ปีการศึกษา 2533

เครื่องควบคุมที่โปรแกรมได้

โดย

นายกิตติ จัยวัฒน์

นายวสันต์ ค้างคง

นายวิชัย พุฒชู่ชื่น

นายวุฒินันท์ จาโรทก

นายไวพจน์ พ่วงสมจิตร

อาจารย์ที่ปรึกษา

อ.วิริยะ กองรัตน์

ปริญญานิพนธ์ปีการศึกษา 2533

ภาควิชา เทคโนโลยีการวัดคุมทางอุตสาหกรรม

คณะ วิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง เครื่องควบคุมที่โปรแกรมได้

ผู้จัดทำ

นายกิตติ	จัยวัฒน์	29-6201
นายวสันต์	ค้างคง	32-6217
นายวิชัย	พุฒชูชื่น	32-6219
นายวุฒินันท์	จาโรทก	32-6222
นายไวพจน์	พ่วงสมจิตร	32-6223

..... อาจารย์ที่ปรึกษา

(อ.วิริยะ กองรัตน์)

เครื่องควบคุมที่โปรแกรมได้

นายกิตติ จัยวัฒน์ 29-6201

นายวสันต์ ค้างคง 32-6217

นายวิชัย พุฒชูชื่น 32-6219

นายวุฒินันท์ จาโรทก 32-6222

นายไวพจน์ พ่วงสมจิตร 32-6223

อ.วิริยะ กองรัตน์ อาจารย์ที่ปรึกษา

บทคัดย่อ

การใช้อุปกรณ์ไฟฟ้าเชิงกล ในการควบคุมกระบวนการทางอุตสาหกรรมจะสิ้นเปลืองกำลังงานสูง มีขนาดใหญ่ และความยืดหยุ่นของระบบต่ำ ปริญญานิพนธ์ฉบับนี้ ขอเสนอการพัฒนาเครื่องควบคุมที่โปรแกรมได้ โดยใช้ไมโครคอนโทรลเลอร์ เป็นตัวจัดการ ซึ่งจะทำให้เครื่องควบคุมมีประสิทธิภาพสูง สิ้นเปลืองกำลังงานต่ำ การพัฒนาเครื่องควบคุมที่โปรแกรมได้ นี้ใช้บอร์ดไมโครคอนโทรลเลอร์ ขนาดเล็ก โดยใช้ภาษาแอสเซมบลี้ สำหรับการพัฒนา โดยใช้ชิพไมโครคอนโทรลเลอร์เบอร์ 8032 ซึ่งอยู่ในตระกูล MCS-51 ของอินเทลทำหน้าที่เป็น CPU ไมโครคอนโทรลเลอร์ ในตระกูล MCS-51 มีข้อดีกว่าไมโครโปรเซสเซอร์ Z-80 เพราะใช้อุปกรณ์ทางด้านฮาร์ดแวร์ น้อยกว่า มีเสถียรภาพในการทำงานแน่นอนกว่า และมีประสิทธิภาพดีกว่า การพัฒนาทางด้านโปรแกรมของ บอร์ดไมโครคอนโทรลเลอร์ ก็สะดวก และมีความรวดเร็ว เพราะสามารถใช้ เครื่องคอมพิวเตอร์ PC ทั่วไปเป็นเครื่องมือช่วยในการพัฒนา

PROGRAMMABLE CONTROLLER

KITTI CHAIYAWAT

WASAN KANGKONG

WICHAI PUTCHUCHUEN

WUTTIINAN CHAROTOK

VIPOT PONGSOMJIT

VIRIYA KONGRATANA ADVISOR

ABSTRACT

The electromechanical devices used in process control usually consume much power, larger size of control system and less flexibility to change the operation. This thesis presents a method of develop a programmable controller using a microcontroller as a processing unit. The developed Programmable Controller consumes low power, high efficiency. This Programmable Controller using a microcontroller number 8032 as a CPU and using assembly language. The advantage of using a microcontroller is the stability, more efficiency and can use personal computer for developing the program.

สารบัญ

บทที่ 1
บทนำ

การควบคุมกระบวนการทางอุตสาหกรรม แต่เดิมจะเป็นการนำ อุปกรณ์ไฟฟ้าเชิงกล เช่น รีเลย์ (RELAY) ตัวตั้งเวลา (TIMER) ตัวนับ (COUNTER) มาต่อรวมกันเป็นระบบควบคุม ระบบควบคุมแบบต่อเนื่องที่ประกอบกันขึ้นจากอุปกรณ์ดังกล่าวข้างต้น จะมีข้อเสียมาก เนื่องจากมีขนาดใหญ่ สิ้นเปลืองกำลังงานสูงและมีอายุการใช้งานค่อนข้างสั้น นอกจากนั้นเมื่อต้องการเปลี่ยนแปลงการทำงานของกระบวนการ ก็ต้องทำการสร้างวงจรควบคุมใหม่ ทำให้ยุ่งยากและเสียเวลามาก

การประยุกต์ใช้ไมโครคอนโทรลเลอร์ในการควบคุมกระบวนการ เป็นวิธีหนึ่งที่มีประสิทธิภาพสูง โดยเฉพาะอย่างยิ่งในด้านการควบคุมแบบต่อเนื่องจึงมีการค้นคว้า และวิจัยเครื่องควบคุมที่โปรแกรมได้ (PROGRAMMABLE CONTROLLER) ขึ้นมาใช้กับการควบคุมทางด้านอุตสาหกรรมโดยเฉพาะเพื่อทดแทนอุปกรณ์ไฟฟ้าเชิงกล

