ปีการศึกษา 2531

SINGLE BOARD WITH 8031 CPU

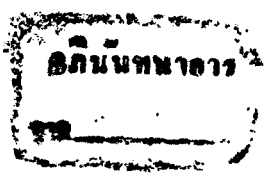
โดย

พีระศักดิ์ กลิ่นสุนทร

พีรายุ ศิคานนท์

อาจารย์ที่ปรึกษา

อ.ชาษฐา แช่ตั้ง



ปริญญานิพนธ์ ปีการศึกษา 2531

ภาควิชาอีเลคทรอนิค

คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง SINGLE BOARD WITH 8031 CPU

ผู้จัดทำ

พีระศักดิ์ เลิ่นสุนทร

พีรายุ ชิตานนท์

.....อาจารย์ที่ปรึกษา

(อ.ขนิษฐา แซ่ตั้ง)

SINGLE BOARD WITH 8031 CPU

พีระศักดิ์ กลิ่นสุนทร
พีรายุ ชินานนท์
ขนิษฐา แซ่ตั้ง อาจารย์ที่ปรึกษา
ปีการศึกษา 2531

บทคัดย่อ

งาน PROJECT ชิ้นนี้ เป็นการสร้าง SINGLE BOARD โดย MICRO CHIP ตระกูล MCS-51 เบอร์ 8031 หรือ 8051 มาออกแบบประกอบเข้ากับหน่วยความจำชนิด EPROM และ RAM (6264) และต่อกับ KEY BOARD โดยใช้หลักการ SCAN และมี DIODE 7 SECMENT เป็นตัวแสดงผล คือใช้ MICRO CHIP เบอร์นี้ทำตัวเป็น CPU ของวงจร และทดลองป้อนโปรแกรมขั้นพื้นฐานต่าง ๆ เพื่อทดสอบการทำงาน ซึ่งถ้าหากป้อนโปรแกรม MONITOR จะสามารถทำงานได้หลายด้าน และใช้ควบคุมวงจรอีเลคทรอนิคต่าง ๆ ด้วย

SINGLE BOARD WITH 8031 CPU

PEERASAK KLINSONTORN.

PEERAYU CHITANON.

KANITTA SAIRTUNG. ADVISER

1988

ABSTRACT

This Project is the construction of single board by Microchip MCS-51 No. 8031. It is designed to combine with EPROM and RAM (6264) memory. This network can work with keyboard by using scan method with 7 secments of diode to display.. This means that the microchip acts like CPU of the network. We had put basic programs to test operating. If we design the advance momitor program, the network can orerate many purposes and also control electronic circuits.

สารบัญ

บทนำ

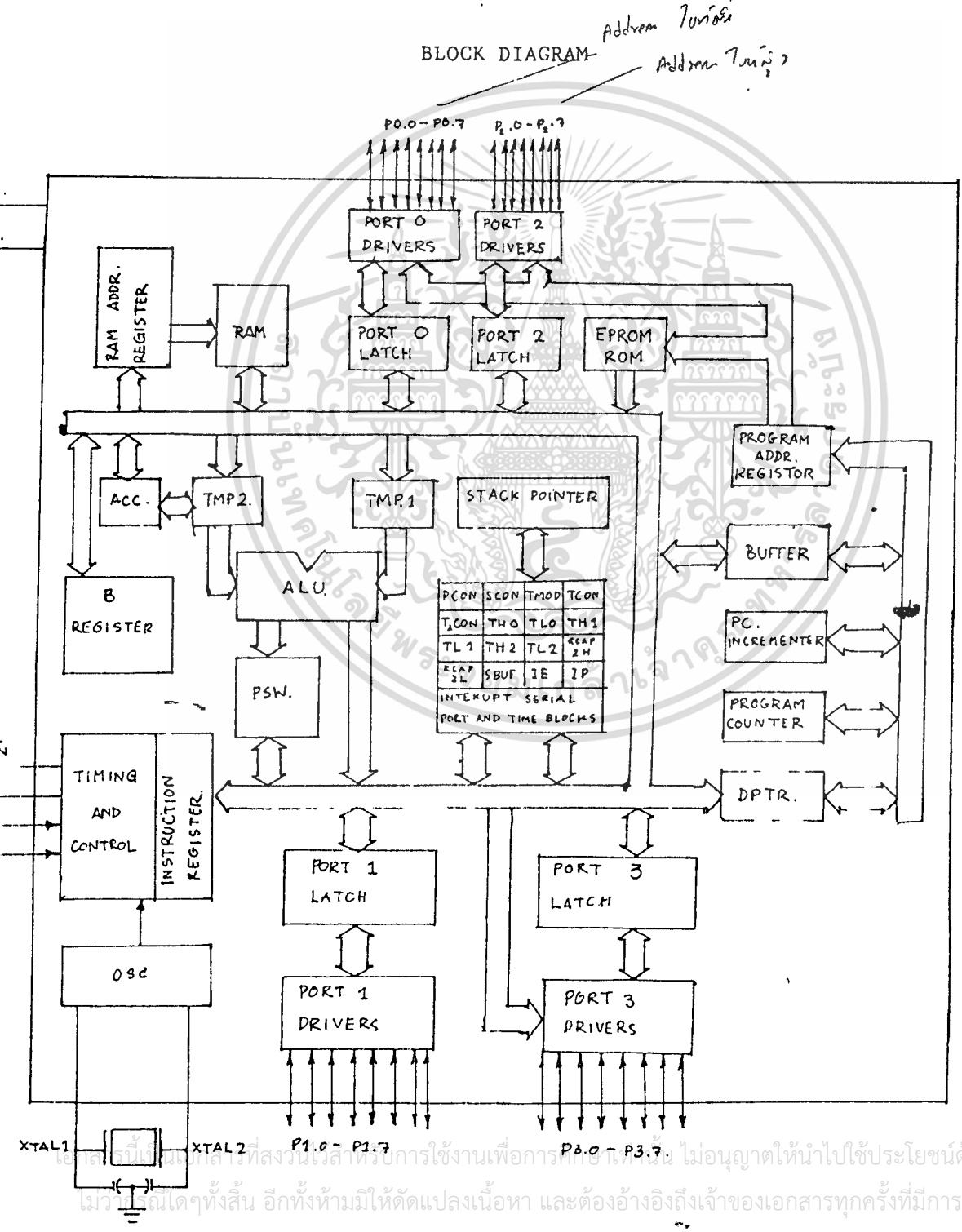
ในการศึกษาวิชาการค้านไมโครคอมพิวเตอร์ในเชิงวิศวกรรมนั้น SINGLE BOARD MICRO COMPUTER เป็นเครื่องมือสำคัญที่จะช่วยให้การศึกษาไมโครเป็นไปอย่างมีประสิทธิภาพ สามารถเรียนรู้และเข้าใจได้รวดเร็วขึ้น แต่ชิงเกิ้ลบอร์ดฯ ที่ใช้กันในปัจจุบันมักจะเป็นของที่ผลิตจากต่างประเทศ ทั้ง ๆ ที่สามารถที่จะพัฒนาขึ้นมาได้ ดังนั้น ในการทำปริญญานิพนธ์ในครั้งนี้ จึงได้ออกแบบชิงเกิ้ลบอร์ดฯ ชุดนี้ขึ้นมา โดยต้องการให้เป็นชิงเกิ้ลบอร์ดฯ ที่มีราคาถูก ใช้งานง่าย และมีประสิทธิภาพสูง

ในการออกแบบ SINGLE BOARD ขึ้นนี้ เราได้ศึกษาชิงเกิ้ลบอร์ดฯ ที่ใช้กันอยู่ทั่วไปหลายรุ่น จากนั้นได้ทำการรวบรวมข้อมูล วิเคราะห์ข้อดีข้อเสีย เพื่อเป็นแนวทางในการพัฒนาและออกแบบ จนสามารถกล่าวได้อย่างมั่นใจว่าเป็นชิงเกิ้ลบอร์ดฯ ที่คุ้มค่าสมราคา

บทที่ 1

1.1 ส่วนประมวลผลกลาง 8031, 8051

เป็น CPU เบอร์ใหม่ ซึ่งผลิตออกมาได้ไม่นานส่วนใหญ่มักจะนำไปใช้งานเฉพาะทาง เนื่องจากมี PORT ถึง 4 PORT ภายในตัว ที่สามารถต่อไปใช้งานได้ทันที ซึ่ง PORT บาง PORT อาจมีลักษณะเป็นทั้ง INPUT PORT และ OUT PUT PORT ใน PORT เดียวกัน ตามข้อกำหนดในการสร้างของบริษัท



1.2 แนะนำเบื้องต้นเกี่ยวกับตระกูล MCS-51

ไมโครชิบตระกูล MCS-51 เป็นไมโครชิบที่แบ่งออกเป็นหลายตัวดังตาราง

PART	TECHNOLOGY	ON-CHIP PROGRAM MEMORY	ON-CHIP DATA MEMORY
8051AH	HMOS II	4K — ROM	128
8031AH	HMOS II	NONE	128
87511H	HMOS I	4K — EPROM	128
80C51	CHMOS	4K — ROM	128
80C31	CHMOS	NONE	128
8052	HMOS II	8K — ROM	256
8032	HMOS II	NONE	256

ต้นแบบ 8051 ถูกสร้างด้วยเทคโนโลยีของ HMOS 1 และ HMOS 2 ที่กำลังผลิตในปัจจุบัน คือ 8051 AH เมื่อกล่าวถึงตัวใหม่สุดของตระกูล คือ 8032 และ 8052 จะมี ON-SHIP MEMORY เพิ่มขึ้นมากกว่าเดิม และมี 16 BIT TIMER/COUNTER TIMER ตัวใหม่ สามารถใช้ได้ในรูปแบบของ TIMER COUNTER หรือใช้ในการทำงานของ BOUD RATES สำหรับ SERIAL PORT ซึ่งเราจะไม่พูดถึงรายละเอียดของ 8052 นี้

จุดเด่นของไมโครชิบตระกูล MCS-51 คือ

- 8 bit CPU
- การจ่ายไฟเลี้ยว และวงจร CLOCK
- มี I/O PORT 32 เส้น
- มี ADDRESS 64 K ว่างไว้สำหรับข้อมูลความจำภายนอก
- มี ADDRESS 64 K ว่างไว้สำหรับโปรแกรมความจำภายนอก
- มี 16 BIT TIMER/COUNTER 2 ฟังก์ชัน และสำหรับ 8052 หรือ 8032 จะมี 3 ฟังก์ชัน
- มีรูปแบบการ INTERUPT 5 แบบ

- มี DUPLEX SERIAL PORT สมบูรณ์
- สามารถใช้งานแบบ BOOLEAN

1.3 การทำงานเกี่ยวกับหน่วยความจำ

8051 มีหน่วยความจำแยกระหว่าง โปรแกรมความจำและโปรแกรมข้อมูล โปรแกรมหน่วยความจำสามารถเก็บได้ 64 K BYTES และ 4 K เก็บได้ในตัว CHIP หน่วยความจำเก็บข้อมูลสามารถเก็บได้ 64 K BYTES สำหรับภายนอกโดยใช้ RAM และสามารถรวม 128 BYTES ในตัว CHIP RAM ด้วย (256 BYTES สำหรับ 8052) ยังสามารถรวมกับฟังก์ชันพิเศษของรีจีสเตอร์ด้วย ดังข้อมูลนี้

Symbol	Name	Address
*ACC	Accumulator	0E0H
*B	B Register	0F0H
*PSW	Program Status Word	0D0H
SP	Stack Pointer	81H
DPTR	Data Pointer (consisting of DPH and DPL	83H 82H
*P0	Port 0	80H
*P1	Port 1	90H
*P2	Port 2	0A0H
*P3	Port 3	0B0H
*IP	Interrupt Priority Control	0B8H
*IE	Interrupt Enable Control	0A8H
TMOD	Timer/Counter Mode Control	89H
+*T2CON	Timer/Counter Control	88H
TCON	Timer/Counter 2 Control	0C8H
TH0	Timer/Counter 0 (high byte)	8CH
TL0	Timer/Counter 0 (low byte)	8AH
TH1	Timer/Counter 1 (high byte)	8DH
TL1	Timer/Counter 1 (low byte)	8BH
+TH2	Timer/Counter 2 (high byte)	0CDH
+TL2	Timer/Counter 2 (low byte)	0CCH
+RCAP2H	Timer/Counter 2 Capture Register (high byte)	0CBH
+RCAP2L	Timer Counter 2 Capture Register (low byte)	0CAH
*SCON	Serial Control	98H
SBUF	Serial Data	
PCON	Power Co	

ACCUMULATOR

ACC เป็น ACCUMNLATOR REGISTER มีลักษณะแบบง่าย ๆ ใช้สัญลักษณ์ A

B REGISTER

เป็น REGISTER ที่ใช้ระหว่างการคูณและหาร และในการทำงานในรูปแบบอื่นก็จะสามารถใช้งานได้เหมือนกับ SCVATCH PAD รีจีสเตอร์

PROGRAM STATUS WORD

เป็น PSW รีจีสเตอร์ บรรจุโปรแกรมสถานะต่าง ๆ ของข้อมูลดังรูป

Symbol	Position	Name and Significance	Symbol	Position	Name and Significance
CY	PSW 7	Carry flag.	OV	PSW.2	Overflow flag.
AC	PSW.6	Auxiliary Carry flag (For BCD operations)	—	PSW.1	(reserved)
			P	PSW.0	(Parity flag.) Set/cleared by hardware each instruction cycle to indicate an odd/even number of "one" bits in the accumulator, i.e., even parity.
F0	PSW 5	Flag 0 (Available to the user for general purposes.)			
RS1	PSW.4	Register bank Select control bits 1 & 0. Set/cleared by software to determine working register bank (see Note)	Note—		the contents of (RS1, RS0) enable the working register banks as follows:
RS0	PSW.3		(0 0	Bank 0	(00H-07H) 00H-07H
			(0 1	Bank 1	(08H-0FH) 08H-0FH
			(1 0	Bank 2	(10H-17H) 10H-17H
			(1 1	Bank 3	(18H-1FH) 18H-1FH

STACK POINTER

STACK POINTER REGISTOR มีความกว้าง 8 BIT จะเพิ่มก่อนข้อมูลที่เก็บไว้ในระหว่างคำสั่ง PUSH และ CALL STACK POINTER จะทำงานที่ 07H หลังจาก RESET ซึ่งแสดงว่า STACK จะเริ่มที่ตำแหน่ง 08H

DATA POINTER

DATA POINTER (DPTR) ประกอบด้วย BITE สูง (DPH) และ BITE ต่ำ (DPL) เป็นฟังก์ชันที่ใช้ HOLD 16 BIT ADDRESS หรือจะทำให้เป็น 8 BIT REGISTER อิสระ 2 ชุด ก็ได้

TIMER REGISTERS

คู่ REGISTER (THO,TLO), (TH 1,TC 1) และ (TH 2, TC 2) เป็นรีจีสเตอร์แบบนับ 16 BIT สำหรับ TIMER/COUNTER 0,1 และ2

CAPTURE REGISTERS

REGISTER (RCAP2H, RCAP2L) เป็น REGISTER สำหรับ TIMER 2 "CAPTURE MODE" ใน MODE นี้ใช้กับ 8052 ซึ่งจะไม่กล่าวถึงรายละเอียด

CONTROL REGISTER

เป็น REGISTER ฟังก์ชันพิเศษ IP, IE, TMOD, TCON, T2CON, SCON และ PCON ประกอบด้วย BIT ควบคุมและ BIT สถานะ สำหรับระบบการ INTERRUPT, TIMER/COUNTER และ SERIAL PORT

1.4 ระบบเวลาการทำงานของ CPU

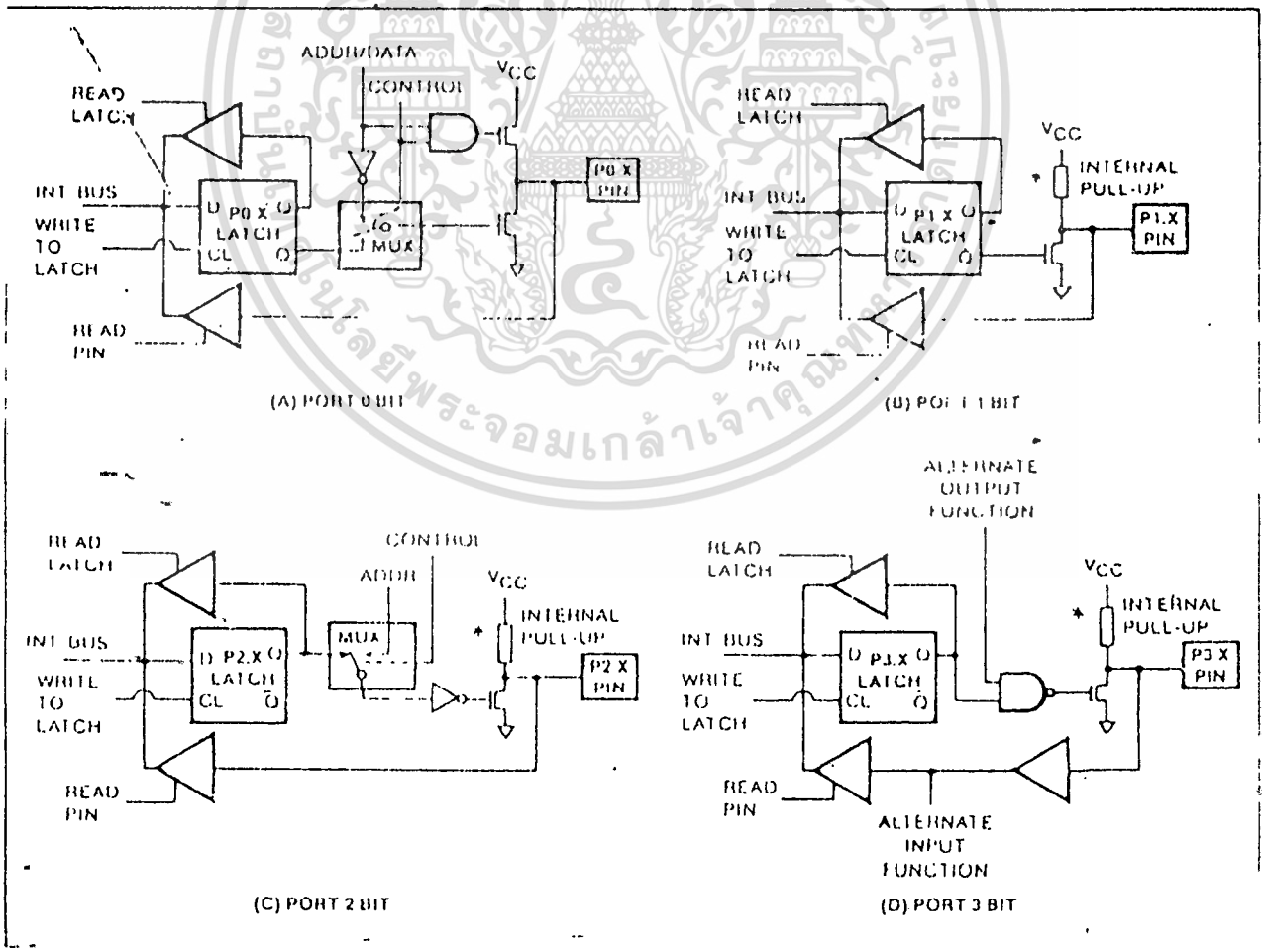
รูปแบบวงจรแบ่งออกเป็น 6 STATES (12 OSCILLATOR PERIODS) แต่ละ STATE แบ่งออกเป็นเฟสครั้งแรก ระหว่างที่ CLOCK ของเฟส 1 ACTIVE และเฟสครั้งที่ 2 ระหว่าง ที่ CLOCK ของ เฟส 2 ACTIVE สัญลักษณ์ S1P1 (STATE 1, PHASE 1) ถึง S6P2 (STATE 6, PHASE 2) แต่ละ PHASE จะให้เวลา 1 คาบเวลา และแต่ละ STATE

จะใช้เวลา 2 คาบเวลา การทำงานเกี่ยวกับตัวเลจและเลขคณิต เกิดขึ้นในระหว่าง PHASE 1 และการส่งผ่านข้อมูลภายในของ REGISTER จะเกิดขึ้นระหว่าง PHASE 2

ในรูปภาพแสดง FETCH/EXECUTE TIMING เปรียบเทียบกับสถานะภายในและเฟส สัญญาณ ALE (ADDRESS LATCH ENGBLE) โดยปกติจะทำงาน 2 รอบ ในแต่ละ MACHINE CYCLE ครั้งหนึ่งระหว่าง S1P2 และ S2P1 และอีกครั้งระหว่าง S4P2 และ S5P1

