busicom 141-PF calculator. The following explains how to operate the system when using a serial terminal as the user interface.

The correspondence between the 141-PF calculator keys and characters entered from the terminal is shown in Table 6.20. Instead of pressing a physical key, input the corresponding single character via the serial terminal. Note that the entered character does not appear on the terminal immediately—it is displayed in synchronization with the printer's timing, thereby emulating the flavorful behavior of the original drum-style printer.

To confirm the input from the terminal, the entered character is displayed on the seven-segment LED of the DE10-Lite board. While it is not possible to visually represent all letters or symbols accurately using seven segments, the format shown in Figure 6.17 is used for display.

To operate the decimal point and rounding switches, input the "ESC" character on the serial terminal. This triggers the following configuration prompts, where you can set each option. Once both configurations are completed, the system will return to the key input standby state.

Key Board on 141-PF	Input Character in Serial Terminal	Function
S	S	Sign (to input negative value)
EX	X	Exchange Operands (for Division)
CE	E	Clear Input Numbers
C	C	Clear All
0	0	Zero
1	1	One
2	2	Two
3	3	Three
4	4	Four
5	5	Five
6	6	Six
7	7	Seven
8	8	Eight
9	9	Nine
00	Z	Double Zero
.	.	Period
+	+	Addition
-	-	Subtraction
x	*	Multiplication
÷	/	Division
=	=	Equal
◇	#	Print Again
√	Q	Square Root
%	%	Percent Calculation
CM	L	Clear Memory
RM	R	Read Memory
M+	P	Add to Memory
M-	M	Subtract from Memory
M=	D	After Equal, Add to Memory
M=	B	After Equal, Subtract from Memory
Paper Feed	F	Paper Feed

Table 6.20: Operation of Key Input on Serial Terminal

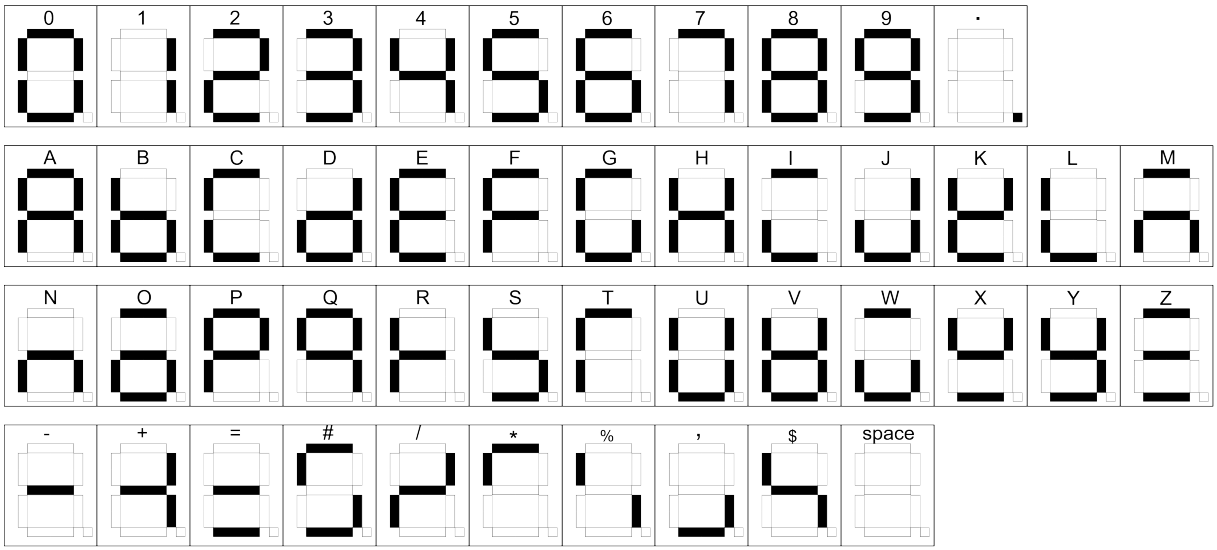


Figure 6.17: Character Display on 7 segment LED

The seven-segment LED display format was inspired by the console representation used in Hitachi' s 8-bit microcomputer 6800 training kit H68/TR, with some extensions made to suit this system.