PROGRAMMABLE CONTROLLER (PC) จะมีส่วนที่เป็นอินพุทที่สามารถต่อใช้งานเข้ากับตัวตรวจจับต่าง ๆ (SENSOR) และส่วนเอาท์พุท จะต่อไปควบคุมการทำงานของอุปกรณ์หรือเครื่องจักรกลโดยสามารถสร้างวงจรหรือเงื่อนไขการทำงานของเครื่องจักรเหล่านี้ได้จากการป้อนเป็นโปรแกรมสั่งงานที่เรียกว่า LADDER DIAGRAM เข้าไปโปรแกรมนี้จะทำหน้าที่เหมือนกับวงจรรีเลย์ ตัวตั้งเวลา ตัวนับและอื่น ๆ ของวงจรรีเลย์อีก

ภายหลังจากที่มีการเลือกสวิทช์บังคับให้ PC เริ่มทำงานหรือรันโปรแกรม PC ก็จะทำงานตามขั้นตอนเหมือนกับโปรแกรมที่ผู้ใช้ป้อนเข้าไปทุกประการ โดยที่ PC จะสร้างอุปกรณ์ควบคุมต่าง ๆ ภายในเองเช่น รีเลย์ ตัวตั้งเวลา ตัวนับ ได้ด้วยซอฟท์แวร์โดยปรากฏอยู่ในรูปของฟังก์ชั่นการทำงานที่ตรงกับสภาพความเป็นจริง นอกจากนี้เงื่อนไขต่าง ๆ ที่เขียนเป็นโปรแกรมจะมีลักษณะคล้ายกับการต่อสายของอุปกรณ์เหล่านี้ กันเป็นวงจรขึ้นมา แต่เนื่องจากว่าเป็นการปฏิบัติการทางซอฟท์แวร์ จึงทำให้ผู้ใช้สามารถแก้ไข และเพิ่มเติมวงจรได้โดยการเปลี่ยนแปลงโปรแกรมของวงจรนั้น ๆ ซึ่งทำให้ สะดวก แน่นอน และง่ายกว่าการเดินสายไฟที่เป็นวงจรรีเลย์ เช่นแต่ก่อนอยู่มาก

ปริญญานิพนธ์ฉบับนี้ ได้นำเสนอเครื่องควบคุมที่โปรแกรมได้ (PC) โดยใช้เครื่องคอมพิวเตอร์ PERSONAL CONPUTER ช่วยในการสร้าง แก้ไขโปรแกรมจากนั้นแปลให้เป็นชุดคำสั่งของเครื่องควบคุมที่โปรแกรมได้ จากนั้นคำสั่งหรือหรือข้อมูลที่ได้จากการสร้างแล้ว จะถูกส่งให้กับเครื่องควบคุมที่ใช้ MICROCONTROLLER เบอร์ 8032 โดยส่งผ่านทาง PORT อนุกรมแบบ RS-232C

ในบทต่อ ๆ ไป จะได้กล่าวถึง ทฤษฎีและหลักการทั่ว ๆ ไปของเครื่องควบคุมที่โปรแกรมได้ จากนั้นจะกล่าวถึงโครงสร้างของเครื่องควบคุมที่โปรแกรมได้ระบบการจัดการของเครื่องควบคุม การพัฒนาโปรแกรม โดยใช้บอร์ดไมโครคอนโทรเลอร์ขนาดเล็กซึ่งเรียกชื่อว่า BOARD ANT-32 คำสั่งที่ใช้ในการเขียนโปรแกรม วิธีการ คอมไพล์ (COMPILES) โปรแกรมการดาวน์โหลด (DOWN LOAD) และตัวอย่างการทดลอง ส่วนบทสุดท้าย จะเป็นบทสรุปและวิจารณ์การทดลอง

บทที่ 2
ทฤษฎีทั่วไปเกี่ยวกับเครื่องควบคุมที่โปรแกรมได้

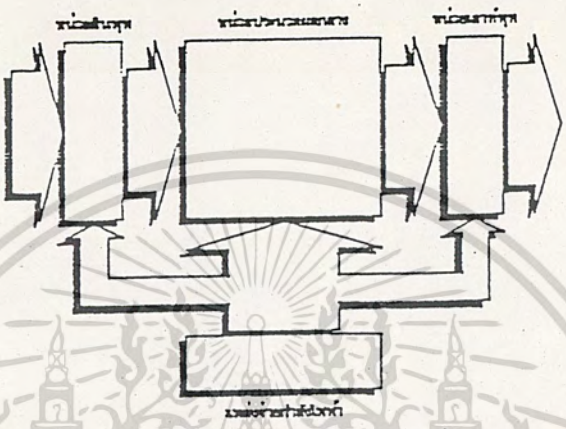
เครื่องควบคุมที่โปรแกรมได้ เป็นการประยุกต์ใช้เทคโนโลยีทางด้านไมโครโปรเซสเซอร์ทั้งทางด้านฮาร์ดแวร์ (HARDWARE) และซอฟท์แวร์ (SOFTWARE) มาใช้ในการควบคุมเครื่องจักรอัตโนมัติ หรือกระบวนการทางอุตสาหกรรม ทำให้การควบคุมกระบวนการหรือเครื่องจักรต่าง ๆ เหล่านั้นเป็นไปอย่างมีประสิทธิภาพ นอกจากนั้นยังสามารถเปลี่ยนแปลงรูปแบบของการควบคุมและสามารถขยายขีดความสามารถให้สูงขึ้นได้