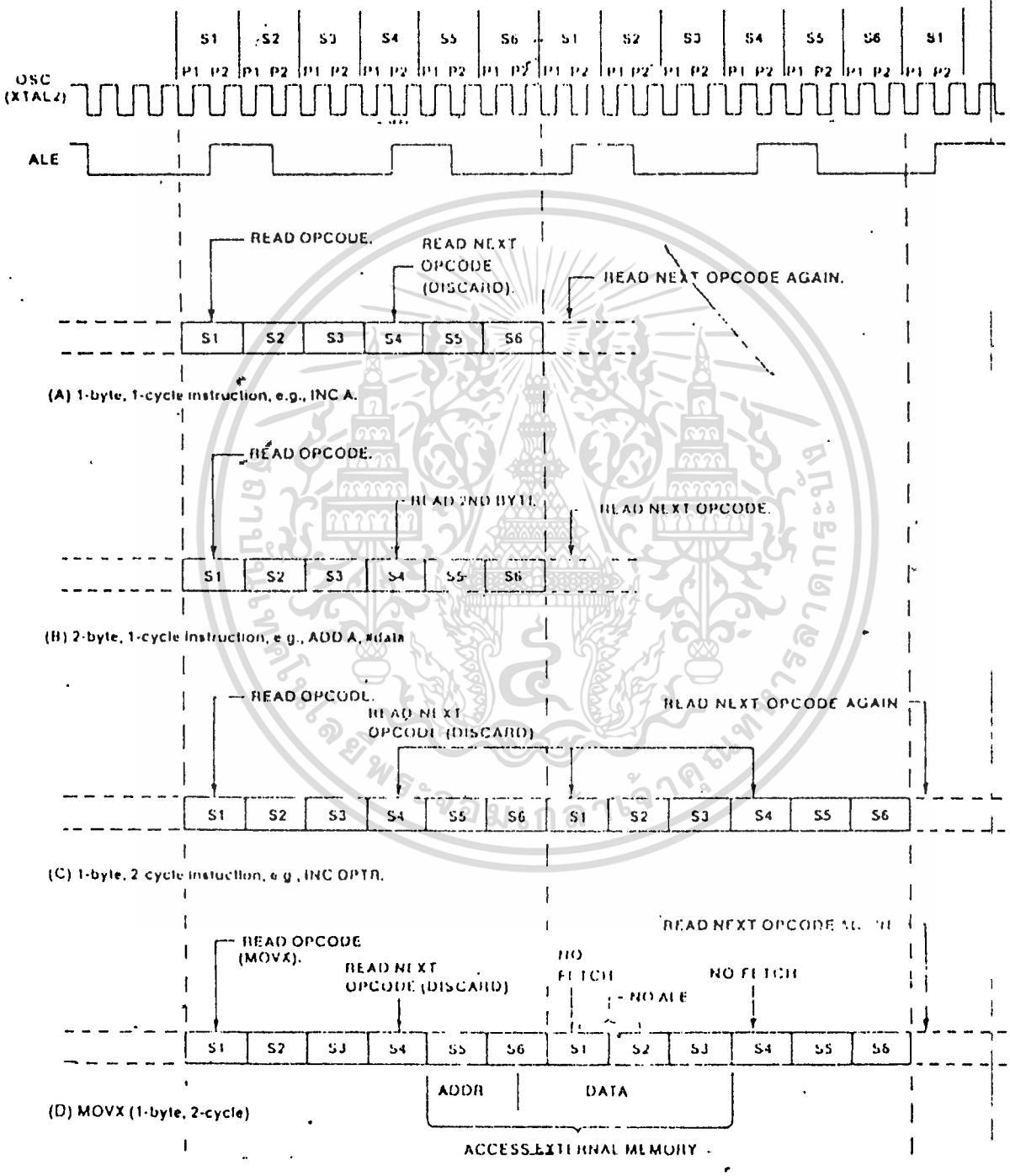
1.5 ส่วนประกอบของ PORT และการทำงาน

PORT ทั้ง 4 ของ 8051 เป็น PORT แบบ 2 ทิศทาง แต่ละ PROT ประกอบด้วย LATCH (REGISTER ฟังก์ชันพิเศษ ของ P0-P3) และมี INPUT DRIVER กับ INPUT BUFFER ด้วย

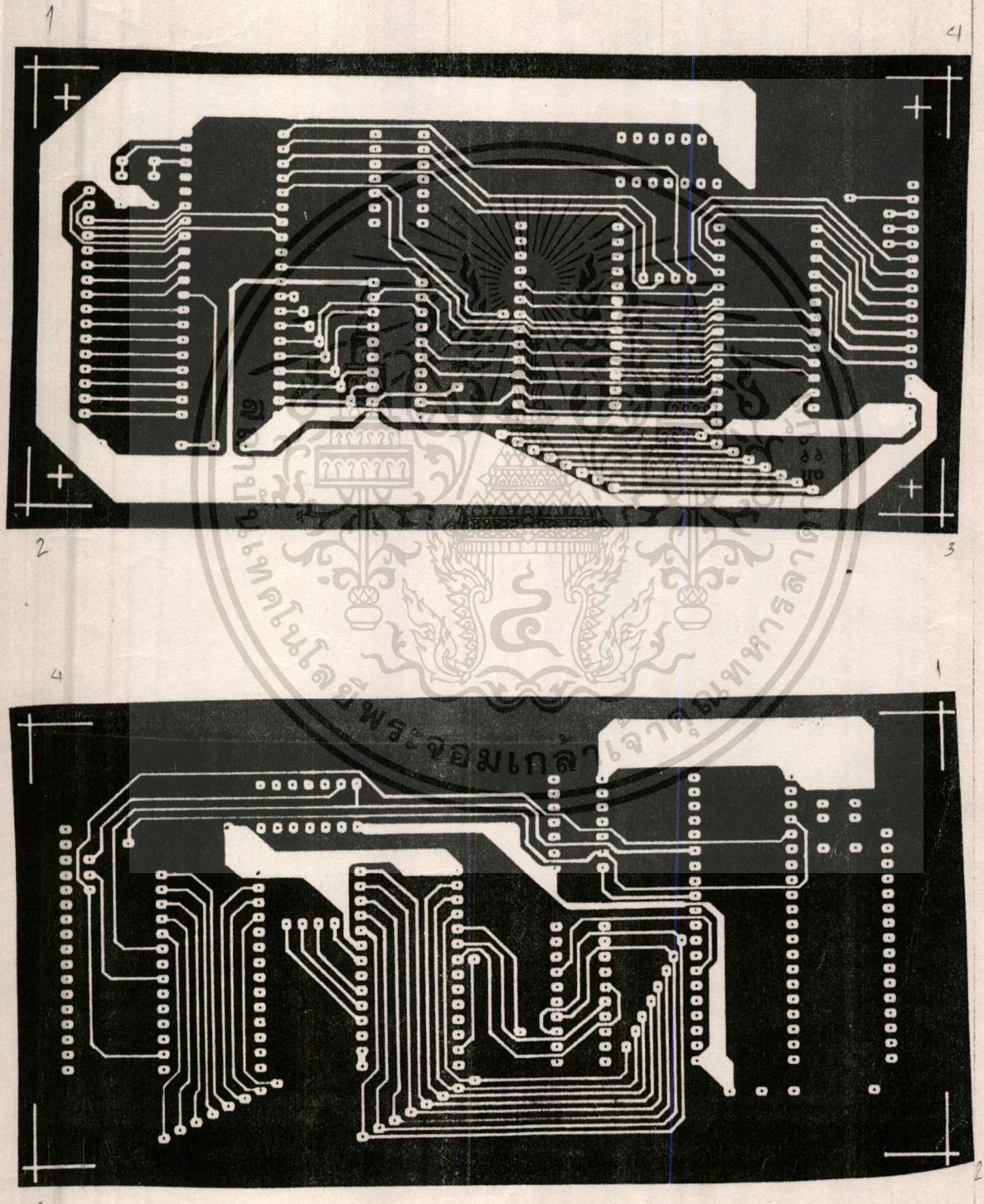


OUTPUT DRIVER ของ PORT 0 และ 2 และ INPUT BUFFERS ของ PORT 0 จะใช้สำหรับการเข้าถึงหน่วยความจำภายนอก

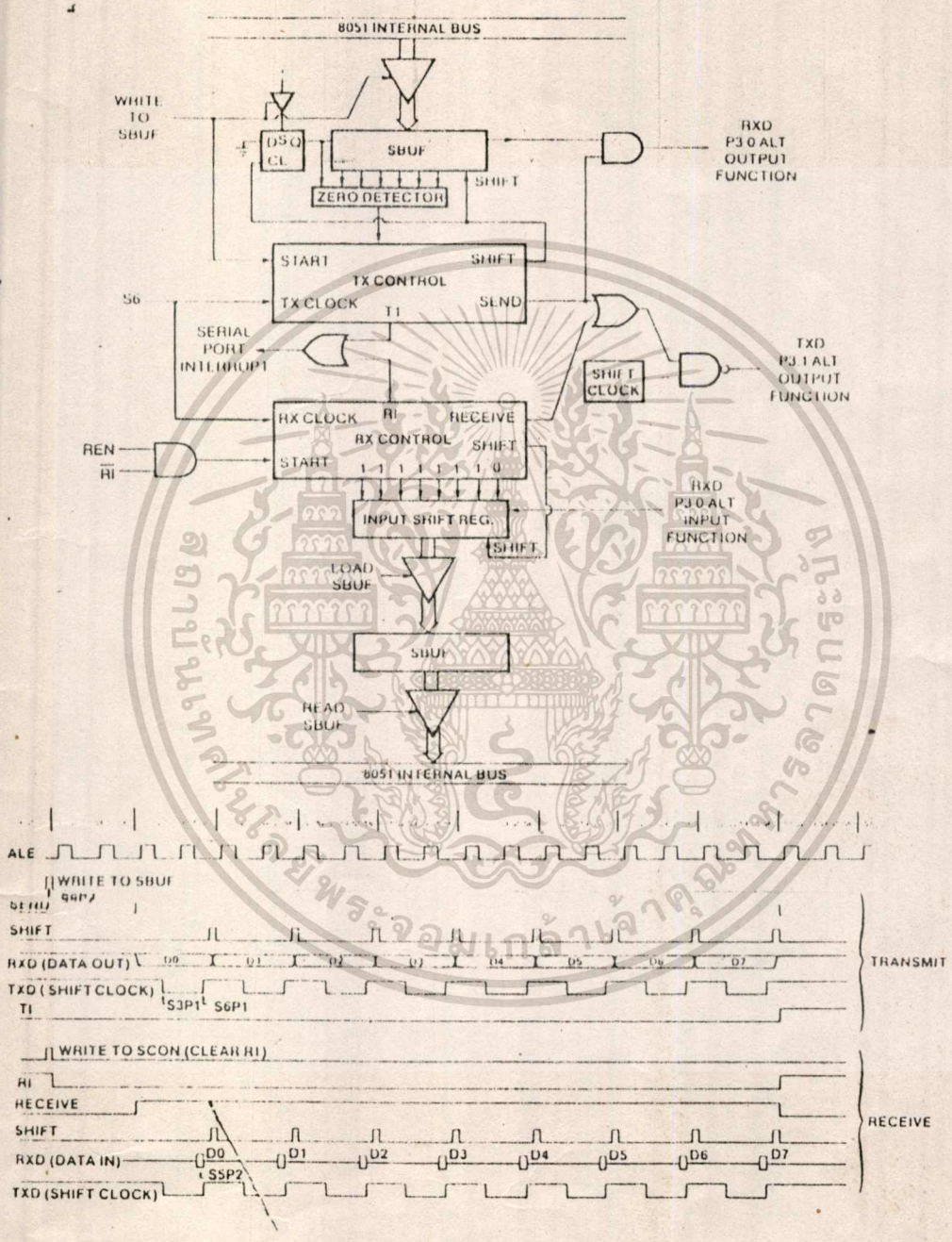
ภาพแสดง FETCH/EXECUTE TIMING



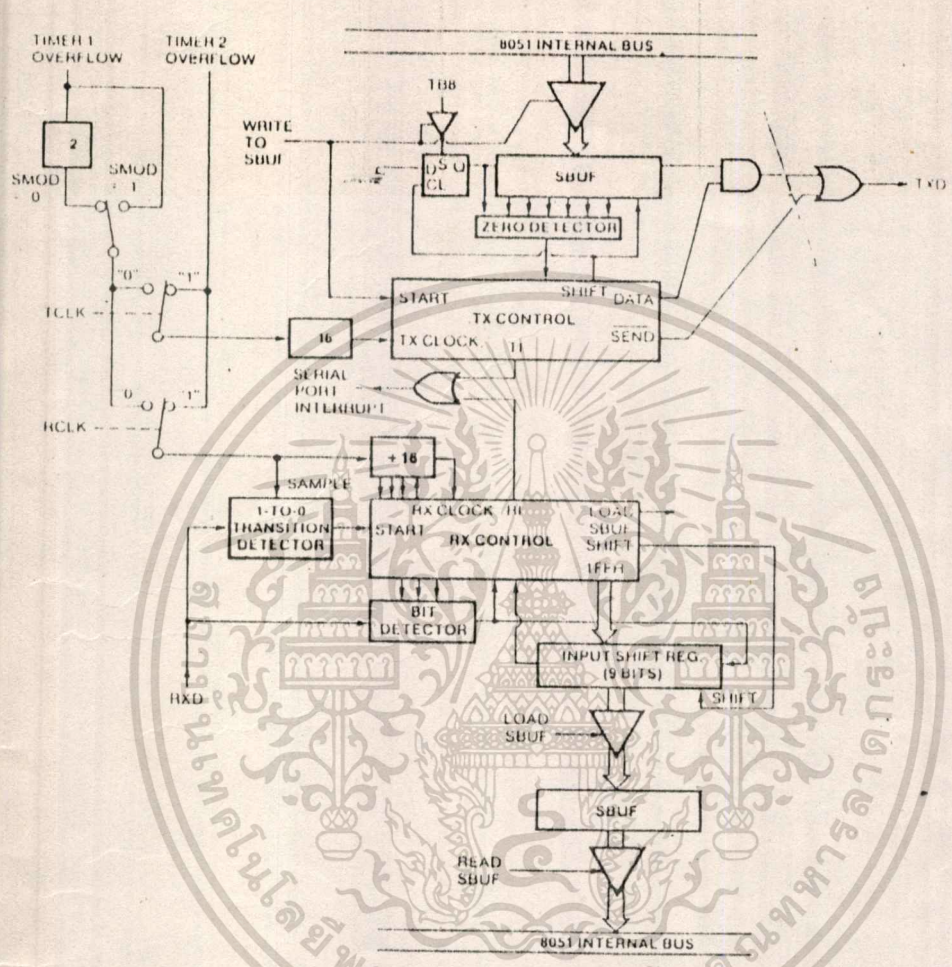
ลาย PRINT แบบ TWO HOLD ของส่วน 8031 และ MEMORY (ไม่รวม KEY BOARD และ DISPLAY)

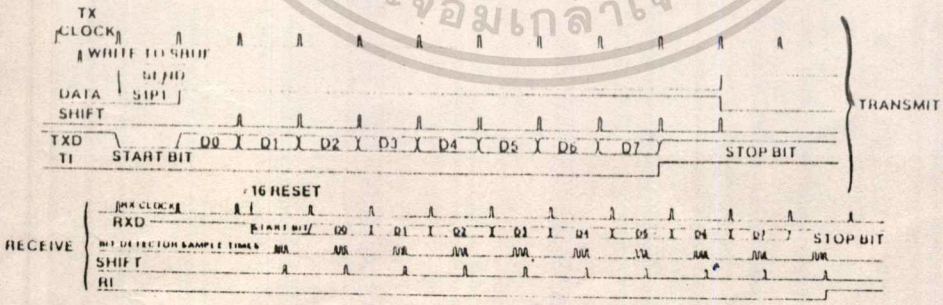


SERLAL PORT MODE 0

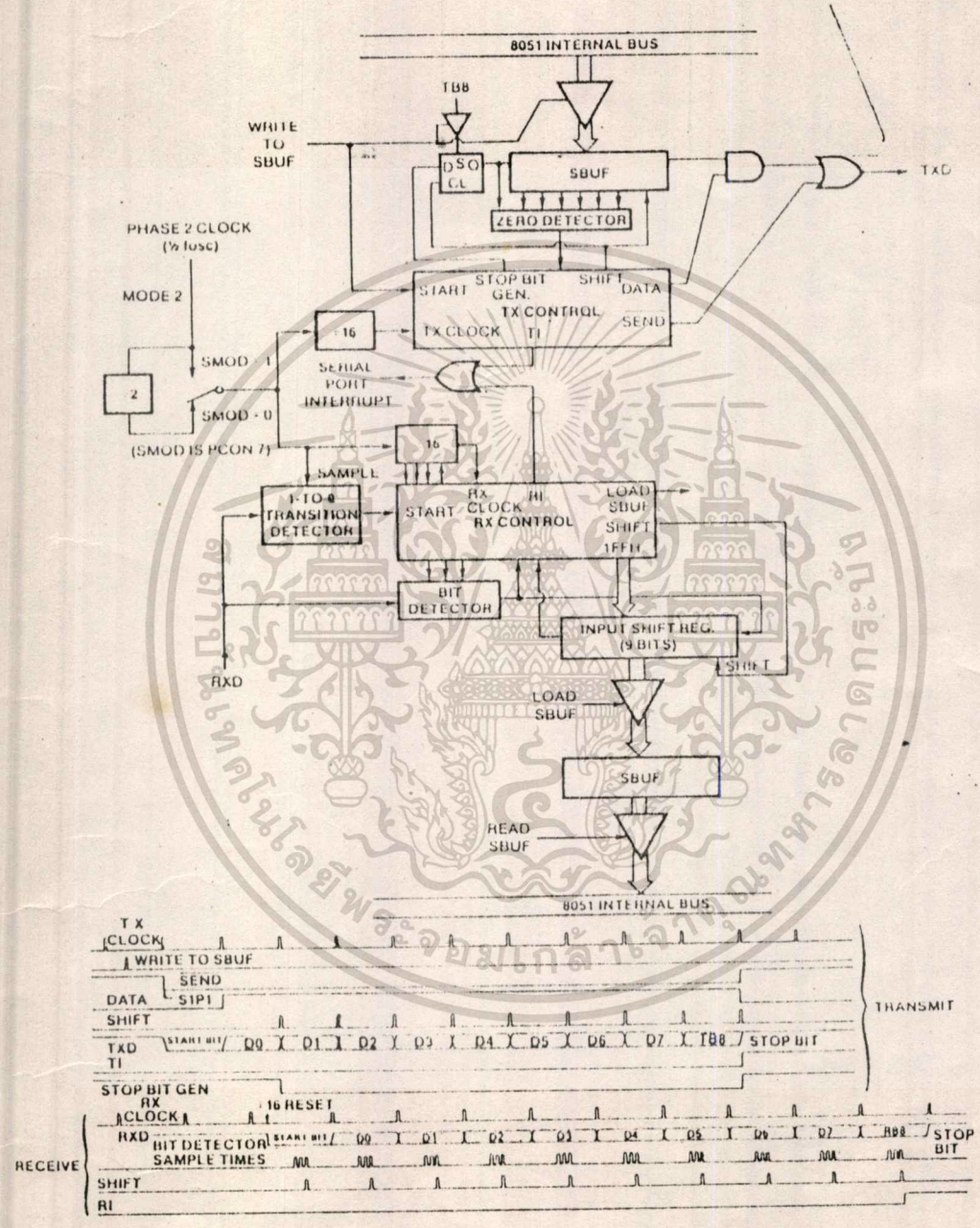


SERLAL PORT MODE 1





SERLAL PORT MODE 2



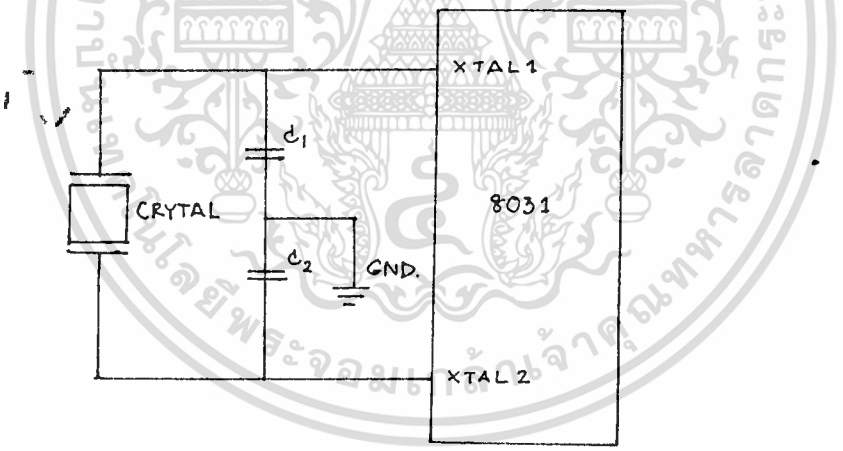
บทที่ 2

ในการสร้าง SINGLE BOARD โดยใช้ 8031 CPU เราจะต้องเพิ่มส่วนของหน่วยความจำเข้าไปเป็นชนิดของ RAM เนื่องจากไม่มีหน่วยความจำสำหรับเก็บข้อมูลและเราจะเพิ่มหน่วยความจำชนิด EPROM เพื่อเก็บข้อมูลเกี่ยวกับ PROGRAM MORNITOR ใน 8051 CPU โครงสร้างจะมีส่วนที่สามารถเก็บข้อมูลอยู่แล้วแต่เราจะไม่สนใจเนื่องจากสามารถเก็บข้อมูลได้น้อย ในการสร้าง SINGLE BOARD ครั้งนี้เราใช้ 2764 EPROM เป็นตัวเก็บ PROGRAM MORNITOR และ 6264 RAM เก็บข้อมูลทั่วไป มี 74 LS 373 เป็นตัว LATCH ADDRESS และใช้ 74 LS 138 เป็นตัว EDCODER ที่จะเลือกใช้หน่วยความจำว่าจะใช้ตัวไหน

2.1 ขาต่าง ๆ ที่สำคัญของ 8031 CPU

- XTAL 1 และ XTAL 2

  CPU จะทำงานได้ เราจะต้องป้อนสัญญาณ CLOCK ให้สามารถใช้ได้ทั้ง CRYSTALS หรือ CERAMIC RESONATORS โดยต่อดังรูป



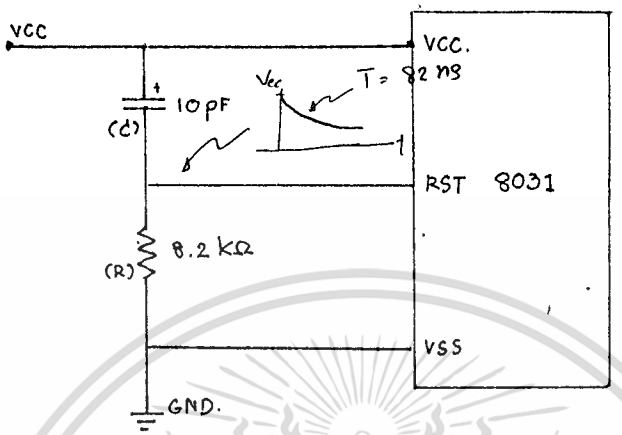
CERAMIC RESONATOR และ CRYSTALS ความถี่ 4-6 MH_z