https://www.hitachihyoron.com/jp/pdf/1977/05/1977_05_12.pdf

```
----------------------------
Set Slide Switches on 141-PF
----------------------------
Decimal Point : 0~6/8 (now 1)=?
Decimal Point SW Not Changed (now 1)
Rounding : FLOT=0/ROUN=1/TRNC=8 (now 0)=? 1
Rounding SW Changed (now 1)
----------------------------
```

Since the serial terminal cannot display red characters, lines that include red-printed values are marked with a [RED] tag at the end of the line. Additionally, the status indicators—overflow (V), negative result (N), and valid memory data (M)—are also displayed at the end of each line.

On the DE10-Lite board, LEDR0 through LEDR9 display the upper 10 bits (PC[11:2]) of the program counter (PC) of the MCS4_CPU (4004 CPU). This allows you to observe how the instruction address of the currently running program changes over time.

6.13 Operation Method Using the Touch LCD Shield in This FPGA System

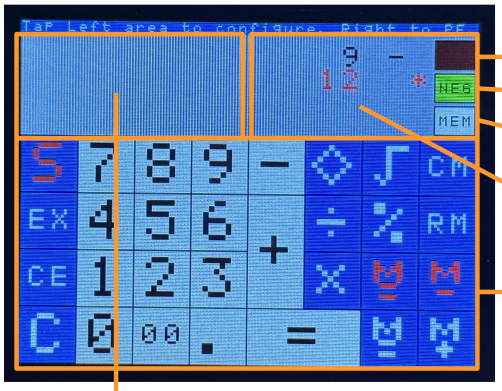
When the touch LCD shield is mounted on the designed FPGA, the graphical user interface allows intuitive operation as shown in Figure 6.18. This interface also faithfully reproduces the charming behavior of the original drum-style printer from the Busicom 141-PF calculator.

Touching the **left half of the print area** brings up a configuration window for controlling the **decimal point switch** and the **rounding switch**. Touching the **right half of the print area** performs a **paper feed** operation.

When each key is tapped on the touchscreen, the input character corresponding to the key is also shown on the **seven-segment LED** of the DE10-Lite board—just as in serial terminal operation. While seven-segment displays have limitations in expressing certain symbols and letters, this visual feedback supports confirmation of input.

Note that all touchscreen operations are designed to match the tactile feedback and timing of the original machine, preserving the authentic retrocomputing experience such as printing sequence of drum printer.

Just same as the serial terminal mode, on the DE10-Lite board, LEDR0 through LEDR9 display the upper 10 bits (PC[11:2]) of the program counter (PC) of the MCS4_CPU (4004 CPU). This allows you to observe how the instruction address of the currently running program changes over time.



Key Board

Key	Function
S	Sign (to input negative value)
EX	Exchange Operands (for Division)
CE	Clear Input Numbers
C	Clear All
0~9 00 .	Input Numbers
+ - X ÷	Addition Subtraction Multiplication Division
=	Equal
◇	Print Again
√	Square Root
%	Percent Calculation
CM	Clear Memory
RM	Read Memory
M+ M-	Add to Memory Subtract from Memory
M= M=	After Equal, Add to Memory After Equal, Subtract from Memory

Figure 6.18: Operation on Touch LCD Panel

Chapter 7

A challenge: Calculating 500 Digits of Pi with the 4004 CPU

In this chapter, we develop and execute a program in 4004 CPU assembly to compute 500 digits of the mathematical constant π. Despite being the world's first microprocessor developed during the dawn of microcontroller technology, the Intel 4004 exhibits surprisingly sufficient computational capability—even with its primitive instruction set.

Through this experiment, we aim to:

- Showcase the computational potential of the 4004 CPU despite its limitations.
- Implement high-precision arithmetic routines within the constraints of a 4-bit architecture.
- Demonstrate practical applications of early microprocessors for complex numerical tasks.
- Celebrate the architectural elegance and historical significance of the MCS-4 system.