2.1 หลักการทำงานของเครื่องควบคุมที่โปรแกรมได้

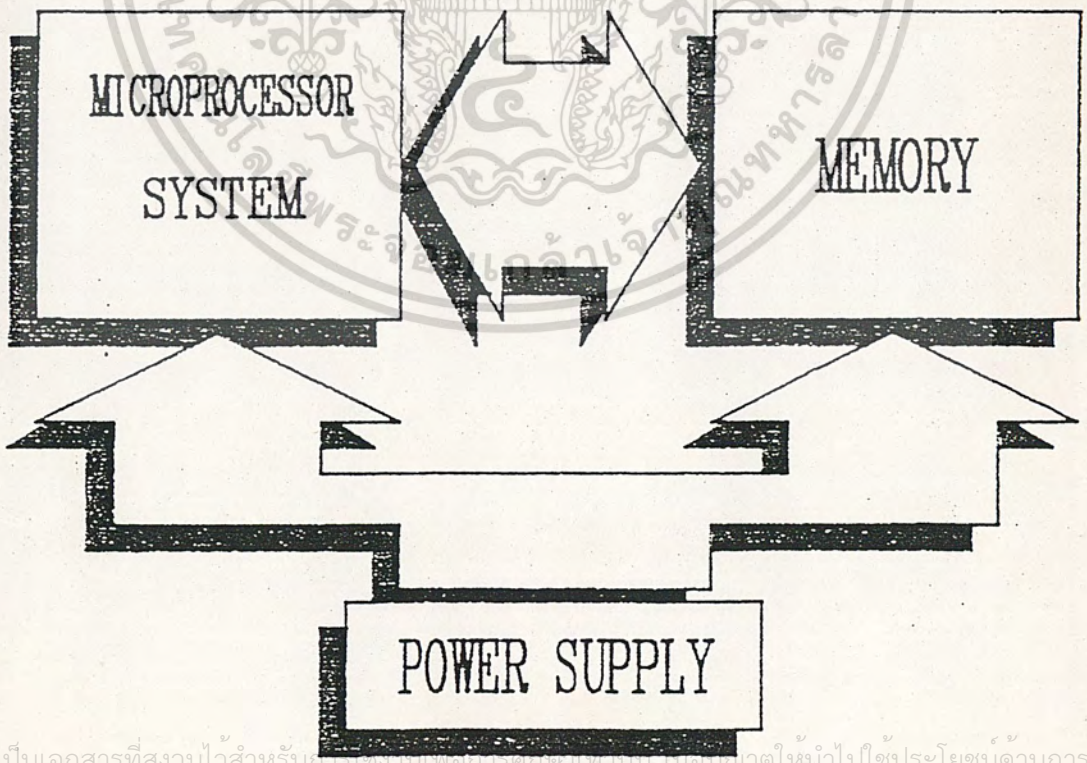
เครื่องควบคุมที่โปรแกรมได้ เป็นระบบคอมพิวเตอร์ที่อินพุทและเอาท์พุทได้ถูกออกแบบให้มีสภาวะเหมาะสมกับกระบวนการที่ต้องการควบคุม และมีหน่วยประมวลผลกลางที่ทำหน้าที่ควบคุมการทำงานของระบบทั้งหมด หน่วยประมวลผลกลางหมายถึงระบบไมโครโปรเซสเซอร์ที่ทำหน้าที่ควบคุมการทำงานทั้งหมด โปรแกรมที่ใช้ควบคุมการทำงานของหน่วยประมวลผลกลางดังกล่าวจะถูกแบ่งออกเป็น 2 ส่วน ส่วนหนึ่งคือโปรแกรมที่ใช้ในการกำหนดลำดับการทำงานของการควบคุม ซึ่งผู้ใช้สามารถที่จะเปลี่ยนแปลงหรือแก้ไขโปรแกรมในส่วนนี้ได้ ซึ่งโปรแกรมในส่วนนี้ก็คือโปรแกรมที่บ่งบอกเงื่อนไขต่าง ๆ ที่ใช้สำหรับการควบคุม โปรแกรมอีกส่วนหนึ่งคือโปรแกรมที่ใช้ในการควบคุมระบบ โปรแกรมส่วนนี้ จะทำหน้าที่ควบคุมการรับค่าสภาวะอินพุทซึ่งได้มาจากกระบวนการแล้วนำมาประมวลผลทางลอจิก คำนวณทางคณิตศาสตร์ เพื่อให้ข้อมูลอยู่ในรูปแบบที่เหมาะสมและส่งค่าสภาวะออกทางเอาท์พุท ตามลำดับขั้นตอนของโปรแกรมในส่วนแรก โครงสร้างของเครื่องควบคุมแสดงดังรูปที่ 2.1

2.1.1 หน่วยประมวลผลกลาง (CENTRAL PROCESSING UNIT : CPU)