โดยมี C_1 , C_2 = 30 pF $\pm$ 10 pF FOR CRYSTALS

= 40 pF $\pm$ 10 pF FOR CERAMIC RESONATORS

- RESET (RST)

เมื่อต้องการ RESET PROGRAM จะมีการต่อควบคู่กับ VSS และ VCC ดังรูป



บางครั้งเราอาจจะต่อให้สามารถ รีเซทด้วยมือได้ด้วยเมื่อต้องการ

- INT 0 และ INT 1

เป็นขาที่ใช้ในการ INTERUPT ต่าง ๆ มักใช้เกี่ยวกับ KEYS BOARD

- T0 , TI (TIMER 0, TIMER 1)

- P0, P1, P2

เป็น PORT ที่สามารถนำไปต่อใช้งานเข้ากับอุปกรณ์อิเลคทรอนิคได้ทันที

- $\overline{WR}$

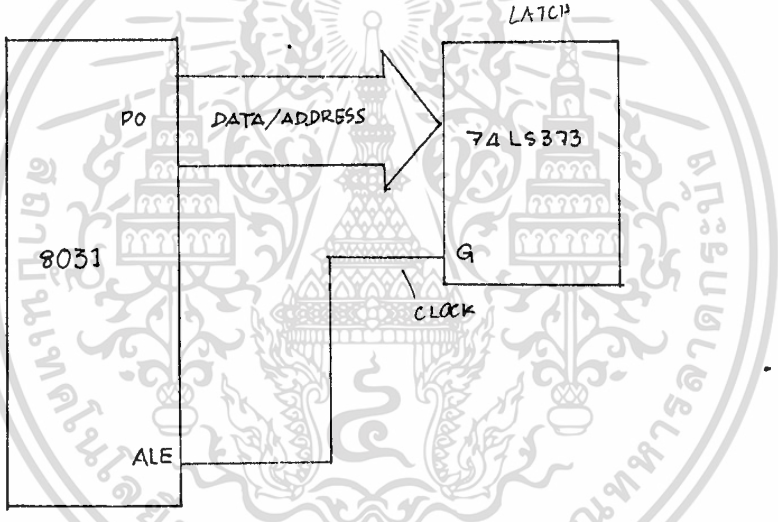
เมื่อสัญญาณมีสถานะต่ำ (0) จะทำให้ขา WRITE เกิด ACTIVE แสดงว่า CPU ต้องการเขียนข้อมูล ใช้ควบคุมขา $\overline{WE}$ ของ RAM

- $\overline{RD}$

เมื่อสัญญาณมีสถานะต่ำ (0) จะทำให้ขา READ เกิด ACTIVE แสดงว่า CPU ต้องการอ่านข้อมูล มักต่อควบคู่กับ $\overline{PSEN}$ เพื่อไปควบคุมขา $\overline{OE}$ (OUTPUT ENABLE) ของ EPROM และ RAM ให้จ่ายข้อมูลมายัง CPU

- ALE

เป็นขา OUTPUT จะป้อนสัญญาณเป็นจังหวะเข้ากับ PORT ที่จะป้อนค่า ADDRESS และ DATA สลับกันไป เราจึงนำขา ALE นี้ไปต่อเป็น CLOCK ให้กับ 74 LS 373 เพื่อให้ทำงานเป็นจังหวะ ๆ จึงทำหน้าที่เป็น LATCH ADDRESS จะทำงานเฉพาะจังหวะที่ CPU ส่งสัญญาณ ADDRESS เท่านั้น



ขั้วต่าง ๆ ของ 6264 RAM

- A0-A12

เป็น BUS ADDRESS รับข้อมูลเกี่ยวกับ ADDRESS ที่เก็บข้อมูลไว้

- D0-07

เป็น BUS DATA จะปล่อยข้อมูลได้ 8 BIT

- $\overline{CS1}$, CS 2

เป็น CHIP SELECT สั่งให้ RAM ทำงานโดย $\overline{CS1}$ จะต้องได้รับสัญญาณต่ำ (0) และ CS 2 จะต้องได้รับสัญญาณสูง (1) จึงจะ ACTIVE $\overline{CS1}$ และ CS2 นี้จะต่อมาจากตัว DECODER อีกที

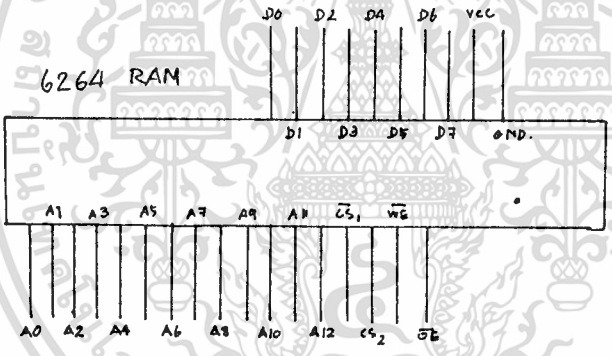
- $\overline{WE}$

ขา WRITE ENABLE จะ ACTIVE ที่สัญญาณต่ำ (0) ตัว 6264 RAM จะรับข้อมูลเข้าไปเก็บไว้

- $\overline{OE}$

ขา OUTPUT ENABLE จะ ACTIVE ที่สัญญาณต่ำ (0) ตัว 6264 RAM จะส่งข้อมูลออก

ภาพแสดงขาต่าง ๆ ของ 6264 RAM



ขาต่าง ๆ ของ 2764 EPROM

- A0-A12

เป็น BUS ADDRESS รับข้อมูลเกี่ยวกับ ADDRESS ที่เก็บข้อมูลไว้

- D0-D7

เป็น BUS DATA เก็บข้อมูลได้ 8 BIT

- $\overline{CE}$

ขา CHIP ENABLE เป็นขาที่เลือกให้ 2764 EPROM ทำงาน จะ ACTIVE ที่สัญญาณต่ำ (0) ต่อมาจากตัว DECODER อีกที

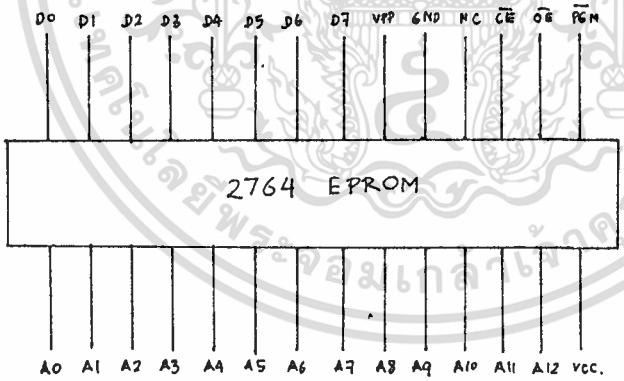
- $\overline{OE}$

ขา OUTPUT ENABLE จะ ACTIVE ที่สัญญาณต่ำ (0) บอกให้ 2764 EPROM จ่ายข้อมูลออกทาง DATA BUS

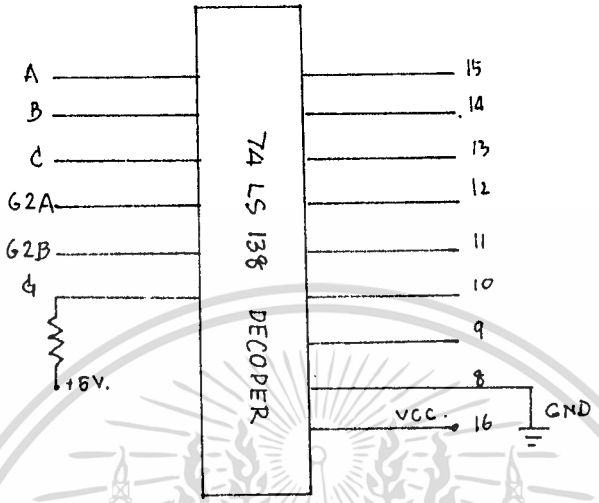
- $\overline{PGM}$

เนื่องจาก EPROM สามารถ PROGRAM ได้เพียงครั้งเดียว เวลาลบก็จะต้องใช้เครื่องลบพิเศษเกี่ยวกับแม่เหล็กไฟฟ้า ขานี้จะใช้ในขณะ PROGRAM ข้อมูลในครั้งแรก และจะไม่มีการ PROGRAM อีกในขณะใช้งาน ขานี้จะ ACTIVE ที่สถานะต่ำ (0) เมื่อใช้งานเราจะต่อให้อยู่ในสถานะสูง (1) เพื่อไม่ให้มีการเขียนข้อมูลเข้าไปใน EPROM อีก

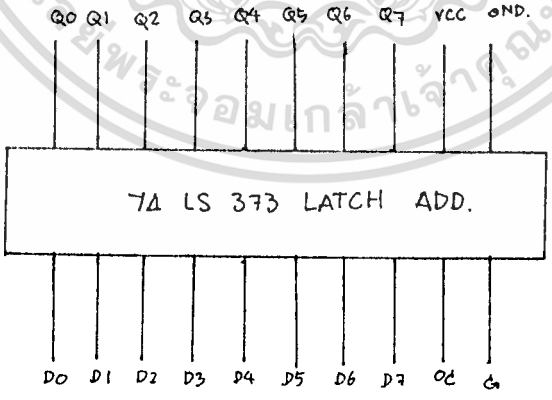
ภาพแสดงขาของ 2764 EPROM



ภาพแสดง 74 LS 138 DECODER



ภาพแสดง 74 LS 373 LATCH ADDRESS



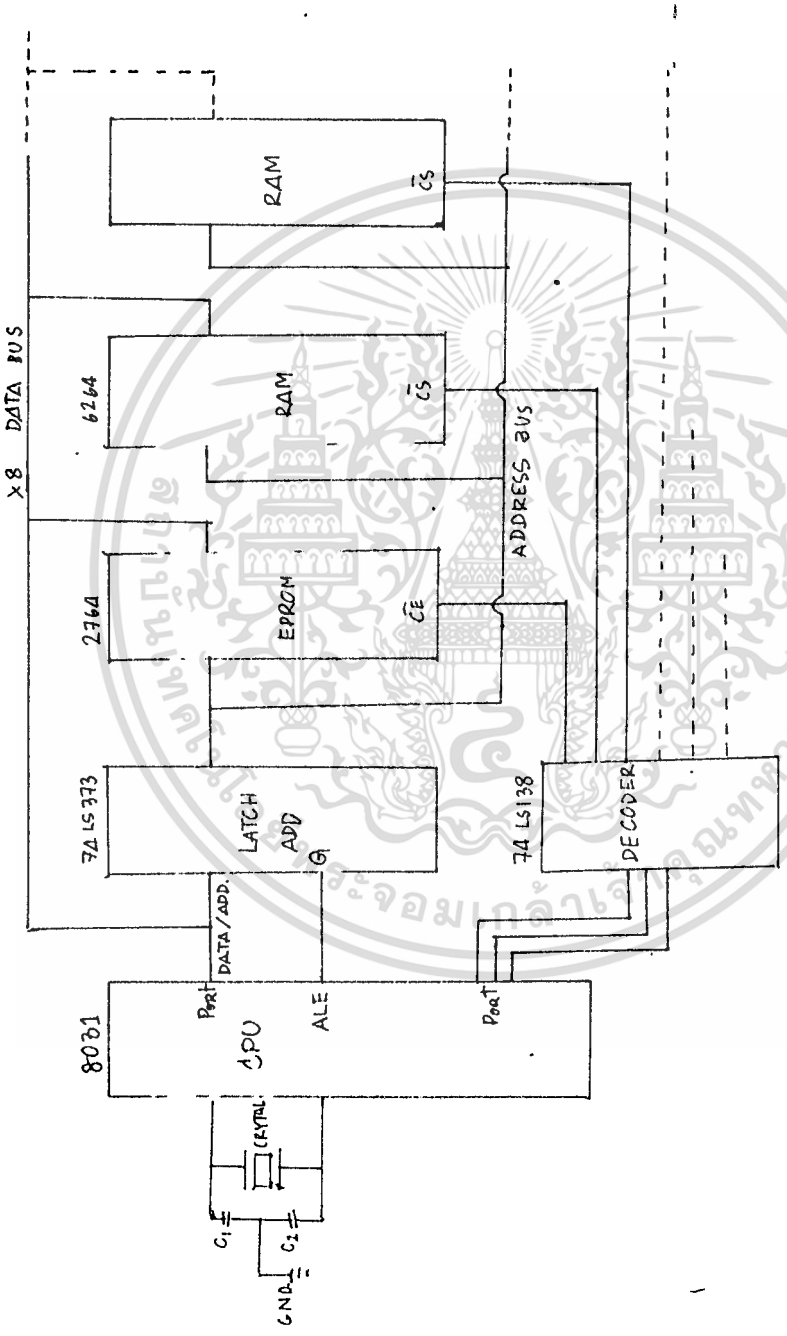
2.2 ระบบการทำงานของ SINGLE BOARD

ตัว SINGLE CHIP 8031 ด้วยสัญญาณ OSCILLATOR ที่สร้างจาก CRYTAL โดยเพียงแต่ต่อเข้าไปรวมกับ CAPACITOR เนื่องจากมีวงจรภายในตัวช่วยสร้างสัญญาณด้วย PORT 0 ซึ่งมีสัญญาณ ADDRESS สลับกับสัญญาณ DATA ผ่าน เมื่อเราต้องการนำไปต่อเข้ากับ EPROM ในด้าน ADDRESS จึงต้องผ่าน LATCH เพื่อคัดสัญญาณเฉพาะ ADDRESS เข้าไป EPROM โดย LATCH จะทำงานตามสัญญาณ CLOCK ที่ต่อมาจาก ALE (ADDRESS LATCH ENABLE) ของ 8031 และต่อขยายหน่วยความจำออกไปอีกโดยใช้ RAM การทำงานของ RAM และ EPROM จะทำโดยสัญญาณ CHIP SELECT ซึ่ง DECODE ด้วย 74LS 138 เลือกจาก ADDRESS BITE สูง ของ PORT 2 โปรแกรม MONITOR จะถูกป้อนเข้าไปยัง EPROM 8031 จะเป็น CPU สั่งคำสั่งต่าง ๆ ส่งสัญญาณ $\overline{RD}$ และ $\overline{WE}$ (สัญญาณ $\overline{OE}$-OUTPUT ENABLE ของ EPROM และ RAM เกิดจากสัญญาณ $\overline{PSEN}$ AND กับสัญญาณ $\overline{RD}$ เนื่องจากต้องการรวม DATA ARRIA และ PROGRAM ARRIA ให้ง่ายต่อการโปรแกรมการใช้งาน) เมื่อต้องการอ่านและเขียนข้อมูล ภายในวงจรจะติดต่อกับภายนอกโดย KEYBOARD ซึ่งใช้หลักการ SCAN และ DISPLAY เป็น 7 SECMENT DIODE เมื่อ KEY ข้อมูลจะเข้าไปเก็บยัง 6264 RAM CPU จะดึงโปรแกรมจาก 2764 DPROM มาจัดการทับข้อมูลที่ KEY เข้าไป แล้วส่งออกมา DISPLAY หรือเก็บไว้ใน RAM อีกครั้งหนึ่งก็ได้แล้วแต่คำสั่ง

2.3 การ SCAN KEYBOARD และ 7 SECMENT DISPLAY

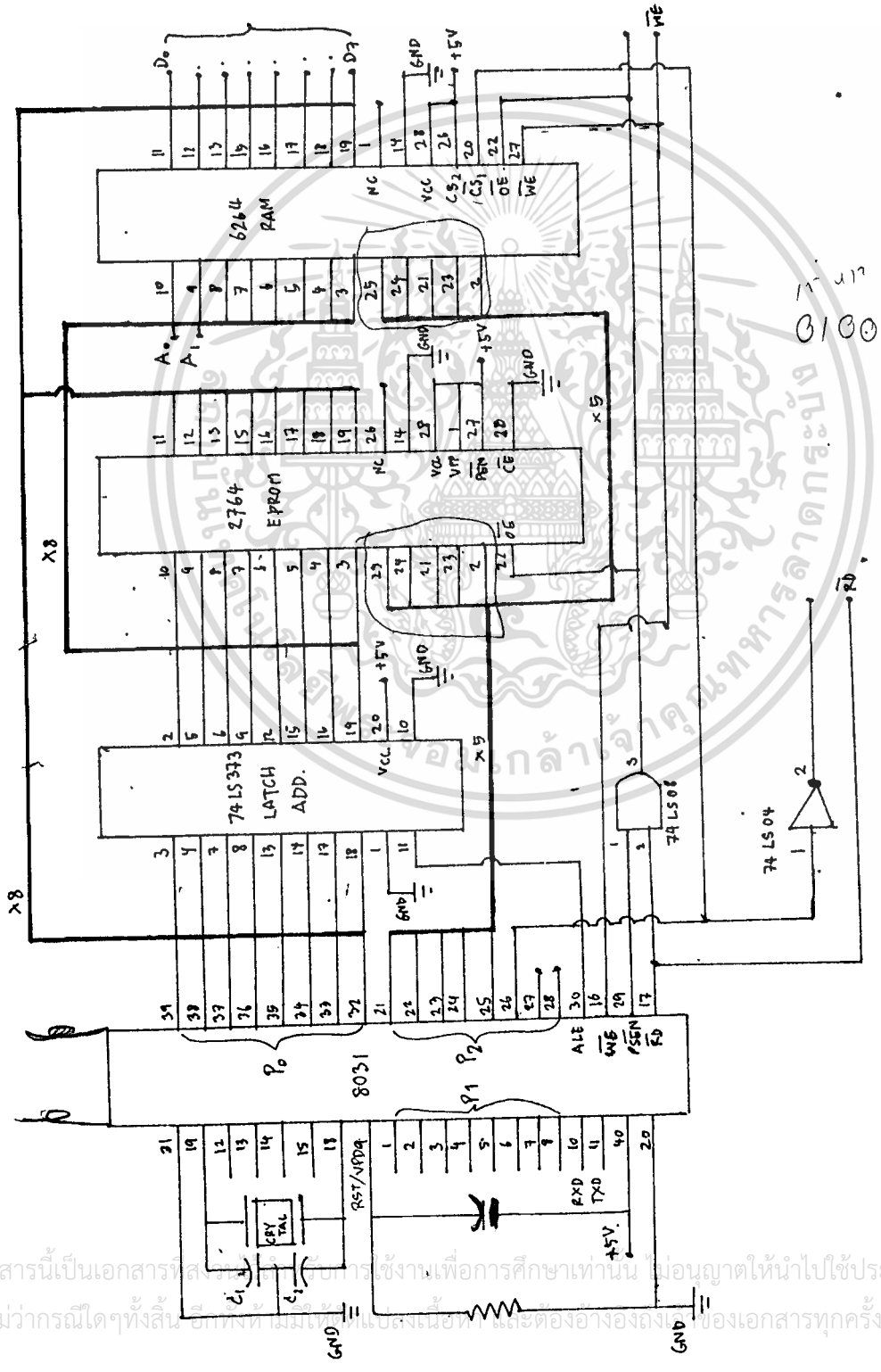
จากแผนภาพ โปรแกรมจะมีคำสั่งเปลี่ยนค่าที่ 15,14,13 ของ 74LS 145 อย่างรวดเร็ว เพื่อให้เกิดการ DECODE ออกมาไล่ขาอย่างรวดเร็ว (ขา 1,2,3,4,.... ตามลำดับ) เป็นลักษณะการ SCAN ในสถานะ LOW เพื่อตรวจจับว่า ที่ COLUM ใดถูกกด ในทาง ROLL ที่ต่อจาก PORT ของ 8255 จะถูก PULL UP ไว้ด้วยค่า HIGH เมื่อกด KEY สายใน ROLL ที่ถูกกดจะช็อตลง GND ทำให้ตรวจจับได้ว่า ถูกกดใน ROLLไหน จากค่าที่ SCAN พบว่าเป็นการกด KEY ตัวใดแล้วจะเข้าโปรแกรมส่งไป MOVE DATA ใน TABLE ที่ถูกโปรแกรมเอาไว้ก่อนแล้ว ให้มาออกที่ PORT B ผ่าน BUFFER ไปยัง 7 SECMENT DISPLAY