The challenge underscores that even vintage systems can tackle sophisticated calculations with careful programming, optimization, and ingenuity.

7.1 Principle of Pi Calculation

This program utilizes Machin's formula to calculate the digits of π. Although the formula converges slowly, it is well-suited for implementation on simple microcontrollers.

Machin's formula is expressed as follows:

$$\frac{\pi}{4} = 4\arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right) \tag{7.1}$$

That is,

$$\pi = 16\arctan\left(\frac{1}{5}\right) - 4\arctan\left(\frac{1}{239}\right) \tag{7.2}$$

The arctangent function can be expanded into the infinite series:

$$\arctan\left(\frac{1}{p}\right) = \frac{1}{p} - \frac{1}{3p^3} + \frac{1}{5p^5} - \frac{1}{7p^7} + \cdots \tag{7.3}$$

Therefore, π can be computed as:

$$\pi = \left(16 \cdot \frac{1}{5} - 4 \cdot \frac{1}{239}\right) - \left(16 \cdot \frac{1}{3 \cdot 5^3} - 4 \cdot \frac{1}{3 \cdot 239^3}\right) + \left(16 \cdot \frac{1}{5 \cdot 5^5} - 4 \cdot \frac{1}{5 \cdot 239^5}\right) - \left(16 \cdot \frac{1}{7 \cdot 5^7} - 4 \cdot \frac{1}{7 \cdot 239^7}\right) + \cdots \quad (7.4)$$

Which can also be written in the following form:

$$\pi = \frac{1}{1}\left(\frac{16 \cdot 5}{(5^2)^1} - \frac{4 \cdot 239}{(239^2)^1}\right) - \frac{1}{3}\left(\frac{16 \cdot 5}{(5^2)^2} - \frac{4 \cdot 239}{(239^2)^2}\right) + \frac{1}{5}\left(\frac{16 \cdot 5}{(5^2)^3} - \frac{4 \cdot 239}{(239^2)^3}\right) - \frac{1}{7}\left(\frac{16 \cdot 5}{(5^2)^4} - \frac{4 \cdot 239}{(239^2)^4}\right) + \cdots \quad (7.5)$$

If multi-byte addition, subtraction, and division operations can be implemented, it becomes feasible to compute π to a large number of digits.

7.2 Pi Calculation Algorithm

This algorithm computes π digit-by-digit using decimal arrays. Each array element stores a single decimal digit.

For example: - Indexes `[0-3]` store the integer part. - Indexes `[4+]` store the fractional part.

Arrays used:

- `PI[]`: Stores the value converging to π.
- `T1[]`: Computed as $(16 \times 5)/(5 \times 5)^n$
- `T2[]`: Computed as $(4 \times 239)/(239 \times 239)^n$
- `T3[]`: Computed as $(-1)^{n+1} \cdot \frac{1}{2n-1} \cdot (\texttt{T1} - \texttt{T2})$

Computation Procedure:

1. $n = 0$: Compute `T1[]` and `T2[]`. Derive `T3[]` from `T1[]` and `T2[]`. Initialize `PI[]` to zero.
2. $n = 1$: Compute `T1[]` and `T2[]`. Compute `T3[]` and add to `PI[]`.
3. $n = 2$: Compute `T1[]` and `T2[]`. Compute `T3[]` and subtract from `PI[]`.
4. $n = 3$: Compute `T1[]` and `T2[]`. Compute `T3[]` and add to `PI[]`.
5. $n = 4$: Compute `T1[]` and `T2[]`. Compute `T3[]`. If `T3[]` is zero, convergence is achieved.

Example Calculation (Up to Four Decimal Places):

n	T1[]	T2[]	T3[]	PI[]
0	0080.0000	0956.0000	0000.0000	0000.0000
1	0003.2000	0000.0167	0003.1833 (+)	0003.1833
2	0000.1280	0000.0000	0000.0426 (−)	0003.1407
3	0000.0051	0000.0000	0000.0010 (+)	0003.1417
4	0000.0002	0000.0000	0000.0000 (−)	0003.1417

Table 7.1: Values of T1[], T2[], T3[], and PI[] for Each n

7.3 RAM Data Assignment

We assign memory data to RAM in the MCS-4 system for the computation of π.