หน่วยประมวลผลกลางประกอบด้วย ระบบไมโครโปรเซสเซอร์ซึ่งทำหน้าที่ในการประมวลผลข้อมูล หน่วยความจำ สำหรับ เก็บโปรแกรมของผู้ใช้ และหน่วยจ่ายกำลังไฟฟ้าดังรูปที่ 2.2



รูปที่ 2.1 โครงสร้างของเครื่องควบคุมที่โปรแกรมได้



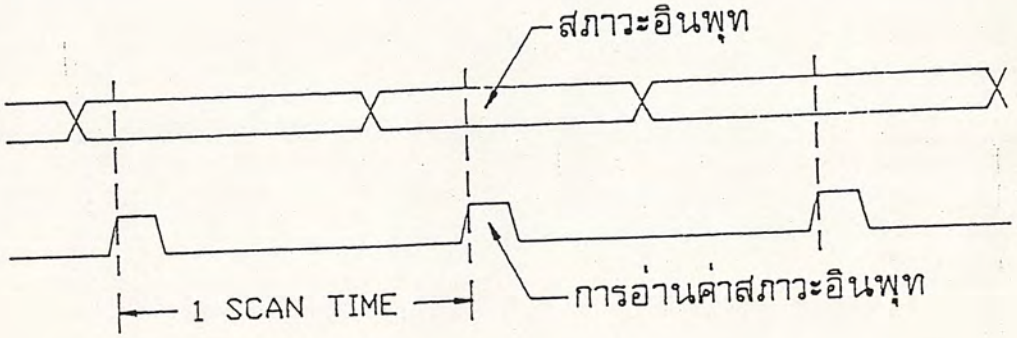
(CPU.DWG)

รูปที่ 2.2 โครงสร้างของหน่วยประมวลผลกลาง

ในเครื่องควบคุมบางแบบอาจจะรวมหน่วยประมวลผลกลาง หน่วยความจำและหน่วยจ่ายกำลังไฟฟ้าเข้าไว้ด้วยกัน เพื่อให้มีขนาดเล็กกระทัดรัด แต่บางแบบก็อาจจะแยกกันออกเป็นส่วน ๆ เพื่อสะดวกในการบำรุงรักษา ระบบไมโครโปรเซสเซอร์ซึ่งทำหน้าที่ในการประมวลผลเป็นส่วนประกอบที่สำคัญของเครื่องควบคุมมีหน้าที่ในการประมวลผลข้อมูล การคำนวณทางคณิตศาสตร์และทางลอจิก รวมทั้งควบคุมการทำงานของส่วนต่าง ๆ ทั้งหมด

ในปัจจุบันเทคโนโลยี ทางด้านไมโครโปรเซสเซอร์ ได้พัฒนาขึ้นมาก จึงได้มีการนำเอาไมโครโปรเซสเซอร์หลาย ๆ ตัวมาทำงานร่วมกัน (MULIT-PROCESSOR) เพื่อให้มีการประมวลผล ได้รวดเร็วและมีประสิทธิภาพยิ่งขึ้น โดยแยกการทำงาน ของไมโครโปรเซสเซอร์แต่ละตัว เป็นอิสระต่อกันแต่มี การแลกเปลี่ยนข้อมูลกันอยู่ตลอดเวลา นอกจากนี้ ในหน่วยอินพุทหรือเอาท์พุท บางชนิดก็จะมี ไมโครโปรเซสเซอร์ แยกอิสระจากเครื่องควบคุม อีกทีหนึ่ง เช่น หน่วยอินพุทเอาท์พุทแบบ PID ซึ่งสามารถทำงานได้ด้วยตัวเอง

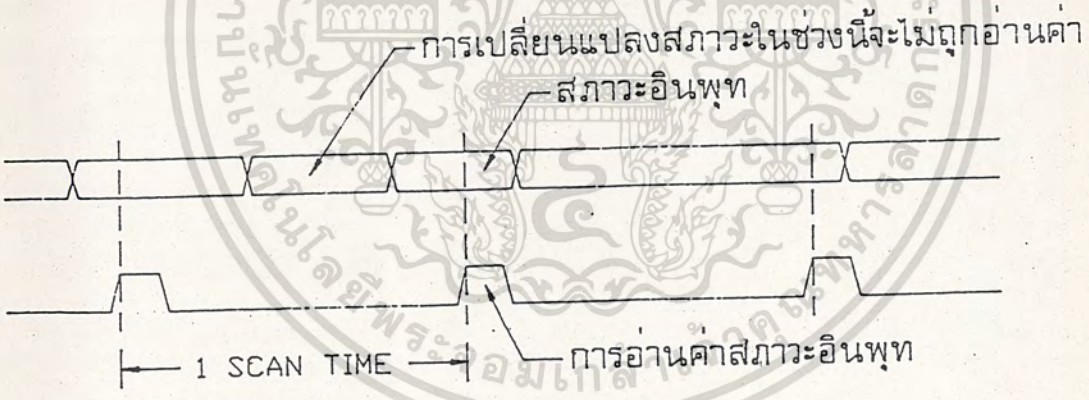
ค่าสภาวะต่าง ๆ ถูกนำมาจากกระบวนการโดยผ่านทางหน่วยอินพุท ผ่านการประมวลผลจากนั้นผลที่ได้จะถูกส่งออกทางเอาท์พุท เพื่อไปควบคุมการทำงาน ของกระบวนการตามต้องการ และจะย้อนกลับมารับค่าสภาวะ จากกระบวนการ เพื่อนำไปประมวลผลและส่งผลออกทางเอาท์พุทอีก จะมีการทำอย่างนี้ตลอดไป การรับค่าสภาวะจากภายนอกมาทำการประมวลผล และส่งค่าออกทางเอาท์พุท ในแต่ละครั้งเรียกว่า การสแกน (SCANNING) หรือหนึ่งรอบของการทำงาน ระยะเวลาในการสแกนหนึ่งรอบ (1 SCAN TIME) จะมีค่าเท่ากับระยะเวลาของการอ่านค่าสภาวะอินพุทครั้งแรกจนกระทั่งมีการอ่านค่าสภาวะอินพุทครั้งต่อไป ดังรูปที่ 2.3