ภาพวงจรรวมการต่อ 8031 CPU และหน่วยความจำ

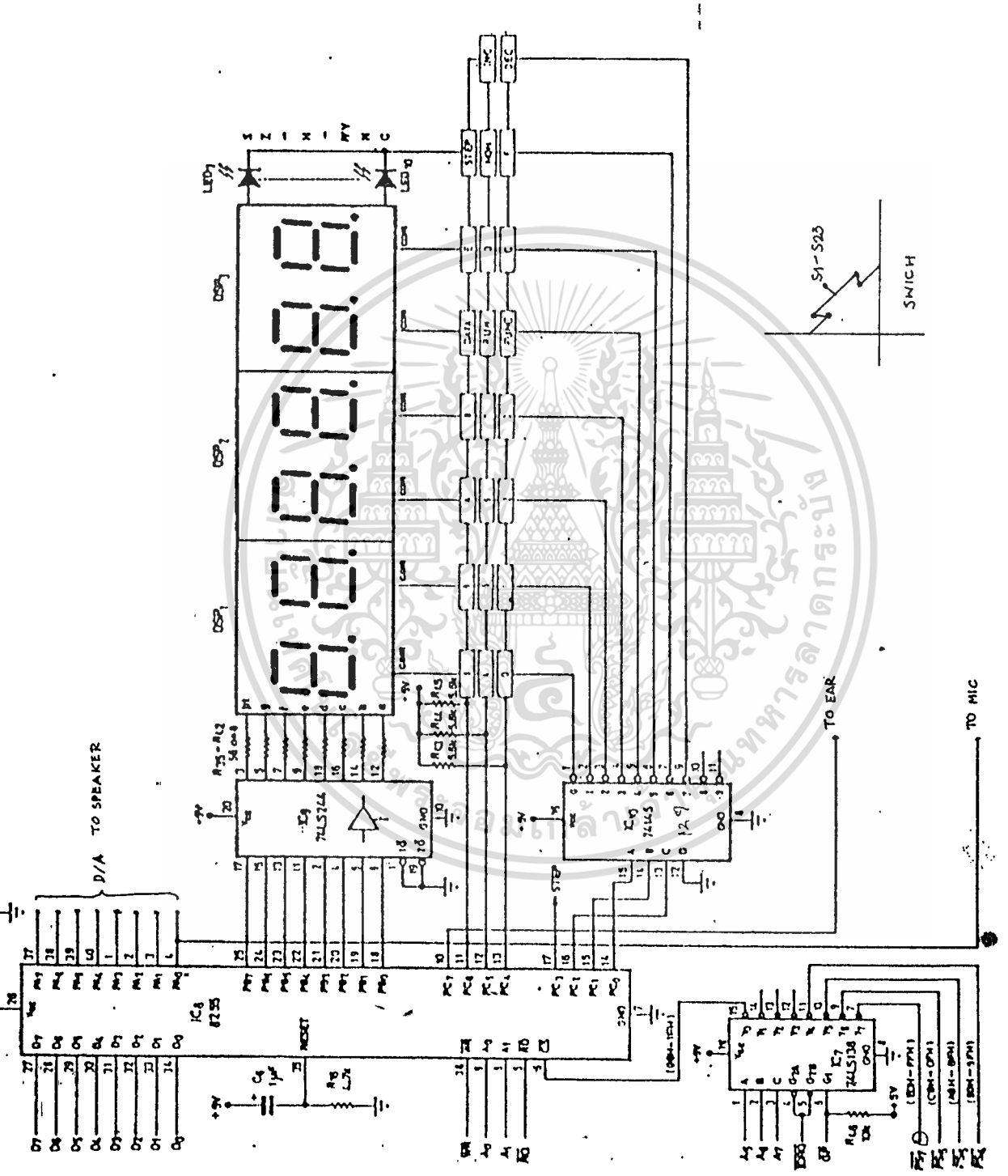


สำหรับ SINGLE BOARD ที่สร้างนี้จะสร้างเพื่อแสดงให้เห็นว่าใช้งานได้เท่านั้น จึงใช้หน่วยความจำ RAM เพียงตัวเดียว ซึ่งที่จริงอาจขยายหน่วยความจำออกไปได้อีกมากมาย โดยอาศัย DECODER เลือกดังรูปที่ผ่านมาแล้ว

ภาพวงจรโดยละเอียดการต่อขาต่าง ๆ ของ 8031 CPU กับหน่วยความจำ 2764 EPROM , 6264 RAM , LATCH ADDRESS 74LS 373 และ 74LS 138 DECOPER

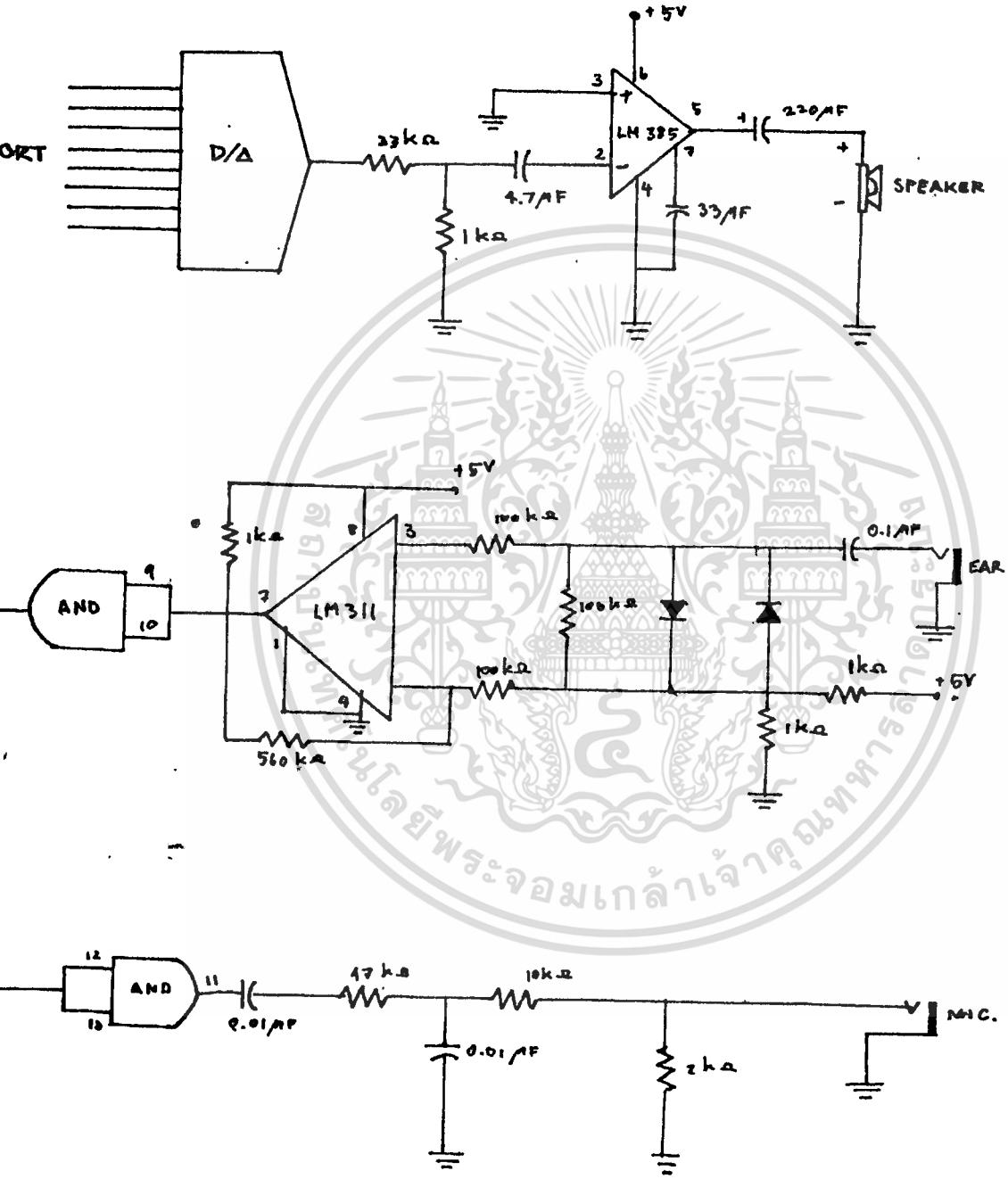


ภาพแสดงวงจรการต่อ KEYBOARD และ DISPLAY โดยใช้ 8255 เป็นตัวขยาย PORT



TAPE INTERFACING + SOUND

ใน SINGLE BOARD จะมีวงจรเพื่อแสดงเสียงเมื่อกด KEYBOARD และ TAPE INTERFACING ด้วยก็ได้ แต่ในงานชิ้นนี้ไม่จำเป็น



2.4 PROGRAM ย่อยของการทำงานของ FUNCTION KEY ต่าง ๆ

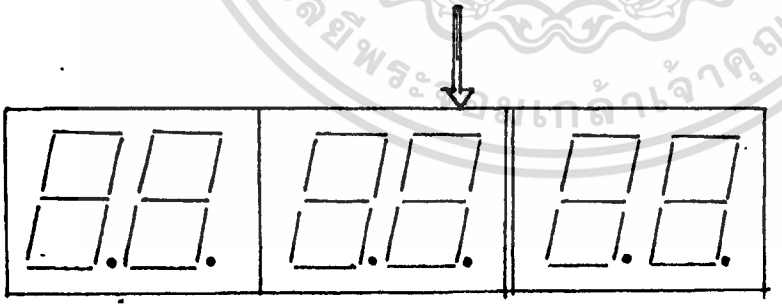
FUNCTIONKEY ต่าง ๆ จะมีโปรแกรมส่วนย่อยเฉพาะตัว แต่บางส่วนก็จะอ้างถึงโปรแกรมหลักด้วย

ตัวอย่าง PROGRAM ของ FUNCTION KEY เช่น

- **INC KEY** (INCREMENT KEY)

```
S3      :  MOV     DPTR. # DISP 2
S0S2    :  MOVX    A,@DPTR
           INC     A
           C JNE   A, # 110; S3S1
           MOV     A, # 0
S3S1    :  MOVX    @DPTR, A
           INC     DPTR
           JZ      S0S2
           AJMP    DATA
```

เริ่มการนำค่า DISP 2 (DISPLAY 2)



SP. No 5 4 3 2 1 0

มาใส่ DPTR (ชี้ตำแหน่ง) แล้ว นำค่าจาก DPTR มาใส่ในตัวแปร A ต่อจากนั้น เพิ่มค่า A ขึ้นไปอีก 1 (ต้องพิจารณากรณีที่ต้องทดหลักเพิ่มด้วย) ใช้คำ (COMPAIR IMMEDIATE AND JUMP IF NOT EQUAL)

คือเปรียบเทียบ A กับ H10 ถ้าไม่เท่ากันจะ JUMP ไปที่ S3S1 (กรณีไม่มีตัวทด)

นำค่า A ที่ได้ส่งเข้าไปแสดงผล ใน DPTR ถ้าหาก A กับ H10 มีค่าเท่ากันจะไม่มีการ JUMP โปรแกรมจะทำงานต่อลงมายังบรรทัดถัดลงมา คือ ป้อนค่า 0 ลงไปที่ A แล้วนำค่าไปแสดงที่ DPTR ต่อจากนั้น เพิ่มค่า DPTR ซึ่งก็คือค่าของ DISP (DISPLAY) นั่นเอง ทำให้ขยับขึ้นไปอีก 1 หลัก ต่อจากนั้นจะถูก JUMP ไปที่ SOS2 จากคำสั่ง JZ (JUMP IF ACC. IS ZERO) เพราะ ถ้าเกิดการทดหลักแรกจะต้องเป็น 0 และตัวทดจะเพิ่มค่าขึ้นอีก 1 ตามโปรแกรมที่วน LOOP การทำงานอีกรอบ

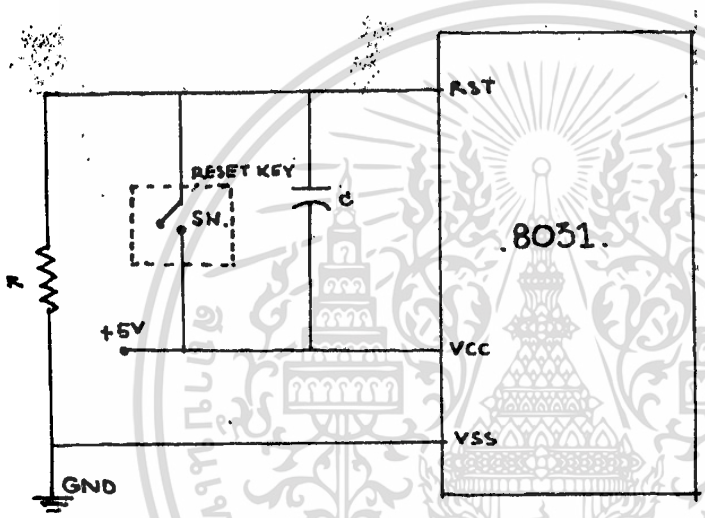
```
- DEC. KEY  (DECREMENT KEY)
  DEC       :  MOV     DPTR, #DISPLAY 2
  DEC 2     :  MOVX    A, @DPTR
               DEC     A
               CINE    A, # HFF, DEC 3
               MOV     A, # H0F
  DEC 3     :  MOVX    @DPTR, A
               INC     DPTR
               CINE    A, #H0F, DEC 2
               AJMP    DATA
```

นำค่า ใน DIS P 2 มาใส่ใน DPTR แล้วนำค่าใน DPTR มาใส่ใน A จากนั้นลดค่า A (ในกรณีที่ตัวท้ายเป็น 0 เมื่อลดค่าลง 1 แล้ว จะต้องดึงให้ตัวทดลดลง แล้วตัวท้ายจะมีค่าเป็น F) ใช้คำสั่ง CJNE ถ้า A ไม่เท่ากับ HFF จะ JUMP ไปยัง DEC 3 แล้วส่งค่า A ลงไปยัง DPTR เพื่อไป DISPLAY ถ้า A เท่ากับ HFF จะไม่มีการ JUMP จะบันทึกค่า H0F ลงเป็นหลักแรก และเพิ่มค่า DPTR ซึ่งก็คือเพิ่มหลัก DISP

เปรียบเทียบค่า A กับ HOF ซึ่งจะ JUMP ไปที่ DEC 2 เพราะถ้ามีการลดตัวหดหลักแรก จะต้องเท่ากับ HOF จาก DEC 2 ตัวหดจะลดค่าลง 1 ตามโปรแกรมที่วน LOOP การทำงานอีกรอบ

- RESET KEY

เป็น FUNCTION ทาง HARD WARE การกด RESET KEY คือการ SHORT ขา RST กับ +5V.



บทที่ 3

โปรแกรมหลัก (MAIN PROGRAM)

เมื่อเริ่มต้น โปรแกรมเครื่องจะทำการแสกนคีย์บอร์ด เพื่อตรวจสอบว่ามีคีย์ใดที่ถูกกดบ้าง ถ้าไม่มีคีย์ใดถูกกดเลย เครื่องจะย้อนกลับไปทำงานแบบเดิมซ้ำไปเรื่อย ๆ แต่หากว่ามีคีย์หนึ่งคีย์ใด ถูกกด เครื่องจะตรวจสอบว่าคีย์ที่ถูกกดนั้น คือคีย์ใดและจะทำงานตามหน้าที่ของคีย์ที่ถูกกด คือ

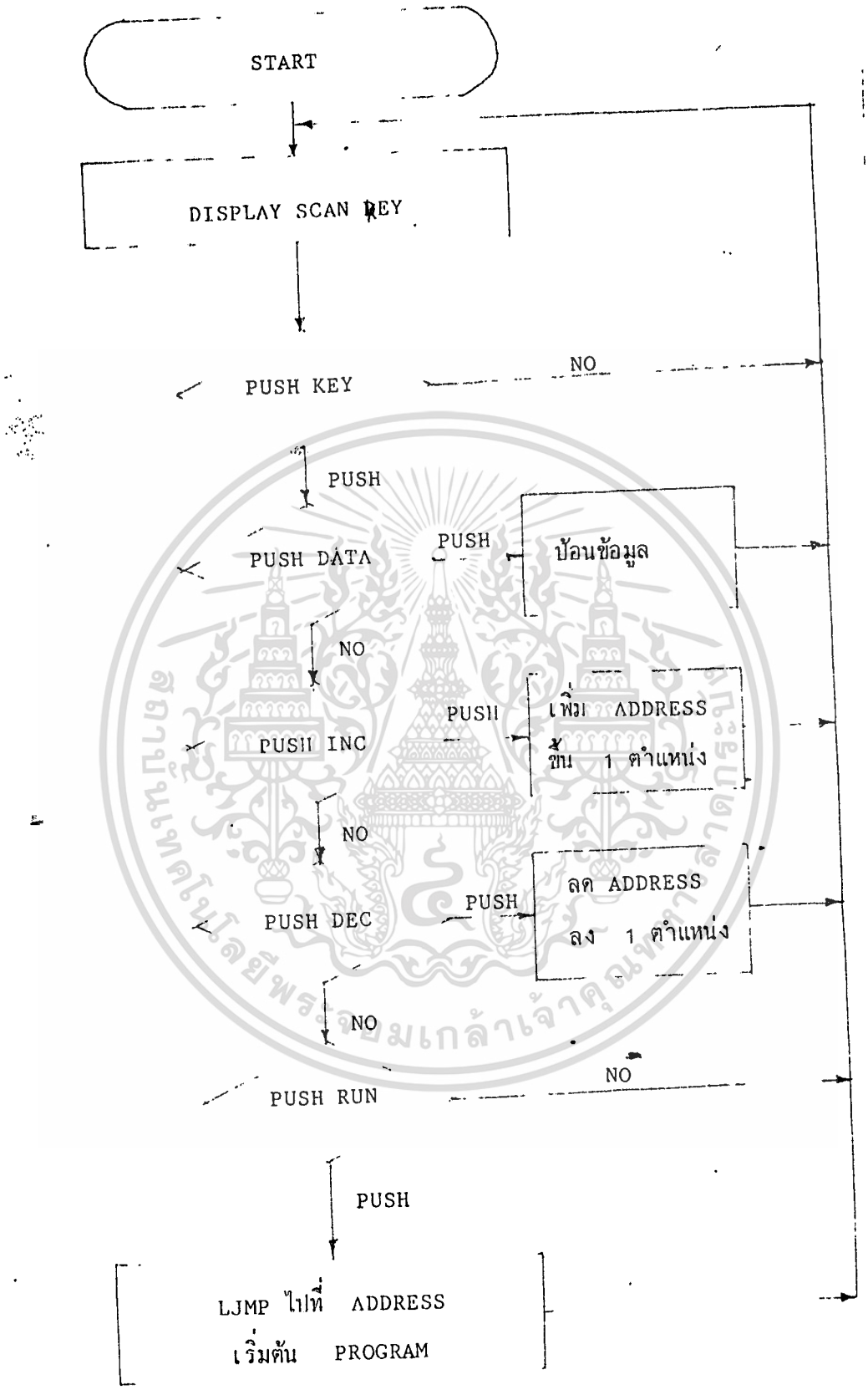
ถ้าคีย์ DATA ถูกกด ก็จะทำให้เครื่องพร้อมที่จะรับข้อมูลเพื่อนำไปเก็บไว้ใน MEMORY หลังจากนั้น เครื่องจะย้อนกลับไปทำการ SCAN KEY ใหม่ เพื่อตรวจสอบว่าเราป้อนข้อมูลใดเข้าไปและจะนำข้อมูลนั้นไปเก็บไว้ใน MEMORY ต่อไป แต่ถ้าคีย์ DATA มิได้ถูกกด เครื่องจะทำการตรวจสอบว่าคีย์ใดถูกกดต่อไป

เมื่อคีย์ INC ถูกกด นั้น เครื่องจะทำการเพิ่ม ADDRESS ขึ้น 1 ตำแหน่ง แล้วย้อนกลับไปทำการ SCAN KEY ใหม่

เมื่อคีย์ DEC ถูกกด นั้น เครื่องจะทำการลด ADDRESS ลง 1 ตำแหน่ง แล้วย้อนกลับไปทำการ SCAN KEY ใหม่อีก โดยจะมีลักษณะการทำงานคล้ายกับคีย์ INC

เมื่อคีย์ RUN ถูกกด นั้น เครื่องจะทำ LJMP คือการกระโดดไปทำงานตามโปรแกรม ณ ADDRESS ที่เริ่มต้นของโปรแกรมนั้น

รูป 3.1 โฟลชาร์ทการทำงานของโปรแกรมมอนิเตอร์ขั้นพื้นฐาน



บทที่ 4

การทดลอง

ในการทดลองป้อนโปรแกรมลงใน EPROM มีการทดลองป้อนโปรแกรมอย่างง่าย ๆ เพื่อทดสอบการทำงานของ DISPLAY

ตัวอย่าง สมมุติต้องการทดลองโชว์ค่าเลข 3 หลัก 0

```
        MOV     DPTR, #TABLE        TABLE : DB    H3F
        MOV     A, # 3                      DB    H06
        MOVC    A, @A+DPTR                  DB    H5B
        MOV     DPTR, # H4001               DB    H4E
        MOVX    @DPTR, A                    DB    H66
        INC     DPTR                        DB    H6D
        MOV     A, # 0                      DB    H7D
        MOVX    @DPTR, A                    DB    H07
HERE :  SJMP    HERE                        DB    H7F
                                            DB    H6F
                                            DB    H77
                                            DB    H7C
                                            DB    H39
                                            DB    H5E
                                            DB    H79
                                            DB    H71
                                            DB    H00
                                            DB    H40
```

เริ่มจากนำ TABLE มาใส่ลงใน DPTR แล้วป้อนเลขลงใน A นำค่า DPTR รวมกับ A จะได้ตำแหน่งใน TABLE ที่ตรงกับค่าที่ป้อนให้ 7 SECMENT แล้วออกมาเป็นเลข 3 เก็บไว้ใน A

F2 DPTR
3 ADDRESS
4 COLUM
5 TABLE

นำค่า H4001(หมายถึง PORT B) ใส่เข้าไปที่ DPTR เพื่อให้ค่าที่จะออก ๆ ที่ PORT B จากนั้นป้อนค่า ออกไปที่ ADDRESS DPTR เราจะเห็นสัญญาณไปติดเป็นเลข 3 จากนั้นเพิ่มค่า DPTR แล้วใส่ค่า 0 ใน A แล้วนำมา OUT ที่ ADDRESS DPTR จะทำให้ไฟที่ติดดับไป

F4

ตัวอย่าง ทดลองป้อนโปรแกรมให้ COLUM ที่ 5 ติดเลข 5 COLUM ที่ 4 ติดเลข 4 , COLUM ที่ 3 ติดเลข 3 เรียงกันทีละ COLUM จนถึง COLUM 0 แล้ววนกลับไป COLUM 5 ใหม่

F1

```
ST Ø    :  MOV   RØ, #6          ทำ COLUM สุดท้ายก่อน
STA     :  MOV   A, RØ
           DEC   A                1000 XXXX XXXX 0010
           MOV   DPTR, #H4002     แสดงค่า PORT C
           MOVX  @DPTR, A         นำค่า COLUM ส่งออก PROT C
           MOV   DPTR, #TABLE     หาค่าตัวเลขจาก TABLE
           MOVC  A; @A + DPTR
           MOV   DPTR, #H4001     กำหนด PORT B เพื่อส่งตัวเลขออกที่ PORT B
           MOVX  @DPTR, A
           MOV   R1, #0           ช่วงโปรแกรมหน่วงเวลา
           MOV   R2, #0
DELA    :  DJNZ  R1, DELA
           DJNZ  R2, DELA
           MOV   A, #0            ทำให้ DISPLAY ดับทันที
           MOVX  @DPTR, A
           DJNZ  RO, STA
           AJMP  STØ
```