The array RAMCH[Bank][SRC] stores four primary arrays: `T1[]`, `T2[]`, `T3[]`, and `PI[]`. The available RAM size is:

$$8 \text{ banks} \times 4 \text{ chips} \times 4 \text{ registers} \times 16 \text{ characters} = 2048 \text{ characters}$$

Since there are four arrays, each is allocated 512 elements. This defines the precision to 500 digits of π. The arrays are assigned to RAM banks as follows:

- `PI[512]`: Bank 0 (SRC=0x00–0xFF), Bank 1 (SRC=0x00–0xFF)
- `T1[512]`: Bank 2 (SRC=0x00–0xFF), Bank 3 (SRC=0x00–0xFF)
- `T2[512]`: Bank 4 (SRC=0x00–0xFF), Bank 5 (SRC=0x00–0xFF)
- `T3[512]`: Bank 6 (SRC=0x00–0xFF), Bank 7 (SRC=0x00–0xFF)

7.4 Source Program for Pi Computation

The source code for the π computation program is written in assembler for the ADS4004 tool. It is located at:

`SOFTWARE/ADS4004/pi4004.src`

The program can be assembled and simulated on a PC. Use memory dumps to verify behavior during execution.

7.5 Printing the Result with 141-PF Printer

To execute the π computation program, set SW8 to ON on the DE10-Lite board and reset the RISC-V 141-PF calculator program. The total computation time at a clock speed of 750 KHz (4004 CPU) is approximately 17 minutes.

The final result—500 digits of π—is printed by the 141-PF printer. The following is an example display from the serial terminal:

```
===== pi4004 ===== [CUI]
Loading...Done
     1415926535
     8979323846
     2643383279
     5028841971
```

```
6939937510
5820974944
5923078164
0628620899
..........
0921861173
8193261179
3105118548
0744623799
6274956735
1885752724
8912279381
8301194912
```

Chapter 8

Conclusion

The author's journey with microcomputers began in 1975 with the acquisition of the TK-80, an 8-bit single-board computer based on the 8080A. Although I have been involved with microcomputers for nearly 50 years, my first in-depth study of the MCS-4 system, including the 4004 CPU, was only about a decade ago. As for the Busicom 141-PF calculator, I had seen it exhibited but had never personally operated one.

In this project, I built a system to run the binary version of the 141-PF calculator program available online, incorporating RTL design of the 4004. Initially, I had no idea how the calculator's keys were supposed to be used. It was only after completing the entire system—from the RISC-V user interface to the MCS-4 logic—that I witnessed the 141-PF program finally come to life. That moment was truly moving. It revealed the calculator's surprisingly practical functionality and performance.

Engaging with the legacy left by our predecessors and rediscovering the profound wisdom and engineering embedded in these early systems has been an immensely valuable learning experience.

Bibliography

[1] 嶋正利 (Masatoshi Shima). **マイクロコンピュータの誕生＜わが青春の** *4004* **＞** *(The Birth of the Microcomputer: My Youth with the 4004).* 岩波書店 (Iwanami Shoten), 1987.

[2] Intel Corporation. *MCS-4 Micro Computer Set Users Manual*, rev.4 edition, Feb., 1973. `http://www.intel.com/Assets/PDF/Manual/msc4.pdf`.

[3] Intel Corporation. *MCS-4 Assembly Language Programming Manual*, preliminary edition edition, Dec., 1973. `http://www.nj7p.org/Manuals/PDFs/Intel/MCS-4_ALPM_Dec73.pdf`.

[4] Munetomo Maruyama. A risc-v subsystem mmrisc-1, risc-v rv32imafc core for mcu. `https://github.com/munetomo-maruyama/mmRISC-1`.