(SCANTIME.DWG)

รูปที่ 2.3 ระยะเวลาการสแกนหนึ่งรอบ

ระยะเวลาในการสแกนจะขึ้นอยู่กับความยาวของโปรแกรมที่ผู้ใช้งานโปรแกรมเงื่อนไขการทำงานเข้าไปเพื่อควบคุม ซึ่งผู้ผลิตมักจะกำหนดไว้ว่าเครื่องควบคุมนั้น ๆ สามารถทำโปรแกรมได้สูงสุดเพียงใด และการใช้ชุดคำสั่งในการโปรแกรมนั้นแต่ละคำสั่งใช้เวลาในการประมวลผลเท่าใด นอกจากนั้นก็จะขึ้นอยู่กับสมรรถนะของเครื่องควบคุมเอง รวมทั้งอุปกรณ์ภายนอกที่ติดต่ออยู่กับเครื่องควบคุม ดังนั้นระยะเวลาในการสแกนหนึ่งรอบจึงเป็นตัวประกอบส่วนหนึ่งที่จะใช้ ในการตรวจสอบประสิทธิภาพของเครื่องควบคุม เนื่องจากการควบคุมกระบวนการจะต้องกระทำได้อย่างรวดเร็ว และแม่นยำ อย่างไรก็ตามถ้าเครื่องควบคุมมีค่าระยะเวลาการสแกนที่มากเกินไป อาจจะทำให้การควบคุมเกิดการผิดพลาดขึ้นได้ เพราะค่าสภาวะบางค่าอาจจะมีการเปลี่ยนแปลงที่เร็วมาก ๆ ซึ่งถ้าการเปลี่ยนแปลงนี้มีค่าเวลาสั้นกว่าระยะเวลาของการแสกนแล้วจะทำให้ข้อมูลที่ได้รับ เพื่อใช้ในการประมวลผิดพลาดได้ ดังรูปที่ 2.4



(MISSING.DWG)

รูปที่ 2.4 ความผิดพลาดที่เกิดขึ้นเมื่อค่าสภาวะอินพุท
มีระยะเวลาการเปลี่ยนแปลงสั้นกว่าระยะเวลาการสแกน

แต่ในทางปฏิบัติแล้ว การเปลี่ยนแปลงค่าสภาวะอินพุทของกระบวนการมักจะมีระยะเวลาที่ยาวนานกว่าระยะเวลาการสแกนของเครื่องควบคุม เนื่องจากอุปกรณ์ที่เป็นอินพุทส่วนใหญ่จะเป็นอุปกรณ์ไฟฟ้าเชิงกล หรือสวิทช์ชนิดต่าง ๆ ซึ่งมีค่าระยะเวลาการเปลี่ยนแปลงที่ค่อนข้างมาก นอกจากจะเป็นค่าสภาวะที่ได้รับมาจากอุปกรณ์อิเลคทรอนิคส์ซึ่งมีความเร็วในการสวิทช์สูง ในการใช้งานลักษณะนี้เครื่องควบคุมจำเป็นจะต้องมีอุปกรณ์พิเศษเพื่อช่วยให้การควบคุมเป็นไปอย่างถูกต้อง ซึ่งบางแบบอาจจะใช้วิธีอินเตอร์รัพท์ (INTERRUPT) เพื่อให้เครื่องควบคุมมาอ่านข้อมูลไปประมวลผลทันที หรือนำข้อมูลส่งออกไปเอาท์พุทในทันทีเป็นต้น

หน่วยความจำในหน่วยประมวลผลกลางจะถูกแบ่งออกเป็น 2 ส่วนใหญ่ ๆ คือ ส่วนแรกที่ใช้ในการเก็บโปรแกรมและข้อมูลสำหรับการควบคุมระบบ และอีกส่วนหนึ่งจะใช้ในการเก็บโปรแกรมของผู้ใช้ และข้อมูลสภาวะอินพุทเอาท์พุท ในเครื่องควบคุมจะมีหน่วยความจำที่มีความจุสูง แต่จะมีราคาแพงและเครื่องควบคุมขนาดเล็กก็จะมีความจุของหน่วยความจำต่ำซึ่งอาจจะไม่เพียงพอต่อการใช้งาน ดังนั้นการเลือกใช้เครื่องควบคุมจะต้องคำนึงถึงสิ่งเหล่านี้ด้วย