นำ 6 ใส่ในค่า RØ นำ RØ ใส่ใน A ลดค่า A เหลือ 5 (คือ COLUM ที่ 5) เอา H4002 (PORT C) ใส่ใน DPTR แล้วนำค่า A ใส่ใน ADDRESS DPTR จะได้ค่า COLUM ออกที่ PORT C นำ TABLE ใส่ใน DPTR ใช้ค่า ADDRESS A + DPTR จะได้ตัวเลขเก็บไว้

F3

(ตัวเลข 5 ถูกเก็บ) ใส่ #4001 (PORT B) ลงใน DPTR แล้วนำค่า A (ตัวเลข) ส่งไป ใน DPTR เพราะฉะนั้น จะได้มีการส่งข้อมูลตัวเลขออกไปทาง PORT B ใน COLUM C จากนั้นต้องการให้ตัวเลขติดอยู่ระยะหนึ่งใช้โปรแกรมหน่วงเวลา จะมีการทำงานวน LOOP R1 256 ครั้ง ต่อ R2 1 ครั้ง จน R2 ครบ 256 ครั้ง (วน LOOP 256×256 รอบ) จากนั้นผ่านโปรแกรมมาโดยการใส่ค่า 0 ใน A แล้วให้แสดงผลใน ADDRESS DPTR ทำให้จอภาพดับ จากนั้นจะ JUMP ไปยัง STA จนกว่า R0 จะมีค่าเป็น 0 ซึ่งจะทำให้ COLUM ลดลงทีละ 1 และตัวเลขก็ลดลงทีละ 1 ด้วย

บทที่ 5

สรุปวิจารณ์

ในการออกแบบ SINGLE BOARD ขึ้นนี้เพื่อต้องการให้ SINGLE BOARD ทำงานในขั้นพื้นฐานได้ คือ การรับส่งข้อมูลจากภายนอก การเก็บข้อมูลความจำ การนำข้อมูลออกมาใช้ทางคณิตศาสตร์ เช่น การบวก, ลบ คูณ, หาร และเครื่อง SINGLE BOARD นี้สามารถแสดงผลออกมาทาง DISPLAY ในรูปแบบต่าง ๆ มากมาย ทั้งนี้ขึ้นอยู่กับโปรแกรมที่เขียนเข้าไป SINGLE BOARD ที่ใช้ CPU ตัวนี้จะมีข้อได้เปรียบเหนือกว่า SINGLE BOARD ที่ใช้ CPU ตัวอื่น เนื่องจากมี FUNCTION ภายในตัวและหน่วยความจำภายในตัว ซึ่งเราสามารถป้อนคำสั่งเข้าไปในหน่วยความจำภายในตัว CHIP แล้วให้ทำงานตาม FUNCTION ได้ทันที ~~อาธิเช่น~~

- TIMER 0, TIMER 1

สามารถทำงานได้หลายลักษณะ อาจเป็นการนับจำนวนสัญญาณที่เข้าตัว CHIP หรือการนับสัญญาณภายในระบบ เพื่อเป็นวงจรหน่วงเวลาก็ได้ โดยไม่จำเป็นต้องใช้โปรแกรมหน่วงเวลา

- RXD, TXD

ใช้ส่งข้อมูลแบบ SERIAL PORT โดยต่อเข้ากับ IC เบอร์ 1488, 1489 จะสามารถติดต่อกับ PORT RS 232 ของ COMPUTER ได้ทันที ทำให้เราสามารถนำข้อมูลที่ทำงานอยู่ออกไปอ่านบนจอภาพได้ หรือใช้งานร่วมกับ COMPUTER ได้ง่ายมาก อาจใช้ติดต่อกับ MODEM ในงานโทรศัพท์ต่าง ๆ ได้อีกด้วย

- INTERRUPT

สามารถ INTERRUPT ได้หลายแบบ ให้ทำงานจากสัญญาณภายนอก หรือทำงานจากสัญญาณภายในระบบก็ได้

SINGLE BOARD มีประโยชน์มากในการใช้งานกับอุปกรณ์อิเลคทรอนิคโดยทั่วไป การพัฒนา CPU ตัวใหม่ ๆ เพื่อใช้สร้างเป็น SINGLE BOARD จะได้ประโยชน์แตกต่างกันออกไป ตามคุณสมบัติเฉพาะของ CPU แต่ละเบอร์ และในฟังค์ชันที่ต่างกันของ CPU ซึ่งเบอร์

ต่างกันอาจจะมีความสามารถต่าง ๆ กัน ทั้งนี้ ผู้ใช้จะต้องศึกษาเกี่ยวกับการเขียนโปรแกรมของ CPU ตัวนั้น ๆ ให้ละเอียด ก็จะสามารถพัฒนาขีดความสามารถในการใช้งานของ SINGLE BOARD ออกไปได้อีก

ภาคผนวก

```
                DPHH    EQU     0FFH
                DPL     EQU     82H
                DPH     EQU     83H
                P0      EQU     80H
                P1      EQU     90H
                P2      EQU     0A0H
                P3      EQU     0B0H
                PCON    EQU     87H
                SCON    EQU     98H
                SBUF    EQU     99H
                TCON    EQU     88H
                TMOD    EQU     89H
                TL0     EQU     8AH
                TH0     EQU     8CH
                TL1     EQU     8BH
                TH1     EQU     8DH
                IE      EQU     0A8H
                IP      EQU     0B8H
                ACC     EQU     0E0H
                B       EQU     0F0H
                PSW     EQU     0D0H
                R0      EQU     00H
                R1      EQU     01H
                R2      EQU     02H
                R3      EQU     03H
                R4      EQU     04H
                R5      EQU     05H
                R6      EQU     06H
                R7      EQU     07H
                EXQ     EQU     0AFH
                EA      EQU     0AFH
                ;
                ORG     0000H
                LJMP    CONP
                ;
                ORG     0030H
CONP:           MOV     R0,#0
                MOV     A,#0FEH
                MOV     P1,A
                MOV     R1,#40H
                MOV     @R1,A
                ;
                CLR     EXQ
                SETB    EA
                MOV     R1,#10
                MOV     DPTR,#0
                MOV     A,#0
                ;
LOOP:           INC     A
                INC     DPTR
```

1000 no.

000A
0008

0002
0000 0010
1 1110
FE

```
A  9040[illegible]          MOV   DPTR,#4002
D  EA        KB_5:          MOV   A,R2
   F0                       MOVX  @DPTR,A
F  EC                       MOVX  A,@DPTR
   54F0                     ANL   A,#0F0H
2  64F0                     XRL   A,#0F0H
4  7006                     JNZ   KB_7
6  0A                       INC   R2
7  BA06F7      KB_6:        CJNE  R2,#6,KB_5
A  0124                     AJMP  KB_4
C  514D        KB_7:        ACALL DIFF
E  EA                       MOV   A,R2
   904002                   MOV   DPTR,#4002
2  F0                       MOVX  @DPTR,A
3  E0                       MOVX  A,@DPTR
4  FC                       MOV   R4,A
5  54F0                     ANL   A,#0F0H
7  64F0                     XRL   A,#0F0H
9  60DB                     JZ    KB_4
8  EC                       MOV   A,R4
C  7B04                     MOV   R3,#4
E  23        KB_9:          RLC   A
F  5004                     JNC   KB_10
1  DBFB                     DJNZ  R3,KB_9
3  0124                     AJMP  KB_4
5  1B           KB_10:      DEC   R3
6  EA                       MOV   A,R2
7  23                       RL    A
8  23                       RL    A
9  2B                       ADD   A,R3
A  541F                     ANL   A,#00011111B
C  900006                   MOV   DPTR,#KBD_06
   F0                       MOVX  @DPTR,A
0  904002       KB_11:      MOV   DPTR,#4002
3  E0                       MOVX  A,@DPTR
4  54F0                     ANL   A,#0F0H
6  64F0                     XRL   A,#0F0H
8  6009                     JZ    KB_12
A  514D                     ACALL DIFF
C  904002                   MOV   DPTR,#4002
F  EA                       MOV   A,R2
0  F0                       MOVX  @DPTR,A
   80ED                     SJMP  KB_11
5  900006       KB_12:      MOV   DPTR,#KBD_06
6  E0                       MOVX  A,@DPTR
7  FA                       MOV   R2,A
8  902007                   MOV   DPTR,#STTS
B  E0                       MOVX  A,@DPTR
   BA0515       F0:         CJNE  R2,#05,F1
   902007                   MOV   DPTR,#STTS
2  7D00                     MOV   R5,#0
   B40005                   CJNE  A,#5,F01
   7401                     MOV   A,#1
   F0                       MOVX  @DPTR,A
   01F7                     AJMP  DATA
   B40165       F01:        CJNE  A,#1,F0E
   7403                     MOV   A,#3
   F0                       MOVX  @DPTR,A
   01F7                     AJMP  DATA
   BA0402       F1:         CJNE  R2,#04,F3
   214D                     AJMP  MD
   BA020C       F3:         CJNE  R2,#02,F4
   7D00                     MOV   R5,#0
   B40102                   CJNE  A,#1,F31
   21F0                     AJMP  S1
   B40740       F31:        CJNE  A,#7,F32
   21F0                     AJMP  S1
   BA[illegible]            CJNE  R2,#[illegible],F5
```

```
0A6  21E0          AJMP   33
0A8  BA0300  F4:   CJNE   R2 #H03,F5
0AB  7D00          MOV    R5,#0
0AD  B40102        CJNE   A,#H01,F41
0B0  21F4          AJMP   DEC
0B2  B4033F  F41:  CJNE   A,#H03,SOE
0B5  21F4          AJMP   DEC
0B7  BA0609  F5:   CJNE   R2 #H04,F7
0BA  B40004        CJNE   A,#0 SOF3B
0BD  7D0           MOV    R5 #
0BF  01F8          AJMP   DATA0
0C1  2164  SOF3B:  AJMP   SOF3
0C3  BA0107  F7:   CJNE   R2 #H01,F20
0C6  B4002B        CJNE   A,#1 SOE
0C9  7D02  F71:    MOV    R5,#2
0CB  01F8          AJMP   DATA0
0CD  90038F  F20:  MOV    DPTR #KD_T
0D0  C0E0          PUSH   ACC
0D2  EA            MOV    A R2
0D3  93            MOVC   A,@A+DPTR
0D4  FA            MOV    R2 A
0D5  D0E0          POP    ACC
0D7  B40002        CJNE   A #0 F21
0DA  2179          AJMP   SOF4
0DC  B40102  F21:  CJNE   A #1,F210
0DF  2192          AJMP   S1
0E1  B40302  F210: CJNE   A #3,F22
0E4  2192          AJMP
0E6  B40204  F22:  CJNE   A #2 F220
0E9  7D00          MOV    R5,#0
0EB  21A2          AJMP   F3
0ED  B40404  F220: CJNE   A #4, F
0F0  7D00          MOV    R5 #0
0F2  21A2          AJMP   F3
0F4  020026  SOE:  LJMP   KB_4
0F7  00      DATA: NOP
0F8  902002  DATA0: MOV   DPTR #DISP2
0FB  7A02          MOV    R2 #2
0FD  E0      DATA2: MOVX  A,@DPTR
0FE  540F          ANI    A,#H0F
100  FB            MOV    R3 A
101  A3            INC    DPTR
102  E0            MOVX   A,@DPTR
103  540F          ANI    A #H0F
105  FC            MOV    R4,A
106  EC            MOV    A,R4
107  C4            SWAP   A
108  2B            ADD    A,R3
109  902009        MOV    DPTR,#PCL
10C  BA0101        CJNE   R2,#1 DATA3
10F  A3            INC    DPTR
110  F0      DATA3: MOVX  @DPTR A
111  902004        MOV    DPTR,#DISP4
114  DAE7          DJNZ   R2 DATA2
116  BD0003        CJNE   R5 #0,SRB
119  FA      DATA4: MOV    R2 A
11A  902006        MOV    DPTR,#PCL
11D  E0            MOVX   A,@DPTR
11E  FB            MOV    R3 A
11F  902007        MOV    DPTR,#STT3
122  E0            MOVX   A,@DPTR
123  B40104        CJNE   A #1,DATA5
126  904005        MOV    DPTR,#H4002
129  7402          MOV    A #2
12B  F0            MOVX   @DPTR A
12C  8B87  DATA5   MOV    STATUS R3
12E  0A            INC    R2
```

```
2E  8A83                MOV     [illegible] F2
30  E0                  MOVX    [illegible] @DPTR
31  FA                  MOV     [illegible],A
32  904002              MOV     DPTR,#H4002
35  7400                MOV     A,#0
37  F0                  MOVX    @DPTR,A
38  EA                  MOV     [illegible],
37  902000              MOV     DPTR,#DIGPO
3C  540F                ANL     A,#0FH
3E  F0                  MOVX    @DPTR,A
3F  EA                  MOV     A,R2
40  C4                  SWAP    A
41  540F                ANL     A,#0FH
43  A3                  INC     DPTR
44  F0                  MOVX    @DPTR,A
45  020026              LJMP    KB_4
48  2192        S1A:    AJMP    S1
4A  4106        SRB:    AJMP    LSR
4C  00          SOF2:   NOP
4D  902007      MON:    MOV     DPTR,#STT5
50  7400                MOV     A,#0
52  F0                  MOVX    @DPTR,A
53  902000              MOV     DPTR,#DTSPO
56  7A06                MOV     R2,#6
58  7410                MOV     A,#10H
5A  F0          SOF20:  MOVX    @DPTR,A
5B  A3                  INC     DPTR
5C  DAFC                DJNZ    R2,SOF20
5E  7411                MOV     A,#11H
60  F0                  MOVX    @DPTR,A
61  020026              LJMP    KB_4
64  902008      SOF3:   MOV     DPTR,#PC1
67  E0                  MOVX    A,@DPTR
68  FA                  MOV     R2,A
69  A3                  INC     DPTR
6A  E0                  MOVX    A,@DPTR
6B  FB                  MOV     R3,A
6C  904002              MOV     DPTR,#H4002
6F  7408                MOV     A,#H08
71  F0                  MOVX    @DPTR,A
72  8B83                MOV     DPH,R3
74  8A82                MOV     DPL,R2
76  7400                MOV     A,#0
78  73                  JMP     @A+DPTR
79  758300      SOF4:   MOV     DPH,#H00
7C  7B04                MOV     R3,#H04
7E  7C03                MOV     R4,#3
80  8B82        SHIFT:  MOV     DPL,R3
82  E0                  MOVX    A,@DPTR
83  A3                  INC     DPTR
84  F0                  MOVX    @DPTR,A
85  1B                  DEC     R3
86  DCF8                DJNZ    R4,SHIFT
88  EA                  MOV     A,R2
89  540F                ANL     A,#H0F
8B  758202              MOV     DPL,#H02
8E  F0                  MOVX    @DPTR,A
8F  020026              JMP     KB_4
92  90[illegible]  S1:  MOV     [illegible],#DTSPO
95  EA                  MOV     A,R2
96  540F        S1L     ANL     [illegible],#0FH
98  F0                  MOVX    @DPTR,A
99  90[illegible]       MOV     [illegible]
9C  E0                  MOVX    [illegible],A
9D  04                  [illegible]
9F  F0                  [illegible]
```

```
5   E0              MOVX    @DPTR,A
6   A3              INC     DPTR
7   F0              MOVX    @DPTR,A
8   C4              SWAP    A
9   FB              MOV     R3,A
A   15B7            DEC     DPH
C   E4              MOV     A,R4
D   540F            ANL     A,#0FH
F   F0              MOVX    @DPTR,A
0   2B              ADD     A,R3
1   FB              MOV     R3,A
2   902008          MOV     DPTR,#2008H
5   E0              MOVX    A,@DPTR
6   FA              MOV     R2,A
7   A3              INC     DPTR
8   E0              MOVX    A,@DPTR
9   C0C0            PUSH    ACC
B   902007          MOV     DPTR,#2007H
E   E0              MOVX    A,@DPTR
F   B4020A          CJNE    A,#02H,S23
2   904007          MOV     DPTR,#4007H
5   7408            MOV     A,#08H
7   F0              MOVX    @DPTR,A
8   D003    S23:    POP     DPH
A   8A82            MOV     DPL,R2
C   EB              MOV     A,R3
D   F0              MOVX    @DPTR,A
E   904002          MOV     DPTR,#4002H
1   7400            MOV     A,#00H
3   F0              MOVX    @DPTR,A
4   902007          MOV     DPTR,#2007H
7   E0              MOVX    A,@DPTR
8   14              DEC     A
9   F0              MOVX    @DPTR,A
A   7890            MOV     R0,#90H
C   514D    S21:    ACALL   DELAY
E   D8FC            DJNZ    R0,S21
0   90200C  ST:     MOV     DPTR,#2007H
3   E0      S052:   MOVX    A,@DPTR
4   04              INC     A
5   B41002          CJNE    A,#10H,S351
8   7400            MOV     R4,#00H
A   540F    S3S1:   ANL     A,#0FH
C   F0              MOVX    @DPTR,A
D   A3              INC     DPTR
E   E0F3            MOVX    A,@DPTR
0   7D00            MOV     R5,#00H
2   01F7            AJMP    DEC7
4   90200C  DCC:    MOV     DPTR,#2007H
7   E0      DEC7:   MOVX    A,@DPTR
8   14              DEC     A
9   B4FF03          CJNE    A,#0FFH,DEC7
C   7A0F            MOV     A,#0FH
E   F0      DEC7:   MOVX    @DPTR,A
F   A3              INC     DPTR
0   E4              CLR     A
1   7B00            MOV     R3,#00H
3   01F7            AJMP    DEC7
5   90200B  S1:     MOV     DPTR,#200BH
8   7A00            MOV     R2,#00H
A   E0      RC1     MOVX    A,@DPTR
B   A3              INC     DPTR
C   AB              MOV     R3,DPH
E   F5              MOV     DPH,A
F   1A              DEC     R2
0   E9FB            MOV     A,R1
2   70F6            JNZ     RC1
4   E0              MOVX    A,@DPTR
5   A3              INC     DPTR
6   AB              MOV     R3,DPH
8   F5              MOV     DPH,A
9   83              MOV     DPL,R3
A   E0              MOVX    A,@DPTR
B   FA              MOV     R2,A
C   A3              INC     DPTR
D   E0              MOVX    A,@DPTR
E   FB              MOV     R3,A
F   EA              MOV     A,R2
0   4B              ORL     A,R3
1   7003            JNZ     S2
3   02000B          LJMP    S1
6   90200C  S2:     MOV     DPTR,#200CH
9   E0              MOVX    A,@DPTR
A   14              DEC     A
B   F0              MOVX    @DPTR,A
C   7890            MOV     R0,#90H
E   514D    S22:    ACALL   DELAY
0   D8FC            DJNZ    R0,S22
2   02000C          LJMP    ST
5   7F00    DELAY:  MOV     R7,#00H
7   7E00    D1:     MOV     R6,#00H
9   DEFE    D2:     DJNZ    R6,D2
B   DFFA            DJNZ    R7,D1
D   22              RET
E   00              NOP
F   00              NOP
```

```
0254  [illegible]      B   ;  [illegible]
[illegible]  [illegible]
[illegible]  [illegible]
025C  [illegible]
025C  0F               INC  [illegible]
025D  [illegible]
0260  [illegible]
0261  [illegible]
0264  F0               MOVX [illegible]
0265  7E46             ADJ  [illegible]
0267  0EFE      KB_T:  DW   [illegible]
0269  [illegible]
026C  [illegible]
026E  F0               [illegible]
026F  BF0C DD          CJNE [illegible]
0272  22               RET
0273  3F         TAB_S DI   3F
0274  06               DB   06Z
0275  5B               DB   [illegible]
0276  4F               DB   4F
0277  66               DB   [illegible]
0278  6D               DB   H6D
0279  7D               DB   H7D
027A  07               DB   [illegible]
027B  7F               DB   [illegible]
027C  6F               DB   H6F
027D  77               DB   H77
027E  7C               DB
027F  39               DB
0280  5E               DB   [illegible]
0281  79               DB   [illegible]
0282  71               DB   [illegible]
0283  00               DB   [illegible]
0284  40               DB   [illegible]
0285  00         KB_T: DB   0
0286  00               DB   [illegible]
0287  00               DB   [illegible]
0288  00               DB   [illegible]
0289  00               DB   [illegible]
028A  00               DB   [illegible]
028B  00               DB   [illegible]
028C  00               DB   [illegible]
028D  0F               DB   H0F
028E  08               DB   H8
028F  07               DB   H7
0290  03               DB   H3
0291  0E               DB   H0E
0292  00               DB   H00
0293  06               DB   H06
0294  02               DB   H02
0295  02               DB   H00
0296  09               DB   H09
0297  05               DB   H05
0298  01               DB   H01
0299  0C               DB   H0C
029A  08               DB   H08
029B  04               DB   H04
029C  0[illegible]     DB   H0[illegible]

No Fatal errors

SYMBOL:
ACC      00E0 : B        00[illegible]  BF       2000 : CONP     0006 : CON      0008 :
DISP4    2004 : DATA0    0FF[illegible]  [illegible]  2001 : DISP2    2002 : DPH      0083 :
DAT[illegible]2  [illegible]  DP_A   [illegible]  A_A5     [illegible]  : D_SP0    2000 : DPL[illegible]  007E :
DPHP     0[illegible]  : D[illegible]  0[illegible]  D_SP5    2[illegible]  : DATA[illegible]  01[illegible]  DATA     00F7 :
DF[illegible]  01[illegible]  : DI_F    02[illegible]  [illegible]  00[illegible]  DF_[illegible]  01E7 : ACC      01F4 :
EX0      00A8 : EA       [illegible]  [illegible]  00[illegible]  [illegible]  00DF : P220     00ED :
P31      [illegible]  : P        0[illegible]  [illegible]  [illegible]  P3[illegible]  00B[illegible]  : P3       0099 :
P0       0[illegible]  : [illegible]  0[illegible]  [illegible]  00       [illegible]  0009 : P0[illegible]  008C :
P2[illegible]0  [illegible]  P7       [illegible]           1[illegible]  : TB_0    0055 : KB_4     0026 :
KB_9     004[illegible]  : K[illegible]  12     00       K[illegible]  00[illegible]  : KB_1     [illegible]285 : KB_3     0267 :
KB_2     [illegible]     I[illegible]                    [illegible]     [illegible]  0[illegible]45    KB_5     002D :
KEY      [illegible]     MON     [illegible]             [illegible]  00[illegible]  PF[illegible]  0009    PCON     0087 :
                                                                                                  T
```

8051 INSTRUCTION SET SUMMARY

BOOLEAN VARIABLE MANIPULATION

Mnemonic		Description	Byte	Oscillator Period
CLR	C	Clear Carry	1	12
CLR	bit	Clear direct bit	2	12
SETB	C	Set Carry	1	12
SETB	bit	Set direct bit	2	12
CPL	C	Complement Carry	1	12
CPL	bit	Complement direct bit	2	12
ANL	C,bit	AND direct bit to Carry	2	24
ANL	C,/bit	AND complement of direct bit to Carry	2	24
ORL	C,bit	OR direct bit to Carry	2	24
ORL	C,/bit	OR complement of direct bit to Carry	2	24
MOV	C,bit	Move direct bit to Carry	2	12
MOV	bit,C	Move Carry to direct bit	2	24
JC	rel	Jump if Carry is set	2	24
JNC	rel	Jump if Carry not set	2	24
JB	bit,rel	Jump if direct Bit is set	3	24
JNB	bit,rel	Jump if direct Bit is Not set	3	24
JBC	bit,rel	Jump if direct Bit is set & clear bit	3	24

8051 INSTRUCTION SET SUMMARY

Rn = Register

LOGICAL OPERATIONS

Mnemonic		Description	Byte	Oscillator Period
ANL	A,Rn	AND register to Accumulator	1	12
ANL	A,direct	AND direct byte to Accumulator	2	12
ANL	A,@Ri	AND indirect RAM to Accumulator	1	12
ANL	A,#data	AND immediate data to Accumulator	2	12
ANL	direct,A	AND Accumulator to direct byte	2	12
ANL	direct,#data	AND immediate data to direct byte	3	24
ORL	A,Rn	OR register to Accumulator	1	12
ORL	A,direct	OR direct byte to Accumulator	2	12
ORL	A,@Ri	OR indirect RAM to Accumulator	1	12
ORL	A,#data	OR immediate data to Accumulator	2	12
ORL	direct,A	OR Accumulator to direct byte	2	12
ORL	direct,#data	OR immediate data to direct byte	3	24
XRL	A,Rn	Exclusive-OR register to Accumulator	1	12
XRL	A,direct	Exclusive-OR direct byte to Accumulator	2	12

LOGICAL OPERATIONS Cont.