การจัดหน่วยพื้นที่ความจำของหน่วยประมวลผลกลางจะแบ่งออกเป็น 4 ส่วน ดังนี้

1. โปรแกรมบริหารงานระบบ (OPERATING SYSTEM)
2. พื้นที่ข้อมูลสำหรับประมวลผล (PROCESSOR WORKING AREA)
3. ตารางข้อมูล (DATA TABLE)
4. โปรแกรมผู้ใช้ (USER PROGRAM)

พื้นที่หน่วยความจำในส่วนที่ 1 และ 2 จะใช้วางโปรแกรมและข้อมูลสำหรับการควบคุมระบบและเป็นพื้นที่ใช้งานของไมโครโปรเซสเซอร์ พื้นที่ในส่วนที่ 3 และ 4 เป็น ส่วนที่ผู้ใช้สามารถเข้าถึงได้ทั้งการอ่านและการเขียน สำหรับส่วนที่ 3 เป็นตารางข้อมูลซึ่งเก็บค่าอินพุท เอาท์พุท รีเลย์ภายใน (INTERNAL RELAY) และค่ารีจิสเตอร์ (REGISTER) และส่วนที่ 4 เป็นโปรแกรมผู้ใช้ซึ่งเป็นชุดคำสั่งสำหรับการควบคุม

การเก็บข้อมูลอินพุทจะเก็บอยู่ในลักษณะเป็นตาราง (TABLE) มีขนาดเท่ากับขนาดของหน่วยอินพุท เช่นเครื่องควบคุมที่มีอินพุทสูงสุด 64 อินพุทก็จะมีตารางของอินพุทที่มีขนาด 64 ช่องและค่าในตารางนี้ก็จะมีค่าเช่นเดียวกับค่าสภาวะที่เครื่องควบคุมอ่านเข้ามาได้ ตารางเอาท์พุทก็มีลักษณะเช่นเดียวกัน สำหรับตารางรีเลย์ภายในนั้นเป็นเอาท์พุทแต่ไม่มีการเชื่อมต่อกับอุปกรณ์ภายนอก เนื่องจากเป็นตำแหน่งในหน่วยความจำเท่านั้นซึ่งจะใช้ในการเก็บค่าสภาวะ

ต่าง ๆ ชั่วคราว ส่วนตารางรีจิสเตอร์นั้นใช้ในการเก็บค่าและค่าสภาวะของตัวตั้งเวลา ตัวนับ และฟังก์ชั่นพิเศษต่าง ๆ

โปรแกรมผู้ใช้เป็นโปรแกรมที่ผู้ใช้เขียนนั้น เพื่อให้เครื่องควบคุมมีการทำงานตามที่กำหนดโดยมีเงื่อนไขตามสภาวะของอินพุท โปรแกรมนี้จะสามารถเปลี่ยนแปลงได้โดยผู้ใช้เป็นผู้กำหนด สำหรับในกรณีที่ไม่ต้องการเปลี่ยนแปลงโปรแกรมบ่อย ๆ ก็อาจจะใช้วิธีเก็บโปรแกรมดังกล่าวลงในรอม (ROM : READ ONLY MEMORY) ก็ได้

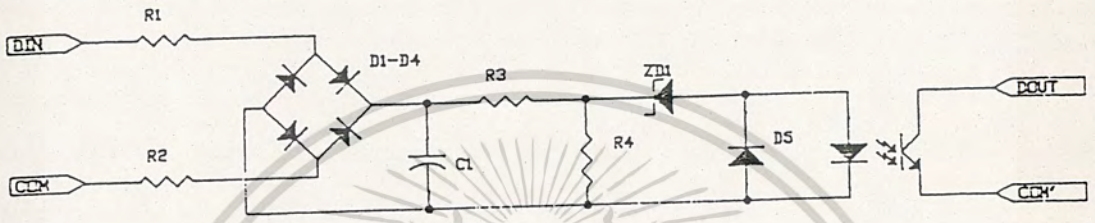
สำหรับแหล่งจ่ายกำลังไฟฟ้าทำหน้าที่จ่ายกระแสไฟฟ้าให้กับส่วนต่าง ๆ ของเครื่องควบคุม แหล่งจ่ายกำลังไฟฟ้าจะต้องถูกออกแบบเป็นพิเศษเพื่อให้จ่ายแรงดันได้คงที่ตลอดช่วงการทำงานแม้ว่าแรงดันทางด้านอินพุทจะมีการเปลี่ยนแปลงก็ตาม และต้องสามารถที่จะป้องกันสัญญาณรบกวนที่เข้ามาได้เป็นอย่างดี เนื่องจากเครื่องควบคุมส่วนใหญ่จะนำไปใช้กับกระบวนการทางอุตสาหกรรมซึ่งมีสัญญาณรบกวนในสายส่งกำลังเป็นอันมาก และในกรณีที่มีสัญญาณรบกวนในสายส่งสัญญาณกำลังสูงหรือแรงดันไม่คงที่ หน่วยจ่ายกำลังไฟฟ้าจะไม่สามารถทำงานได้ตามปกติ จึงจำเป็นจะต้องมีการปรับสภาพของแรงดันไฟฟ้าให้คงที่เสียก่อน ส่วนอุปกรณ์ที่ใช้ในการปรับแรงดันไฟฟ้า หรือลดสัญญาณรบกวนในลักษณะนี้มีอยู่ด้วยกันหลายชนิดบางชนิดสามารถลดหรือกำจัดสัญญาณรบกวนได้เพียงอย่างเดียว บางชนิดก็สามารถปรับแรงดันไฟฟ้าให้คงที่ได้ด้วย หรือบางชนิดอาจจะมีคุณสมบัติพิเศษในการกำจัดฮาร์โมนิค (HARMONIC) ได้เป็นต้น

2.1.2 หน่วยอินพุท เอาท์พุท (INPUT/OUTPUT UNIT)

ส่วนนี้จะทำหน้าที่ติดต่อระหว่างเครื่องควบคุมกับกระบวนการ ซึ่งหน่วยอินพุทจะทำหน้าที่รับค่าสภาวะมาจากกระบวนการ เช่น สวิทช์ (SWITCH) ชนิดต่าง ๆ อุปกรณ์ตรวจจับ (SENSOR) และอุปกรณ์วัด ซึ่งค่าสภาวะต่าง ๆ เหล่านี้แบ่งออกเป็นแบบดิจิตอล (DIGITAL) และแบบอนาลอก (ANALOG) ทั้งสองชนิดก็มีการออกแบบพิเศษสำหรับงานเฉพาะอย่างดีมาก ซึ่งการเลือกใช้หน่วยอินพุทหรือเอาท์พุทก็จะต้องพิจารณาจากรูปแบบของข้อมูลหรือสภาวะที่ต้องการ การเลือกใช้หน่วยอินพุทเอาท์พุทที่เหมาะสมจะทำให้การควบคุมเป็นไปอย่างมีประสิทธิภาพ

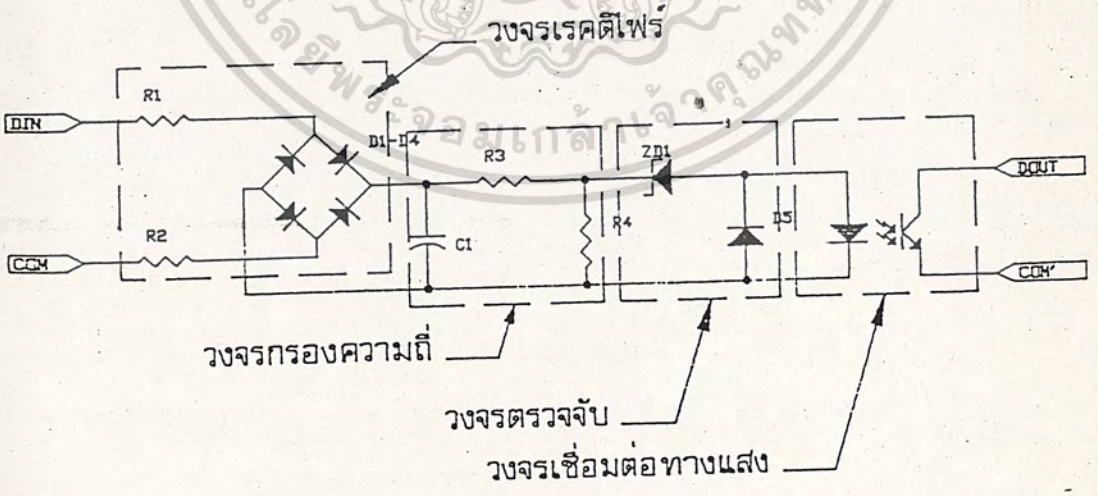
หน่วยอินพุทเอาท์พุทแบบดิจิตอล ทำหน้าที่ติดต่อกับอุปกรณ์ภายนอกที่มีการเปลี่ยนแปลงสภาวะเพียง 2 สภาวะเท่านั้น คือ 0 และ 1 อุปกรณ์ที่มีค่าสภาวะเหล่านี้เช่นสวิทช์ชนิดต่าง ๆ อุปกรณ์สวิทช์ชิ่งทางอิเลคทรอนิคส์ หรือในกรณีที่เป็นเอาท์พุทก็จะเป็นพวกมอเตอร์ โซลินอยด์

วาลว์ (SOLINOID VALVE) หลอดไฟฟ้า เป็นต้น สำหรับระดับสัญญาณของหน่วยอินพุทเอาท์พุท แบบดิจิตอลก็มักจะอยู่ในช่วง 0-24 โวลท์ หรือ 0-220 โวลท์ หรืออาจจะเป็นระดับแรงดันทีทีแอล (TTL) คือ 0-5 โวลท์ก็ได้ ซึ่งการกำหนดระดับการเปลี่ยนแปลงของลอจิก 0 และ 1 ก็เป็นไปตามการออกแบบของหน่วยอินพุทเอาท์พุทนั้น ๆ ดังตัวอย่างในรูปที่ 2.5



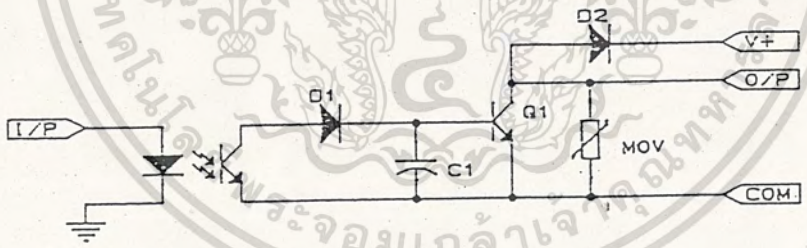
รูปที่ 2.5 วงจรอินพุทแบบดิจิตอล

จากวงจรในรูปที่ 2.5 เป็นวงจรพื้นฐานที่ใช้กันโดยทั่วไป วงจรดังกล่าวสามารถรับอินพุทได้ทั้งไฟสลับและไฟตรงส่วนขนาดแรงดันสูงสุดขึ้นอยู่กับการออกแบบ หลักการทำงานโดยทั่วไปอธิบายได้ตามผังภาพ (BLOCK DIAGRAM) ในรูปที่ 2.6