Mnemonic		Description	Byte	Oscillator Period
XRL	A,@Ri	Exclusive-OR indirect RAM to Accumulator	1	12
XRL	A,#data	Exclusive-OR immediate data to Accumulator	2	12
XRL	direct,A	Exclusive-OR Accumulator to direct byte	2	12
XRL	direct,#data	Exclusive-OR immediate data to direct byte	3	24
CLR	A	Clear Accumulator	1	12
CPL	A	Complement Accumulator	1	12
RL	A	Rotate Accumulator Left	1	12
RLC	A	Rotate Accumulator Left through the Carry	1	12
RR	A	Rotate Accumulator Right	1	12
RRC	A	Rotate Accumulator Right through the Carry	1	12
SWAP	A	Swap nibbles within the Accumulator	1	12

8051 INSTRUCTION SET SUMMARY

ARITHMETIC OPERATIONS

Mnemonic		Description	Byte	Oscillator Period
ADD	A,Rn	Add register to Accumulator	1	12
ADD	A,direct	Add direct byte to Accumulator	2	12
ADD	A,@Ri	Add indirect RAM to Accumulator	1	12
ADD	A,#data	Add immediate data to Accumulator	2	12
ADDC	A,Rn	Add register to Accumulator with Carry	1	12
ADDC	A,direct	Add direct byte to Accumulator with Carry	2	12
ADDC	A,@Ri	Add indirect RAM to Accumulator with Carry	1	12
ADDC	A,#data	Add immediate data to Acc with Carry	2	12
SUBB	A,Rn	Subtract register from Acc with borrow	1	12
SUBB	A,direct	Subtract direct byte from Acc with borrow	2	12

8051 INSTRUCTION SET SUMMARY

ARITHMETIC OPERATIONS Cont.

Mnemonic		Description	Byte	Oscillator Period
SUBB	A,@Ri	Subtract indirect RAM from Acc with borrow	1	12
SUBB	A,#data	Subtract immediate data from Acc with borrow	2	12
INC	A	Increment Accumulator	1	12
INC	Rn	Increment register	1	12
INC	direct	Increment direct byte	2	12
INC	@Ri	Increment indirect RAM	1	12
DEC	A	Decrement Accumulator	1	12
DEC	Rn	Decrement Register	1	12
DEC	direct	Decrement direct byte	2	12
DEC	@Ri	Decrement indirect RAM	1	12
INC	DPTR	Increment Data Pointer	1	24
MUL	AB	Multiply A & B	1	48
DIV	AB	Divide A by B	1	48
DA	A	Decimal Adjust Accumulator	1	12

PROGRAM BRANCHING

Mnemonic		Description	Byte	Oscillator Period
ACALL	addr11	Absolute Subroutine Call	2	24
LCALL	addr16	Long Subroutine Call	3	24
RET		Return from Subroutine	1	24

8051 INSTRUCTION SET SUMMARY

PROGRAM BRANCHING Cont

Mnemonic		Description	Byte	Oscillator Period
RETI		Return from interrupt	1	24
AJMP	addr11	Absolute Jump	2	24
LJMP	addr16	Long Jump	3	24
SJMP	rel	Short Jump (relative addr)	2	24
JMP	@A+DPTR	Jump indirect relative to the DPTR	1	24
JZ	rel	Jump if Accumulator is Zero	2	24
JNZ	rel	Jump if Accumulator is Not Zero	2	24
CJNE	A,direct,rel	Compare direct byte to Acc and Jump if Not Equal	3	24
CJNE	A,#data,rel	Compare immediate to Acc and Jump if Not Equal	3	24
CJNE	Rn,#data,rel	Compare immediate to register and Jump If Not Equal	3	24
CJNE	@Ri,#data,rel	Compare immediate to indirect and Jump if Not Equal	3	24
DJNZ	Rn,rel	Decrement register and Jump if Not Zero	3	24
DJNZ	direct,rel	Decrement direct byte and Jump if Not Zero	3	24
NOP		No Operation	1	12

8051 INSTRUCTION SET SUMMARY

DATA TRANSFER

Mnemonic		Description	Byte	Oscillator Period
MOV	A,Rn	Move register to Accumulator	1	12
MOV	A,direct	Move direct byte to Accumulator	2	12
MOV	A,@Ri	Move indirect RAM to Accumulator	1	12
MOV	A,#data	Move immediate data to Accumulator	2	12
MOV	Rn,A	Move Accumulator to register	1	12
MOV	Rn,direct	Move direct byte to register	2	24
MOV	Rn,#data	Move immediate data to register	2	12
MOV	direct,A	Move Accumulator to direct byte	2	12
MOV	direct,Rn	Move register to direct byte	2	24
MOV	direct,direct	Move direct byte to direct	3	24
MOV	direct,@Ri	Move indirect RAM to direct byte	2	24
MOV	direct,#data	Move immediate data to direct byte	3	24
MOV	@Ri,A	Move Accumulator to indirect RAM	1	12
MOV	@Ri,direct	Move direct byte to indirect RAM	2	24
MOV	@Ri,#data	Move immediate data to indirect RAM	2	12

8051 INSTRUCTION SET SUMMARY

DATA TRANSFER Cont.

Mnemonic		Description	Byte	Oscillator Period
MOV	DPTR #data16	Load Data Pointer with a 16-bit constant	3	24
MOVC	A,@A+DPTR	Move Code byte relative to DPTR to Acc	1	24
MOVC	A,@A+PC	Move Code byte relative to PC to Acc	1	24
MOVX	A,@Ri	Move External RAM (8-bit addr) to Acc	1	24
MOVX	A,@DPTR	Move External RAM (16-bit addr) to Acc	1	24
MOVX	@Ri,A	Move Acc to External RAM (8-bit addr)	1	24
MOVX	@DPTR,A	Move Acc to External RAM (16-bit addr)	1	24
PUSH	direct	Push direct byte onto stack	2	24
POP	direct	Pop direct byte from stack.	2	24
XCH	A,Rn	Exchange register with Accumulator	1	12
XCH	A,direct	Exchange direct byte with Accumulator	2	12
XCH	A,@Ri	Exchange indirect RAM with Accumulator	1	12
XCHD	A,@Ri	Exchange low-order Digit indirect RAM with Acc	1	12

ACALL addr11

Function: Absolute Call

Description: ACALL unconditionally calls a subroutine located at the indicated address. The instruction increments the PC twice to obtain the address of the following instruction, then pushes the 16-bit result onto the stack (low-order byte first) and increments the stack pointer twice. The destination address is obtained by successively concatenating the five high-order bits of the incremented PC, op-code bits 7-5, and the second byte of the instruction. The subroutine called must therefore start within the same 2K block of the program memory as the first byte of the instruction following ACALL. No flags are affected.

Example: Initially SP equals 07H. The label "SUBRTN" is at program memory location 0345H. After executing the instruction,

```
ACALL   SUBRTN
```

at location 0123H, SP will contain 09H, internal RAM locations 08H and 09H will contain 25H and 01H, respectively, and the PC will contain 0345H.

Bytes: 2

Cycles: 2

Encoding:

a10	a9	a8	1	0	0	0	1	a7	a6	a5	a4	a3	a2	a1	a0

Operation:

ACALL
(PC) ← (PC) + 2
(SP) ← (SP) + 1
((SP)) ← (PC_{7-0})
(SP) ← (SP) + 1
((SP)) ← (PC_{15-8})
(PC_{10-0}) ← page address

ADD A,<src-byte>

Function: Add

Description: ADD adds the byte variable indicated to the accumulator, leaving the result in the accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

OV is set if there is a carry-out of bit 6 but not out of bit 7, or a carry-out of bit 7 but not bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative operands.

Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example: The accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B). The instruction,

```
ADD     A,R0
```

will leave 6DH (01101101B) in the accumulator with the AC flag cleared and both the carry flag and OV set to 1.

ADD A,Rn

Bytes: 1

Cycles: 1

Encoding:

0	0	1	0	1	r	r	r

Operation:

```
ADD
(A) ← (A) + (Rn)
```

ADD A,direct

Bytes: 2

Cycles: 1

Encoding:

0	0	1	0	0	1	0	1	direct address

Operation:

```
ADD
(A) ← (A) + (direct)
```

ADD A,@Ri

Bytes: 1

Cycles: 1

Encoding:

0	0	1	0	0	1	1	i

Operation:

```
ADD
(A) ← (A) + ((Ri))
```

ADD A,#data

Bytes: 2

Cycles: 1

Encoding:

0	0	1	0	0	1	0	0	immediate data

Operation:

```
ADD
(A) ← (A) + #data
```

ADDC A, <src-byte>

Function: Add with Carry

Description: ADDC simultaneously adds the byte variable indicated, the carry flag and the accumulator contents, leaving the result in the accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

OV is set if there is a carry-out of bit 6 but not out of bit 7, or a carryout of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands or a positive sum from two negative operands.

Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example: The accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) with the carry flag set. The instruction,

```
ADDC    A,R0
```

will leave 6EH (01101110B) in the accumulator with AC cleared and both the carry flag and OV set to 1.

ADDC A,Rn

Bytes: 1

Cycles: 1

Encoding:

0	0	1	1	1	r	r	r

Operation: ADDC

$(A) \leftarrow (A) + (C) + (R_n)$

ADDC A,direct

Bytes: 2

Cycles: 1

Encoding:

0	0	1	1	0	1	0	1	direct address

Operation: ADDC

$(A) \leftarrow (A) + (C) + (direct)$

ADDC A,@Ri

Bytes: 1
Cycles: 1

Encoding:

0	0	1	1	0	1	1	i

Operation: ADDC
(A) ← (A) + (C) + ((Ri))

ADDC A,#data

Bytes: 2
Cycles: 1

Encoding:

0	0	1	1	0	1	0	0	immediate data

Operation: ADDC
(A) ← (A) + (C) + #data

AJMP addr11

Function: Absolute Jump
Description: AJMP transfers program execution to the indicated address, which is formed at run-time by concatenating the high-order five bits of the PC (*after* incrementing the PC twice), opcode bits 7-5, and the second byte of the instruction. The destination must therefore be within the same 2K block of program memory as the first byte of the instruction following AJMP.
Example: The label "JMPADR" is at program memory location 0123H. The instruction,

AJMP JMPADR

is at location 0345H and will load the PC with 0123H.

Bytes: 2
Cycles: 2

Encoding:

a10	a9	a8	0	0	0	0	1	a7	a6	a5	a4	a3	a2	a1	a0

Operation: AJMP
(PC) ← (PC) + 2
(PC10-0) ← page address

ANL <dest-byte> , <src-byte>

Function: Logical-AND for byte variables

Description: ANL performs the bitwise logical-AND operation between the variables indicated and stores the results in the destination variable. No flags are affected.

The two operands allow six addressing mode combinations. When the destination is the accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the accumulator or immediate data.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, *not* the input pins.

Example: If the accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) then the instruction,

```
ANL     A,R0
```

will leave 41H (01000001B) in the accumulator.

When the destination is a directly addressed byte, this instruction will clear combinations of bits in any RAM location or hardware register. The mask byte determining the pattern of bits to be cleared would either be a constant contained in the instruction or a value computed in the accumulator at run-time. The instruction,

```
ANL     P1,#01110011B
```

will clear bits 7, 3, and 2 of output port 1.

ANL A,Rn

Bytes: 1

Cycles: 1

Encoding:

0	1	0	1	1	r	r	r

Operation: ANL

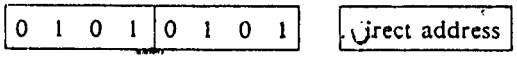
(A) ← (A) ∧ (Rn)

ANL A,direct

Bytes: 2

Cycles: 1

Encoding:



Operation: ANL

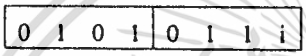
(A) ← (A) ∧ (direct)

ANL A,@RI

Bytes: 1

Cycles: 1

Encoding:



Operation: ANL

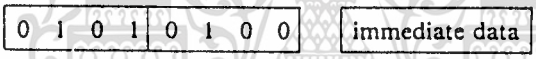
(A) ← (A) ∧ ((Ri))

ANL A,#data

Bytes: 2

Cycles: 1

Encoding:



Operation: ANL

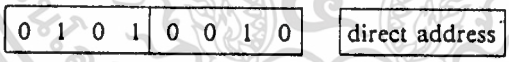
(A) ← (A) ∧ #data

ANL direct,A

Bytes: 2

Cycles: 1

Encoding:



Operation: ANL

(direct) ← (direct) ∧ (A)

ANL direct,#data

Bytes: 3

Cycles: 2

Encoding:

Operation: ANL

(direct) ← (direct) ∧ #data

ANL C, <src-bit>

Function: Logical-AND for bit variables

Description: If the Boolean value of the source bit is a logical 0 then clear the carry flag; otherwise leave the carry flag in its current state. A slash ("/") preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, *but the source bit itself is not affected.* No other flags are affected.

Only direct bit addressing is allowed for the source operand.

Example: Set the carry flag if, and only if, P1.0 = 1, ACC. 7 = 1, and OV = 0:

```
MOV  C,P1.0        ;LOAD CARRY WITH INPUT PIN STATE
ANL  C,ACC.7       ;AND CARRY WITH ACCUM. BIT 7
ANL  C,/OV         ;AND WITH INVERSE OF OVERFLOW
                    FLAG
```

ANL C,bit

Bytes: 2

Cycles: 2

Encoding:

1	0	0	0	0	0	1	0	bit address

Operation: ANL

(C) ← (C) ∧ (bit)

ANL C,/bit

Bytes: 2

Cycles: 2

Encoding:

1	0	1	1	0	0	0	0	bit address

Operation: ANL

(C) ← (C) ∧ ¬ (bit)

CJNE <dest-byte>,<src-byte>, rel

Function: Compare and Jump if Not Equal.

Description: CJNE compares the magnitudes of the first two operands, and branches if their values are not equal. The branch destination is computed by adding the signed relative-displacement in the last instruction byte to the PC, after incrementing the PC to the start of the next instruction. The carry flag is set if the unsigned integer value of <dest-byte> is less than the unsigned integer value of <src-byte>; otherwise, the carry is cleared. Neither operand is affected.

The first two operands allow four addressing mode combinations: the accumulator may be compared with any directly addressed byte or immediate data, and any indirect RAM location or working register can be compared with an immediate constant.

Example: The accumulator contains 34H. Register 7 contains 56H. The first instruction in the sequence,

```
                CJNE    R7,#60H, NOT_EQ
;               ...     .....                   ; R7 = 60H.
NOT_EQ:         JC      REQ_LOW                 ; IF R7 < 60H.
;               ...     .....                   ; R7 > 60H.
```

sets the carry flag and branches to the instruction at label NOT_EQ. By testing the carry flag, this instruction determines whether R7 is greater or less than 60H.

If the data being presented to port 1 is also 34H, then the instruction,

```
WAIT:  CJNE   A,P1,WAIT
```

clears the carry flag and continues with the next instruction in sequence, since the accumulator does equal the data read from P1. (If some other value was being input on P1, the program will loop at this point until the P1 data changes to 34H.)

CJNE A,direct,rel

Bytes: 3

Cycles: 2

Encoding:

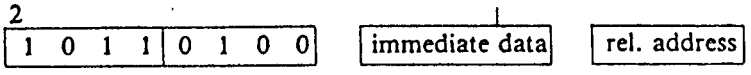
1 0 1 1	0 1 0 1	direct address	rel. address

Operation:

```
(PC) ← (PC) + 3
IF (A) <> (direct)
THEN
                (PC) ← (PC) + relative offset
IF (A) < (direct)
THEN
                (C) ← 1
ELSE
                (C) ← 0
```

CJNE A,#data,rel

Bytes: 3
Cycles: 2
Encoding:



Operation:

(PC) ← (PC) + 3
IF (A) <> *data*
THEN
(PC) ← (PC) + *relative offset*
IF (A) < *data*
THEN
(C) ← 1
ELSE
(C) ← 0

CJNE Rn,#data,rel

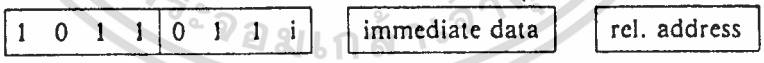
Bytes: 3
Cycles: 2
Encoding:

Operation:

(PC) ← (PC) + 3
IF (Rn) <> *data*
THEN
(PC) ← (PC) + *relative offset*
IF (Rn) < *data*
THEN
(C) ← 1
ELSE
(C) ← 0

CJNE @Ri,#data,rel

Bytes: 3
Cycles: 2
Encoding:



Operation:

(PC) ← (PC) + 3
IF ((Ri)) <> *data*
THEN
(PC) ← (PC) + *relative offset*
IF ((Ri)) < *data*
THEN
(C) ← 1
ELSE
(C) ← 0

CLR A

Function: Clear Accumulator
Description: The accumulator is cleared (all bits set to zero). No flags are affected.
Example: The accumulator contains 5CH (01011100B). The instruction,

```
CLR     A
```

will leave the accumulator set to 00H (00000000B).

Bytes: 1
Cycles: 1

Encoding:

1	1	1	0	0	1	0	0

Operation:

```
CLR
(A)←—0
```

CLR bit

Function: Clear bit
Description: The indicated bit is cleared (reset to zero). No other flags are affected. CLR can operate on the carry flag or any directly addressable bit.
Example: Port 1 has previously been written with 5DH (01011101B). The instruction,

```
CLR     P1.2
```

will leave the port set to 59H (01011001B).

CLR C

Bytes: 1
Cycles: 1

Encoding:

1	1	0	0	0	0	1	1

Operation:

```
CLR
(C)←—0
```

CLR bit

Bytes: 2
Cycles: 1

Encoding:

1	1	0	0	0	0	1	0	bit address

Operation:

```
CLR
(bit)←—0
```

CPL A

Function: Complement Accumulator

Description: Each bit of the accumulator is logically complemented (one's complement). Bits which previously contained a one are changed to zero and vice-versa. No flags are affected.

Example: The accumulator contains 5CH (01011100B). The instruction,

```
CPL   A
```

will leave the accumulator set to 0A3H (10100011B).

Bytes: 1

Cycles: 1

Encoding:

1	1	1	1	0	1	0	0

Operation:

```
CPL
(A)←¬(A)
```

CPL bit

Function: Complement bit

Description: The bit variable specified is complemented. A bit which had been a one is changed to zero and vice-versa. No other flags are affected. CLR can operate on the carry or any directly addressable bit.

Note: When this instruction is used to modify an output pin, the value used as the original data will be read from the output data latch, *not* the input pin.

Example: Port 1 has previously been written with 5BH (01011101B). The instruction sequence,

```
CPL   P1.1
CPL   P1.2
```

will leave the port set to 5BH (01011011B).

CPL C

Bytes: 1

Cycles: 1

Encoding:

1	0	1	1	0	0	1	1

Operation:

```
CPL
(C)←¬(C)
```

CPL bit

Bytes: 2
Cycles: 1

Encoding:

1	0	1	1	0	0	1	0	bit address

Operation: CPL
(bit)←¬(bit)

DA A

Function: Decimal-adjust Accumulator for Addition

Description: DA A adjusts the eight-bit value in the accumulator resulting from the earlier addition of two variables (each in packed-BCD format), producing two four-bit digits. Any ADD or ADDC instruction may have been used to perform the addition.

If accumulator bits 3-0 are greater than nine (xxxx1010-xxxx1111), or if the AC flag is one, six is added to the accumulator producing the proper BCD digit in the low-order nibble. This internal addition would set the carry flag if a carry-out of the low-order four-bit field propagated through all high-order bits, but it would not clear the carry flag otherwise.

If the carry flag is now set, or if the four high-order bits now exceed nine (1010xxxx-1111xxxx), these high-order bits are incremented by six, producing the proper BCD digit in the high-order nibble. Again, this would set the carry flag if there was a carry-out of the high-order bits, but wouldn't clear the carry. The carry flag thus indicates if the sum of the original two BCD variables is greater than 100, allowing multiple precision decimal addition. OV is not affected.

All of this occurs during the one instruction cycle. Essentially, this instruction performs the decimal conversion by adding 00H, 06H, 60H, or 66H to the accumulator, depending on initial accumulator and PSW conditions.

Note: DA A *cannot* simply convert a hexadecimal number in the accumulator to BCD notation, nor does DA A apply to decimal subtraction.

Example: The accumulator holds the value 56H (01010110B) representing the packed BCD digits of the decimal number 56. Register 3 contains the value 67H (01100111B) representing the packed BCD digits of the decimal number 67. The carry flag is set. The instruction sequence,

```
ADDC    A,R3
DA      A
```

will first perform a standard twos-complement binary addition, resulting in the value 0BEH (10111110) in the accumulator. The carry and auxiliary carry flags will be cleared.

The Decimal Adjust instruction will then alter the accumulator to the value 24H (00100100B), indicating the packed BCD digits of the decimal number 24, the low-order two digits of the decimal sum of 56, 67, and the carry-in. The carry flag will be set by the Decimal Adjust instruction, indicating that a decimal overflow occurred. The true sum 56, 67, and 1 is 124.

BCD variables can be incremented or decremented by adding 01H or 99H. If the accumulator initially holds 30H (representing the digits of 30 decimal), then the instruction sequence,

```
ADD     A,#99H
DA      A
```

will leave the carry set and 29H in the accumulator, since 30 + 99 = 129. The low-order byte of the sum can be interpreted to mean 30 − 1 = 29.

Bytes: 1
Cycles: 1

Encoding:

1	1	0	1	0	1	0	0

Operation:

```
DA
-contents of Accumulator are BCD
IF  [[(A3-0) > 9] ∨ [(AC) = 1]]
    THEN  (A3-0) ← (A3-0) + 6
          AND
IF  [[(A7-4) >9] ∨ [(C) = 1]]
    THEN  (A7-4) ← (A7-4) + 6
```

DEC byte

Function: Decrement

Description: The variable indicated is decremented by 1. An original value of 00H will underflow to 0FFH. No flags are affected. Four operand addressing modes are allowed: accumulator, register, direct, or register-indirect.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, *not* the input pins.

Example: Register 0 contains 7FH (01111111B). Internal RAM locations 7EH and 7FH contain 00H and 40H, respectively. The instruction sequence,

```
DEC  @R0
DEC  R0
DEC  @R0
```

will leave register 0 set to 7EH and internal RAM locations 7EH and 7FH set to 0FFH and 3FH.

DEC A

Bytes: 1

Cycles: 1

Encoding:

0	0	0	1	0	1	0	0

Operation: DEC

(A) ← (A) − 1

DEC Rn

Bytes: 1

Cycles: 1

Encoding:

0	0	0	1	1	r	r	r

Operation: DEC

(Rn) ← (Rn) − 1

DEC direct

Bytes: 2
Cycles: 1

Encoding:

0	0	0	1	0	1	0	1	direct address

Operation: DEC
(direct) ← (direct) − 1

DEC @Ri

Bytes: 1
Cycles: 1

Encoding:

0	0	0	1	0	1	1	i

Operation: DEC
((Ri)) ← ((Ri)) − 1

DIV AB

Function: Divide

Description: DIV AB divides the unsigned eight-bit integer in the accumulator by the unsigned eight-bit integer in register B. The accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry and OV flags will be cleared.

Exception: if B had originally contained 00H, the values returned in the accumulator and B-register will be undefined and the overflow flag will be set. The carry flag is cleared in any case.

Example: The accumulator contains 251 (0FBH or 11111011B) and B contains 18 (12H or 00010010B). The instruction,

DIV AB

will leave 13 in the accumulator (0DH or 00001101B) and the value 17 (11H or 00010001B) in B, since 251 = (13 x 18) + 17. Carry and OV will both be cleared.

Bytes: 1
Cycles: 4

Encoding:

1	0	0	0	0	1	0	0

Operation: DIV
$(A)_{15-8} \leftarrow (A) / (B)$
$(B)_{7-0}$

DJNZ <byte>,<rel-addr>

Function: Decrement and Jump if Not Zero

Description: DJNZ decrements the location indicated by 1, and branches to the address indicated by the second operand if the resulting value is not zero. An original value of 00H will underflow to 0FFH. No flags are affected. The branch destination would be computed by adding the signed relative-displacement value in the last instruction byte to the PC, after incrementing the PC to the first byte of the following instruction.

The location decremented may be a register or directly addressed byte.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, *not* the input pins.

Example: Internal RAM locations 40H, 50H, and 60H contain the values 01H, 70H, and 15H, respectively. The instruction sequence,

```
DJNZ    40H,LABEL__1
DJNZ    50H,LABEL__2
DJNZ    60H,LABEL__3
```

will cause a jump to the instruction at label LABEL__2 with the values 00H, 6FH, and 15H in the three RAM locations. The first jump was *not* taken because the result was zero.

This instruction provides a simple way of executing a program loop a given number of times, or for adding a moderate time delay (from 2 to 512 machine cycles) with a single instruction. The instruction sequence,

```
        MOV   R2,#8
TOGGLE: CPL   P1.7
        DJNZ  R2,TOGGLE
```

will toggle P1.7 eight times, causing four output pulses to appear at bit 7 of output port 1. Each pulse will last three machine cycles; two for DJNZ and one to alter the pin.

DJNZ Rn,rel

Bytes: 2
Cycles: 2

Encoding:

Operation: DJNZ
(PC) ← (PC) + 2
(Rn) ← (Rn) − 1
IF (Rn) > 0 or (Rn) < 0
THEN
(PC) ← (PC) + rel

DJNZ direct,rel

Bytes: 3
Cycles: 2

Encoding:

Operation: DJNZ
(PC) ← (PC) + 2
(direct) ← (direct) − 1
IF (direct) > 0 or (direct) < 0
THEN
(PC) ← (PC) + rel

INC <byte>

Function: Increment
Description: INC increments the indicated variable by 1. An original value of 0FFH will overflow to 00H. No flags are affected. Three addressing modes are allowed: register, direct, or register-indirect.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, *not* the input pins.

INC @Ri

Bytes: 1
Cycles: 1

Encoding:

0	0	0	0	0	1	1	i

Operation: INC
((Ri)) ← ((Ri)) + 1

INC DPTR

Function: Increment Data Pointer

Description: Increment the 16-bit data pointer by 1. A 16-bit increment (modulo 2^{16}) is performed; an overflow of the low-order byte of the data pointer (DPL) from 0FFH to 00H will increment the high-order byte (DPH). No flags are affected.

This is the only 16-bit register which can be incremented.

Example: Registers DPH and DPL contain 12H and 0FEH, respectively. The instruction sequence,

```
                12FE
INC     DPTR    12FF
INC     DPTR    1300
INC     DPTR    1301
```

will change DPH and DPL to 13H and 01H.

Bytes: 1
Cycles: 2

Encoding:

1	0	1	0	0	0	1	1

Operation: INC
(DPTR) ← (DPTR) + 1

JB bit,rel

Function: Jump if Bit set

Description: If the indicated bit is a one, jump to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. *The bit tested is not modified.* No flags are affected.

Example: The data present at input port 1 is 11001010B. The accumulator holds 56 (01010110B). The instruction sequence,

```
JB    P1.2,LABEL1
JB    ACC.2,LABEL2
```

will cause program execution to branch to the instruction at label LABEL2.

Bytes: 3

Cycles: 2

Encoding:

0 0 1 0	0 0 0 0	bit address	rel. address

Operation:

```
JB
(PC) ← (PC) + 3
IF  (bit) = 1
    THEN
        (PC) ← (PC) + rel
```

JBC bit,rel

Function: Jump if Bit is set and Clear bit

Description: If the indicated bit is one, branch to the address indicated; otherwise proceed with the next instruction. *In either case, clear the designated bit.* The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. No flags are affected.

Note: When this instruction is used to test an output pin, the value used as the original data will be read from the output data latch, *not* the input pin.

Example: The accumulator holds 56H (01010110B). The instruction sequence,

```
JBC   ACC.3,LABEL1
JBC   ACC.2,LABEL2
```

will cause program execution to continue at the instruction identified by the label LABEL2, with the accumulator modified to 52H (01010010B).

Bytes: 3
Cycles: 2

Encoding:

0 0 0 1	0 0 0 0	bit address	rel. address

Operation: JBC

(PC) ← (PC) + 3
IF (bit) = 1.
THEN
(bit) ← 0
(PC) ← (PC) + rel

JC rel

Function: Jump if Carry is set

Description: If the carry flag is set, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. No flags are affected.

Example: The carry flag is cleared. The instruction sequence,

```
JC    LABEL1
CPL   C
JC    LABEL2
```

will set the carry and cause program execution to continue at the instruction identified by the label LABEL2.

Bytes: 2
Cycles: 2

Encoding:

0 1 0 0	0 0 0 0	rel. address

Operation: JC

(PC) ← (PC) + 2
IF (C) = 1
THEN
(PC) ← (PC) + rel

JMP @A + DPTR

Function: Jump indirect

Description: Add the eight-bit unsigned contents of the accumulator with the sixteen-bit data pointer, and load the resulting sum to the program counter. This will be the address for subsequent instruction fetches. Sixteen-bit addition is performed (modulo 2^{16}): a carry-out from the low-order eight bits propagates through the higher-order bits. Neither the accumulator nor the data pointer is altered. No flags are affected.

Example: An even number from 0 to 6 is in the accumulator. The following sequence of instructions will branch to one of four AJMP instructions in a jump table starting at JMP__TBL:

```
            MOV     DPTR,#JMP__TBL
            JMP     @A + DPTR
JMP__TBL:   AJMP    LABEL0
            AJMP    LABEL1
            AJMP    LABEL2
            AJMP    LABEL3
```

If the accumulator equals 04H when starting this sequence, execution will jump to label LABEL2. Remember that AJMP is a two-byte instruction, so the jump instructions start at every other address.

Bytes: 1

Cycles: 2

Encoding:

0	1	1	1	0	0	1	1

Operation: JMP

(PC) ← (A) + (DPTR)

JNB bit,rel

Function: Jump if Bit Not set

Description: If the indicated bit is a zero, branch to the indicated address; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. *The bit tested is not modified.* No flags are affected.

Example: The data present at input port 1 is 11001010B. The accumulator holds 56H (01010110B). The instruction sequence,

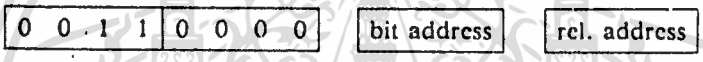
```
JNB     P1.3,LABEL1
JNB     ACC.3,LABEL2
```

will cause program execution to continue at the instruction at label LABEL2.

Bytes: 3

Cycles: 2

Encoding:



Operation:

```
JNB
(PC) ← (PC) + 3
IF (bit) = 0
    THEN (PC) ← (PC) + rel.
```

JNC rel

Function: Jump if Carry not set

Description: If the carry flag is a zero, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice to point to the next instruction. The carry flag is not modified.

Example: The carry flag is set. The instruction sequence,

```
JNC   LABEL1
CPL   C
JNC   LABEL2
```

will clear the carry and cause program execution to continue at the instruction identified by the label LABEL2.

Bytes: 2
Cycles: 2

Encoding:

0 1 0 1	0 0 0 0	rel. address

Operation:

```
JNC
(PC) ← (PC) + 2
IF (C) = 0
    THEN (PC) ← (PC) + rel
```

JNZ rel

Function: Jump if accumulator Not Zero

Description: If any bit of the accumulator is a one, branch to the indicated address; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The accumulator is not modified. No flags are affected.

Example: The accumulator originally holds 00H. The instruction sequence,

```
JNZ  LABEL1
INC  A
JNZ  LABEL2
```

will set the accumulator to 01H and continue at label LABEL2.

Bytes: 2
Cycles: 2

Encoding:

0 1 1 1	0 0 0 0	rel. address

Operation:

```
JNZ
(PC) ← (PC) + 2
IF (A) ≠ 0
    THEN (PC) ← (PC) + rel
```

JZ rel

Function: Jump if Accumulator Zero

Description: If all bits of the accumulator are zero, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The accumulator is not modified. No flags are affected.

Example: The accumulator originally contains 01H. The instruction sequence,

```
JZ    LABEL1
DEC   A
JZ    LABEL2
```

will change the accumulator to 00H and cause program execution to continue at the instruction identified by the label LABEL2.

Bytes: 2

Cycles: 2

Encoding:

0 1 1 0	0 0 0 0	rel. address

Operation:

```
JZ
(PC) ← (PC) + 2
IF  (A) = 0
    THEN  (PC) ← (PC) + rel
```

LCALL addr16

Function: Long Call

Description: LCALL calls a subroutine located at the indicated address. The instruction adds three to the program counter to generate the address of the next instruction and then pushes the 16-bit result onto the stack (low byte first), incrementing the stack pointer by two. The high-order and low-order bytes of the PC are then loaded, respectively, with the second and third bytes of the LCALL instruction. Program execution continues with the instruction at this address. The subroutine may therefore begin anywhere in the full 64K-byte program memory address space. No flags are affected.

Example: Initially the stack pointer equals 07H. The label "SUBRTN" is assigned to program memory location 1234H. After executing the instruction,

LCALL SUBRTN

at location 0123H, the stack pointer will contain 09H, internal RAM locations 08H and 09H will contain 26H and 01H, and the PC will contain 1235H.

Bytes: 3
Cycles: 2

Encoding:

0 0 0 1	0 0 1 0	addr15 - addr8	addr7 - addr0

Operation: LCALL

(PC) ← (PC) + 3
(SP) ← (SP) + 1
((SP)) ← (PC_{7-0})
(SP) ← (SP) + 1
((SP)) ← (PC_{15-8})
(PC) ← $addr_{15-0}$

LJMP addr16

Function: Long Jump

Description: LJMP causes an unconditional branch to the indicated address, by loading the high-order and low-order bytes of the PC (respectively) with the second and third instruction bytes. The destination may therefore be anywhere in the full 64K program memory address space. No flags are affected.

Example: The label "JMPADR" is assigned to the instruction at program memory location 1234H. The instruction,

LJMP JMPADR

at location 0123H will load the program counter with 1234H.

Bytes: 3
Cycles: 2

Encoding:

0 0 0 0	0 0 1 0	addr15 - addr8	addr7 - addr0

Operation: LJMP

(PC) ← $addr_{15-0}$

MOV **<dest-byte>,<src-byte>**

Function: Move byte variable

Description: The byte variable indicated by the second operand is copied into the location specified by the first operand. The source byte is not affected. No other register or flag is affected.

This is by far the most flexible operation. Fifteen combinations of source and destination addressing modes are allowed.

Example: Internal RAM location 30H holds 40H. The value of RAM location 40H is 10H. The data present at input port 1 is 11001010B (0CAH).

```
MOV   R0,#30H          ;R0<= 30H
MOV   A,@R0            ;A <= 40H
MOV   R1,A             ;R1 <= 40H
MOV   B,@R1            ;B <= 10H
MOV   @R1,P1           ;RAM (40H) <= 0CAH
MOV   P2, P1           ;P2 #0CAH
```

leaves the value 30H in register 0, 40H in both the accumulator and register 1, 10H in register B, and 0CAH (11001010B) both in RAM location 40H and output on port 2.

MOV A,Rn

Bytes: 1

Cycles: 1

Encoding:

1	1	1	0	1	r	r	r

Operation: MOV
(A) ← (Rn)

***MOV A,direct**

Bytes: 2

Cycles: 1

Encoding:

1	1	1	0	0	1	0	1	direct address

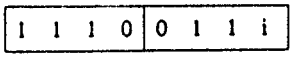
Operation: MOV
(A) ← (direct)

*MOV A,ACC is not a valid instruction.

MOV A,@Ri

Bytes: 1
Cycles: 1

Encoding:

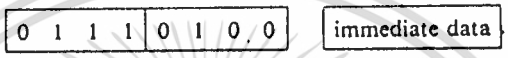


Operation: MOV
(A) ← ((Ri))

MOV A,#data

Bytes: 2
Cycles: 1

Encoding:



Operation: MOV
(A) ← #data

MOV Rn,A

Bytes: 1
Cycles: 1

Encoding:

Operation: MOV
(Rn) ← (A)

MOV Rn,direct

Bytes: 2
Cycles: 2

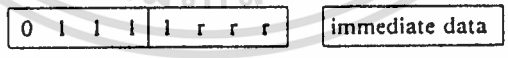
Encoding:

Operation: MOV
(Rn) ← (direct)

MOV Rn,#data

Bytes: 2
Cycles: 1

Encoding:



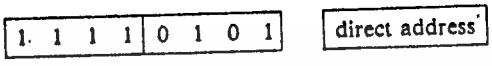
Operation: MOV
(Rn) ← #data

MOV direct,A

Bytes: 2

Cycles: 1

Encoding:



Operation: MOV
(direct) ← (A)

MOV direct,Rn

Bytes: 2

Cycles: 2

Encoding:

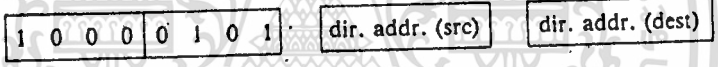
Operation: MOV
(direct) ← (Rn)

MOV direct,direct

Bytes: 3

Cycles: 2

Encoding:



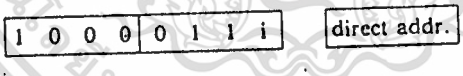
Operation: MOV
(direct) ← (direct)

MOV direct,@Ri

Bytes: 2

Cycles: 2

Encoding:



Operation: MOV
(direct) ← ((Ri))

MOV direct,#data

Bytes: 3

Cycles: 2

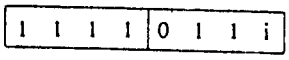
Encoding:

Operation: MOV
(direct) ← #data

MOV @Ri,A

Bytes: 1
Cycles: 1

Encoding:

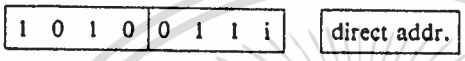


Operation: MOV
((Ri)) ← (A)

MOV @Ri,direct

Bytes: 2
Cycles: 2

Encoding:



Operation: MOV
((Ri)) ← (direct)

MOV @Ri,#data

Bytes: 2
Cycles: 1

Encoding:

Operation: MOV
((RI)) ← #data

MOV <dest-bit>,<src-bit>

Function: Move bit data

Description: The Boolean variable indicated by the second operand is copied into the location specified by the first operand. One of the operands must be the carry flag; the other may be any directly addressable bit. No other register or flag is affected.

Example: The carry flag is originally set. The data present at input port 3 is 11000101B. The data previously written to output port 1 is 35H (00110101B).

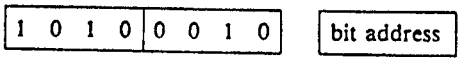
```
MOV  P1.3,C
MOV  C,P3.3
MOV  P1.2,C
```

will leave the carry cleared and change port 1 to 39H (00111001B).

MOV C,bit

Bytes: 2
Cycles: 1

Encoding:

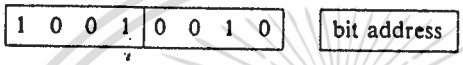


Operation: MOV
(C) ← (bit)

MOV bit,C

Bytes: 2
Cycles: 2

Encoding:



Operation: MOV
(bit) ← (C)

MOV DPTR,#data16

Function: Load Data Pointer with a 16-bit constant

Description: The data pointer is loaded with the 16-bit constant indicated. The 16-bit constant is loaded into the second and third bytes of the instruction. The second byte (DPH) is the high-order byte, while the third byte (DPL) holds the low-order byte. No flags are affected.

This is the only instruction which moves 16 bits of data at once.

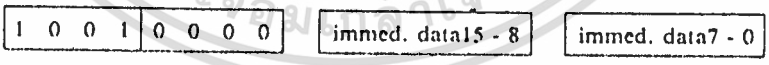
Example: The instruction,

MOV DPTR,#1234H

will load the value 1234H into the data pointer: DPH will hold 12H and DPL will hold 34H.

Bytes: 3
Cycles: 2

Encoding:



Operation: MOV
(DPTR) ← #data$_{15\text{-}0}$
DPH ☐ DPL ← #data$_{15\text{-}8}$ ☐ #data$_{7\text{-}0}$

MOVC A,@A + ‹base-reg›

Function: Move Code byte

Description: The MOVC instructions load the accumulator with a code byte, or constant from program memory. The address of the byte fetched is the sum of the original unsigned eight-bit accumulator contents and the contents of a sixteen-bit base register, which may be either the data pointer or the PC. In the latter case, the PC is incremented to the address of the following instruction before being added with the accumulator: otherwise the base register is not altered. Sixteen-bit addition is performed so a carry-out from the low-order eight bits may propagate through higher-order bits. No flags are affected.

Example: A value between 0 and 3 is in the accumulator. The following instructions will translate the value in the accumulator to one of four values defined by the DB (define byte) directive.

```
REL_PC:  INC     A
         MOVC    A,@A+PC
         RET
         DB      66H
         DB      77H
         DB      88H
         DB      99H
```

If the subroutine is called with the accumulator equal to 01H, it will return with 77H in the accumulator. The INC A before the MOVC instruction is needed to "get around" the RET instruction above the table. If several bytes of code separated the MOVC from the table, the corresponding number would be added to the accumulator instead.

MOVC A,@A+DPTR

Bytes: 1

Cycles: 2

Encoding:

1	0	0	1	0	0	1	1

Operation: MOVC

(A) ← ((A) + (DPTR))

MOVC A,@A+PC

Bytes: 1
Cycles: 2

Encoding:

1	0	0	0	0	0	1	1

Operation: MOVC
(PC)←(PC) + 1
(A)←((A) + (PC))

MOVX <dest-byte>,<src-byte>

Function: Move External

Description: The MOVX instructions transfer data between the accumulator and a byte of external data memory, hence the "X" appended to MOV. There are two types of instructions, differing in whether they provide an eight-bit or sixteen-bit indirect address to the external data RAM.

In the first type, the contents of R0 or R1 in the current register bank provide an eight-bit address multiplexed with data on P0. Eight bits are sufficient for external I/O expansion decoding or a relatively small RAM array. For somewhat larger arrays, any output port pins can be used to output higher-order address bits. These pins would be controlled by an output instruction preceding the MOVX.

In the second type of MOVX instruction, the data pointer generates a sixteen-bit address. P2 outputs the high-order eight address bits (the contents of DPH) while P0 multiplexes the low-order eight bits (DPL) with data. The P2 Special Function Register retains its previous contents while the P2 output buffers are emitting the contents of DPH. This form is faster and more efficient when accessing very large data arrays (up to 64K bytes), since no additional instructions are needed to set up the output ports.

It is possible in some situations to mix the two MOVX types. A large RAM array with its high-order address lines driven by P2 can be addressed via the data pointer, or with code to output high-order address bits to P2 followed by a MOVX instruction using R0 or R1.

Example: An external 256 byte RAM using multiplexed address/data lines (e.g., an Intel 8155 RAM/I/O/Timer) is connected to the 8051 Port 0. Port 3 provides control lines for the external RAM. Ports 1 and 2 are used for normal I/O. Registers 0 and 1 contain 12H and 34H. Location 34H of the external RAM holds the value 56H. The instruction sequence,

```
MOVX    A,@R1
MOVX    @R0,A
```

copies the value 56H into both the accumulator and external RAM location 12H.

MOVX A,@Ri

Bytes: 1
Cycles: 2

Encoding:

1	1	1	0	0	0	1	i

Operation: MOVX
(A) ← ((Ri))

MOVX A,@DPTR

Bytes: 1
Cycles: 2

Encoding:

1	1	1	0	0	0	0	0

Operation: MOVX
(A) ← ((DPTR))

MOVX @Ri,A

Bytes: 1
Cycles: 2

Encoding:

1	1	1	1	0	0	1	i

Operation: MOVX
((Ri)) ← (A)

MOVX @DPTR,A

Bytes: 1
Cycles: 2

Encoding:

1	1	1	1	0	0	0	0

Operation: MOVX
(DPTR) ← (A)

MUL AB

Function: Multiply

Description: MUL AB multiplies the unsigned eight-bit integers in the accumulator and register B. The low-order byte of the sixteen-bit product is left in the accumulator, and the high-order byte in B. If the product is greater than 255 (0FFH) the overflow flag is set; otherwise it is cleared. The carry flag is always cleared.

Example: Originally the accumulator holds the value 80 (50H). Register B holds the value 160 (0A0H). The instruction,

```
MUL     AB
```

will give the product 12,800 (3200H), so B is changed to 32H (00110010B) and the accumulator is cleared. The overflow flag is set, carry is cleared.

Bytes: 1

Cycles: 4

Encoding:

1 0 1 0	0 1 0 0

Operation:

```
MUL
(A)7-0 ←— (A) X (B)
(B)15-8
```

NOP

Function: No Operation

Description: Execution continues at the following instruction. Other than the PC, no registers or flags are affected.

Example: It is desired to produce a low-going output pulse on bit 7 of port 2 lasting exactly 5 cycles. A simple SETB/CLR sequence would generate a one-cycle pulse, so four additional cycles must be inserted. This may be done (assuming no interrupts are enabled) with the instruction sequence,

```
CLR     P2.7
NOP
NOP
NOP
NOP
SETB    P2.7
```

Bytes: 1

Cycles: 1

Encoding:

0 0 0 0	0 0 0 0

Operation:

```
NOP
(PC) ←— (PC) + 1
```

ORL <dest-byte> <src-byte>

Function: Logical-OR for byte variables

Description: ORL performs the bitwise logical-OR operation between the indicated variables, storing the results in the destination byte. No flags are affected.

The two operands allow six addressing mode combinations. When the destination is the accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the accumulator or immediate data.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, *not* the input pins.

Example: If the accumulator holds 0C3H (11000011B) and R0 holds 55H (01010101B) then the instruction,

```
ORL     A,R0
```

will leave the accumulator holding the value 0D7H (11010111B).

When the destination is a directly addressed byte, the instruction can set combinations of bits in any RAM location or hardware register. The pattern of bits to be set is determined by a mask byte, which may be either a constant data value in the instruction or a variable computed in the accumulator at run-time. The instruction,

```
ORL     P1,#00110010B
```

will set bits 5, 4, and 1 of output port 1.

ORL A,Rn

Bytes: 1

Cycles: 1

Encoding:

0	1	0	0	1	r	r	r

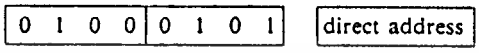
Operation: ORL

(A)←—(A) ∨ (Rn)

ORL A,direct

Bytes: 2
Cycles: 1

Encoding:

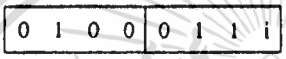


Operation: ORL
(A) ← (A) ∨ (direct)

ORL A,@Ri

Bytes: 1
Cycles: 1

Encoding:

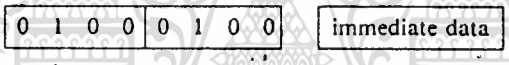


Operation: ORL
(A) ← (A) ∨ ((Ri))

ORL A,#data

Bytes: 2
Cycles: 1

Encoding:

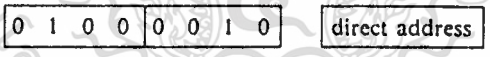


Operation: ORL
(A) ← (A) ∨ #data

ORL direct,A

Bytes: 2
Cycles: 1

Encoding:



Operation: ORL
(direct) ← (direct) ∨ (A)

ORL direct,#data

Bytes: 3
Cycles: 2

Encoding:

Operation: ORL
(direct) ← (direct) ∨ #data

ORL C, <src-bit>

Function: Logical-OR for bit variables

Description Set the carry flag if the Boolean value is a logical 1; leave the carry in its current state otherwise. A slash ("/") preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is not affected. No other flags are affected.

Example: Set the carry flag if and only if P1.0 = 1, ACC.7 = 1, or OV = 0:

```
MOV  C,P1.0        ;LOAD CARRY WITH INPUT PIN P10
ORL  C,ACC.7       ;OR CARRY WITH THE ACC. BIT 7
ORL  C,/OV         :OR CARRY WITH THE INVERSE OF OV.
```

ORL C,bit

Bytes: 2

Cycles: 2

Encoding:

0 1 1 1	0 0 1 0	bit address

Operation: ORL

(C)←(C) ∨ (bit)

ORL C,/bit

Bytes: 2

Cycles: 2

Encoding:

1 0 1 0	0 0 0 0	bit address

Operation: ORL

(C)←(C) ∨ ($\overline{\text{bit}}$)

POP direct

Function: Pop from stack.

Description: The contents of the internal RAM location addressed by the stack pointer is read, and the stack pointer is decremented by one. The value read is the transfer to the directly addressed byte indicated. No flags are affected.

Example: The stack pointer originally contains the value 32H, and internal RAM locations 30H through 32H contain the values 20H, 23H, and 01H, respectively. The instruction sequence,

```
POP   DPH
POP   DPL
```

will leave the stack pointer equal to the value 30H and the data pointer set to 0123H. At this point the instruction,

```
POP   SP
```

will leave the stack pointer set to 20H. Note that in this special case the stack pointer was decremented to 2FH before being loaded with the value popped (20H).

Bytes: 2
Cycles: 2

Encoding:

1	1	0	1	0	0	0	0	direct address

Operation:

```
POP
(direct) ← ((SP))
  (SP) ← (SP) – 1
```

PUSH direct

Function: Push onto stack

Description: The stack pointer is incremented by one. The contents of the indicated variable is then copied into the internal RAM location addressed by the stack pointer. Otherwise no flags are affected.

Example: On entering an interrupt routine the stack pointer contains 09H. The data pointer holds the value 0123H. The instruction sequence,

```
PUSH   DPL
PUSH   DPH
```

will leave the stack pointer set to 0BH and store 23H and 01H in internal RAM locations 0AH and 0BH, respectively.

Bytes: 2
Cycles: 2

Encoding:

1	1	0	0	0	0	0	0	direct address

Operation:

```
PUSH
 (SP) ← (SP) + 1
((SP)) ← (direct)
```

RET

Function: Return from subroutine

Description: RET pops the high- and low-order bytes of the PC successively from the stack, decrementing the stack pointer by two. Program execution continues at the resulting address, generally the instruction immediately following an ACALL or LCALL. No flags are affected.

Example: The stack pointer originally contains the value 0BH. Internal RAM locations 0AH and 0BH contain the values 23H and 01H, respectively. The instruction,

RET

will leave the stack pointer equal to the value 09H. Program execution will continue at location 0123H.

Bytes: 1

Cycles: 2

Encoding:

0	0	1	0	0	0	1	0

Operation: RET

$(PC_{15-8}) \leftarrow ((SP))$
$(SP) \leftarrow (SP) - 1$
$(PC_{7-0}) \leftarrow ((SP))$
$(SP) \leftarrow (SP) - 1$

RETI

Function: Return from interrupt

Description: RETI pops the high- and low-order bytes of the PC successively from the stack, and restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. The stack pointer is left decremented by two. No other registers are affected; the PSW is *not* automatically restored to its pre-interrupt status. Program execution continues at the resulting address, which is generally the instruction immediately after the point at which the interrupt request was detected. If a lower- or same-level interrupt had been pending when the RETI instruction is executed, that one instruction will be executed before the pending interrupt is processed.

Example: The stack pointer originally contains the value 0BH. An interrupt was detected during the instruction ending at location 0122H. Internal RAM locations 0AH and 0BH contain the values 23H and 01H, respectively. The instruction,

RETI

will leave the stack pointer equal to 09H and return program execution to location 0123H.

Bytes: 1
Cycles: 2

Encoding:

0	0	1	1	0	0	1	0

Operation: RETI
$(PC_{15-8}) \leftarrow ((SP))$
$(SP) \leftarrow (SP) - 1$
$(PC_{7-0}) \leftarrow ((SP))$
$(SP) \leftarrow (SP) - 1$

RL A

Function: Rotate accumulator Left
Description: The eight bits in the accumulator are rotated one bit to the left. Bit 7 is rotated into the bit 0 position. No flags are affected.
Example: The accumulator holds the value 0C5H (11000101B). The instruction,

RL A

leaves the accumulator holding the value 8BH (10001011B) with the carry unaffected.

Bytes: 1
Cycles: 1

Encoding:

0	0	1	0	0	0	1	1

Operation: RL
$(A_{n+1}) \leftarrow (An)$ $n = 0-6$
$(A0) \leftarrow (A7)$

RLC A

Function: Rotate accumulator Left through the Carry flag
Description: The eight bits in the accumulator and the carry flag are together rotated one bit to the left. Bit 7 moves into the carry flag; the original state of the carry flag moves into the bit 0 position. No other flags are affected.
Example: The accumulator holds the value 0C5H (11000101B), and the carry is zero. The instruction,

RLC A

leaves the accumulator holding the value 8BH (10001010B) with the carry set.

Bytes: 1
Cycles: 1

Encoding:

0	0	1	1	0	0	1	1

Operation: RLC
$(An+1) \leftarrow (An)$ $n = 0-6$
$(A0) \leftarrow (C)$
$(C) \leftarrow (A7)$

RR A

Function: Rotate accumulator Right

Description: The eight bits in the accumulator are rotated one bit to the right. Bit 0 is rotated into the bit 7 position. No flags are affected.

Example: The accumulator holds the value 0C5H (11000101B). The instruction,

```
RR    A
```

leaves the accumulator holding the value 0E2H (11100010B) with the carry unaffected.

Bytes: 1

Cycles: 1

Encoding:

0	0	0	0	0	0	1	1

Operation:

```
RR
(An) ← (An+1)     n=0–6
(A7) ← (A0)
```

RRC A

Function: Rotate accumulator Right through Carry flag

Description: The eight bits in the accumulator and the carry flag are together rotated one bit to the right. Bit 0 moves into the carry flag; the original value of the carry flag moves into the bit 7 position. No other flags are affected.

Example: The accumulator holds the value 0C5H (11000101B), the carry is zero. The instruction,

```
RRC    A
```

leaves the accumulator holding the value 62 (01100010B) with the carry set.

Bytes: 1

Cycles: 1

Encoding:

0	0	0	1	0	0	1	1

Operation:

```
RRC
(An) ← (An+1)     n=0–6
(A7) ← (C)
(C) ← (A0)
```

SETB <bit>

Function: Set Bit

Description: SETB sets the indicated bit to one. SETB can operate on the carry flag or any directly addressable bit. No other flags are affected.

Example: The carry flag is cleared. Output port 1 has been written with the value 34H (00110100B). The instructions,

```
SETB    C
SETB    P1.0
```

will leave the carry flag set to 1 and change the data output on port 1 to 35H (00110101B).

SETB C

Bytes: 1

Cycles: 1

Encoding:

1	1	0	1	0	0	1	1

Operation: SETB
(C)←1

SETB bit

Bytes: 2

Cycles: 1

Encoding:

1	1	0	1	0	0	1	0	bit address

Operation: SETB
(bit)←1

SJMP rel

Function: Short Jump

Description: Program control branches unconditionally to the address indicated. The branch destination is computed by adding the signed displacement in the second instruction byte to the PC, after incrementing the PC twice. Therefore, the range of destinations allowed is from 128 bytes preceding this instruction to 127 bytes following it.

Example: The label "RELADR" is assigned to an instruction at program memory location 0123H. The instruction,

SJMP RELADR

will assemble into location 0100H. After the instruction is executed, the PC will contain the value 0123H.

(Note: Under the above conditions the instruction following SJMP will be at 102H. Therefore, the displacement byte of the instruction will be the relative offset (0123H-0102H) = 21H. Put another way, an SJMP with a displacement of 0FEH would be a one-instruction infinite loop.)

Bytes: 2
Cycles: 2

Encoding:

1	0	0	0	0	0	0	0	rel. address

Operation: SJMP
(PC) ← (PC) + 2
(PC) ← (PC) + rel

SUBB A, <src-byte>

Function: Subtract with borrow

Description: SUBB subtracts the indicated variable and the carry flag together from the accumulator, leaving the result in the accumulator. SUBB sets the carry (borrow) flag if a borrow is needed for bit 7, and clears C otherwise. (If C was set *before* executing a SUBB instruction, this indicates that a borrow was needed for the previous step in a multiple precision subtraction, so the carry is subtracted from the accumulator along with the source operand.) AC is set if a borrow is needed for bit 3, and cleared otherwise. OV is set if a borrow is needed into bit 6, but not into bit 7, or into bit 7, but not bit 6.

When subtracting signed integers OV indicates a negative number produced when a negative value is subtracted from a positive value, or a positive result when a positive number is subtracted from a negative number.

The source operand allows four addressing modes: register, direct, register-indirect, or immediate.

Example: The accumulator holds 0C9H (11001001B), register 2 holds 54H (01010100B), and the carry flag is set. The instruction,

SUBB A,R2

will leave the value 74H (01110100B) in the accumulator, with the carry flag and AC cleared but OV set.

Notice that 0C9H minus 54H is 75H. The difference between this and the above result is due to the carry (borrow) flag being set before the operation. If the state of the carry is not known before starting a single or multiple-precision subtraction, it should be explicitly cleared by a CLR C instruction.

SUBB A,Rn

Bytes: 1
Cycles: 1

Encoding:

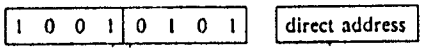
1	0	0	1	1	r	r	r

Operation: SUBB
(A) ← (A) − (C) − (Rn)

SUBB A,direct

Bytes: 2
Cycles: 1

Encoding:

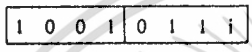


Operation: SUBB
(A) ← (A) − (C) − (direct)

SUBB A,@Ri

Bytes: 1
Cycles: 1

Encoding:

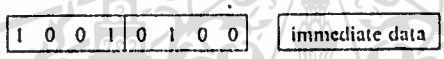


Operation: SUBB
(A) ← (A) − (C) − ((Ri))

SUBB A,#data

Bytes: 2
Cycles: 1

Encoding:



Operation: SUBB
(A) ← (A) − (C) − #data

SWAP A

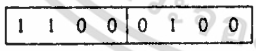
Function: Swap nibbles within the Accumulator
Description: SWAP A interchanges the low- and high-order nibbles (four-bit fields) of the accumulator (bits 3-0 and bits 7-4). The operation can also be thought of as a four-bit rotate instruction. No flags are affected.
Example: The accumulator holds the value 0C5H (11000101B). The instruction,

SWAP A

leaves the accumulator holding the value 5CH (01011100B).

Bytes: 1
Cycles: 1

Encoding:



Operation: SWAP
$(A_{3-0}) \rightleftarrows (A_{7-4})$, $(A_{7-4}) \leftarrow (A_{3-0})$

XCH A,<byte>

Function: Exchange Accumulator with byte variable

Description: XCH loads the accumulator with the contents of the indicated variable, at the same time writing the original accumulator contents to the indicated variable. The source/destination operand can use register, direct, or register-indirect addressing.

Example: R0 contains the address 20H. The accumulator holds the value 3FH (00111111B). Internal RAM location 20H holds the value 75H (01110101B). The instruction,

```
XCH     A,@R0
```

will leave RAM location 20H holding the values 3FH (00111111B) and 75H (01110101B) in the accumulator.

XCH A,Rn

Bytes: 1

Cycles: 1

Encoding:

1	1	0	0	1	r	r	r

Operation: XCH

(A) ⇄ (Rn)

XCH A,direct

Bytes: 2

Cycles: 1

Encoding:

1	1	0	0	0	1	0	1	direct address

Operation: XCH

(A) ⇄ (direct)

XCH A,@Ri

Bytes: 1

Cycles: 1

Encoding:

1	1	0	0	0	1	1	i

Operation: XCH

(A) ⇄ ((Ri))

XCHD A,@Ri

Function: Exchange Digit

Description: XCHD exchanges the low-order nibble of the accumulator (bits 3-0), generally representing a hexadecimal or BCD digit), with that of the internal RAM location indirectly addressed by the specified register. The high-order nibbles (bits 7-4) of each register are not affected. No flags are affected.

Example: R0 contains the address 20H. The accumulator holds the value 36H (00110110B). Internal RAM location 20H holds the value 75H (01110101B). The instruction,

```
XCHD    A,@R0
```

will leave RAM location 20H holding the value 76H (01110110B) and 35H (00110101B) in the accumulator.

Bytes: 1

Cycles: 1

Encoding:

1	1	0	1	0	1	1	i

Operation: XCHD

$(A_{3-0}) \rightleftarrows ((Ri_{3-0}))$

XRL **<dest-byte>, <src-byte>**

Function: Logical Exclusive-OR for byte variables

Description: XRL performs the bitwise logical Exclusive-OR operation between the indicated variables, storing the results in the destination. No flags are affected.

The two operands allow six addressing mode combinations. When the destination is the accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the accumulator or immediate data.

(Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, *not* the input pins.)

Example: If the accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) then the instruction,

```
XRL   A,R0
```

will leave the accumulator holding the value 69H (01101001B).

When the destination is a directly addressed byte, this instruction can complement combinations of bits in any RAM location or hardware register. The pattern of bits to be complemented is then determined by a mask byte, either a constant contained in the instruction or a variable computed in the accumulator at run-time. The instruction,

```
XRL   P1,#00110001B
```

will complement bits 5, 4, and 0 of output port 1.

XRL A,Rn

Bytes: 1

Cycles: 1

Encoding:

0	1	1	0	1	r	r	r

Operation:

```
XRL
(A) ← (A) ∨ (Rn)
```

XRL A,direct

Bytes: 2

Cycles: 1

Encoding:

0	1	1	0	0	1	0	1	direct address

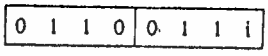
Operation:

```
XRL
(A) ← (A) ∨ (direct)
```

XRL A,@Ri

Bytes: 1
Cycles: 1

Encoding:

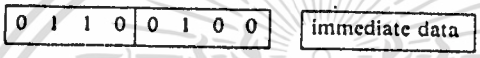


Operation: XRL
(A)←(A) ∨ ((Ri))

XRL A,#data

Bytes: 2
Cycles: 1

Encoding:

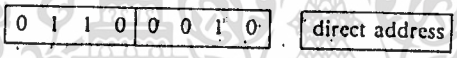


Operation: XRL
(A)←(A) ∨ #data

XRL direct,A

Bytes: 2
Cycles: 1

Encoding:



Operation: XRL
(direct)←(direct) ∨ (A)

XRL direct,#data

Bytes: 3
Cycles: 2

Encoding:

Operation: XRL
(direct)←(direct) ∨ #data

กิตติกรรมประกาศ

รายงานฉบับนี้ได้สำเร็จลงด้วยความช่วยเหลือจากเพื่อน ๆ หลายคนที่เกี่ยวข้อง และอาจารย์ที่ปรึกษา ผู้จัดทำใคร่ขอขอบคุณมา ณ. ที่นี้ด้วย

หนังสืออ้างอิง

1) ET Z-80 SINGLE BOARD MICRO COMPUTER VERSION 2 USER'S MANUAL , ETT GROUP

2) HANDBOOK OF MICROCONTROLLER MCS-51 , INTEL CORPORATION 1984

3) IC TTL DATABOOK , SCIENCE , ENGINEERING & EDUCATION CO.,LTD.