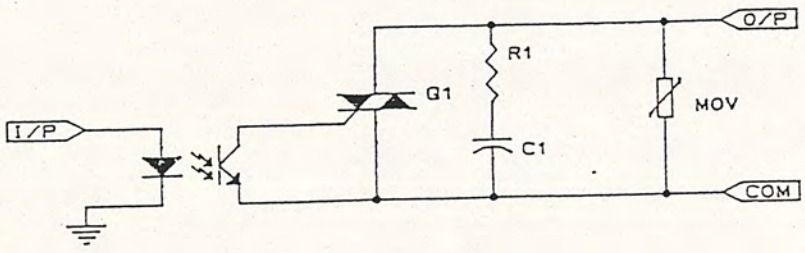


รูปที่ 2.6 ผังภาพของวงจรในรูปที่ 2.5

วงจรเรคติไฟร์ (RECTIFIER) $D_1 - D_4$ ทำหน้าที่เปลี่ยนสัญญาณไฟสลับให้เป็นไฟตรงหรือในกรณีที่เป็นไฟตรงก็สามารถผ่านได้ทันที ความต้านทาน R_1 และ R_2 ทำหน้าที่ในการลดแรงดันค่าของความต้านทานขึ้นอยู่กับแรงดันอินพุทสูงสุด ส่วนวงจรกรองความถี่ทำหน้าที่กรองสัญญาณรบกวนซึ่งอาจจะทำให้วงจรทำงานผิดพลาดได้ออก วงจรตรวจจับทำหน้าที่เปรียบเทียบระดับแรงดันเพื่อกำหนดสภาวะในการเป็น 0 หรือ 1 ค่าสภาวะดังกล่าวจะเป็นสภาวะในแบบดิจิตอลแล้ว จากนั้นก็จะส่งค่าสภาวะนี้ไปยังเครื่องควบคุมแต่จะใช้การเชื่อมต่อกันทางแสง (OPTICAL ISOLATOR) เพื่อป้องกันไม่ให้เครื่องควบคุมเสียหายอันเนื่องมาจากอันตรายจากแรงดันอินพุทสูงเกินกำหนดหรือเกิดการลัดวงจรขึ้นทางอินพุท นอกจากนี้หน่วยอินพุทแบบดิจิตอลอีกหลายชนิดซึ่งได้ออกแบบมาให้เหมาะสมกับกระบวนการทางอุตสาหกรรมบางชนิด เช่น หน่วยอินพุทแบบดิจิตอลที่มีแรงดันเป็นทีทีแอลเป็นต้น สำหรับหน่วยเอาท์พุทแบบดิจิตอลนั้นชนิดที่ใช้กันมากคือ เอาท์พุทที่เป็นคอนแทครีเลย์ (CONTACT RELAY) ส่วนเอาท์พุทชนิดอื่น ๆ เช่น ชนิดเอาท์พุทเป็นทรานซิสเตอร์ (TRANSISTOR) หรือเป็นหน่วยเอาท์พุทสำหรับไฟฟ้าสลับ ดังรูปที่ 2.7 และ 2.8



รูปที่ 2.7 วงจรเอาท์พุทแบบดิจิตอลชนิดทรานซิสเตอร์



รูปที่ 2.8 วงจรเอาท์พุทแบบดิจิตอลชนิดไฟสลับ

นอกจากที่ได้กล่าวมาทั้งหมดแล้วหน่วยอินพุทเอาท์พุทแบบดิจิตอลอีกชนิดหนึ่งเป็นการรับค่าส่งสภาวะเป็นกลุ่มของข้อมูล เช่น รหัส BCD หรือเป็น BINARY เป็นต้น หน่วยอินพุทเอาท์พุทเหล่านี้ได้ถูกออกแบบสำหรับการประมวลเชิงตัวเลขเพื่อการติดตั้งค่าเป็นตัวเลขหรือให้เครื่องควบคุมสามารถส่งข้อมูลออกมาในรูปของรหัสที่ต้องการ เป็นต้น

หน่วยอินพุทเอาท์พุทแบบอนาลอก ทำหน้าที่ติดต่อกับอุปกรณ์ภายนอกที่มีการส่งหรือรับค่าสภาวะ ซึ่งค่าสภาวะเหล่านี้จะเป็นค่าปริมาณทางไฟฟ้า เช่น อุณหภูมิ ความดัน ความชื้น อัตราการไหล หรือแรงดันไฟฟ้าเป็นต้น สำหรับทางเอาท์พุทก็มักจะเป็นระบบเซอร์โว (SERVO) หรือวาลว์ควบคุม (CONTROL VALVE) การรับส่งค่าสภาวะมักจะเป็นลูปกระแส (CURRENT LOOP) 4-20 มิลลิแอมแปร์ หรือแรงดัน 1-5 โวลท์

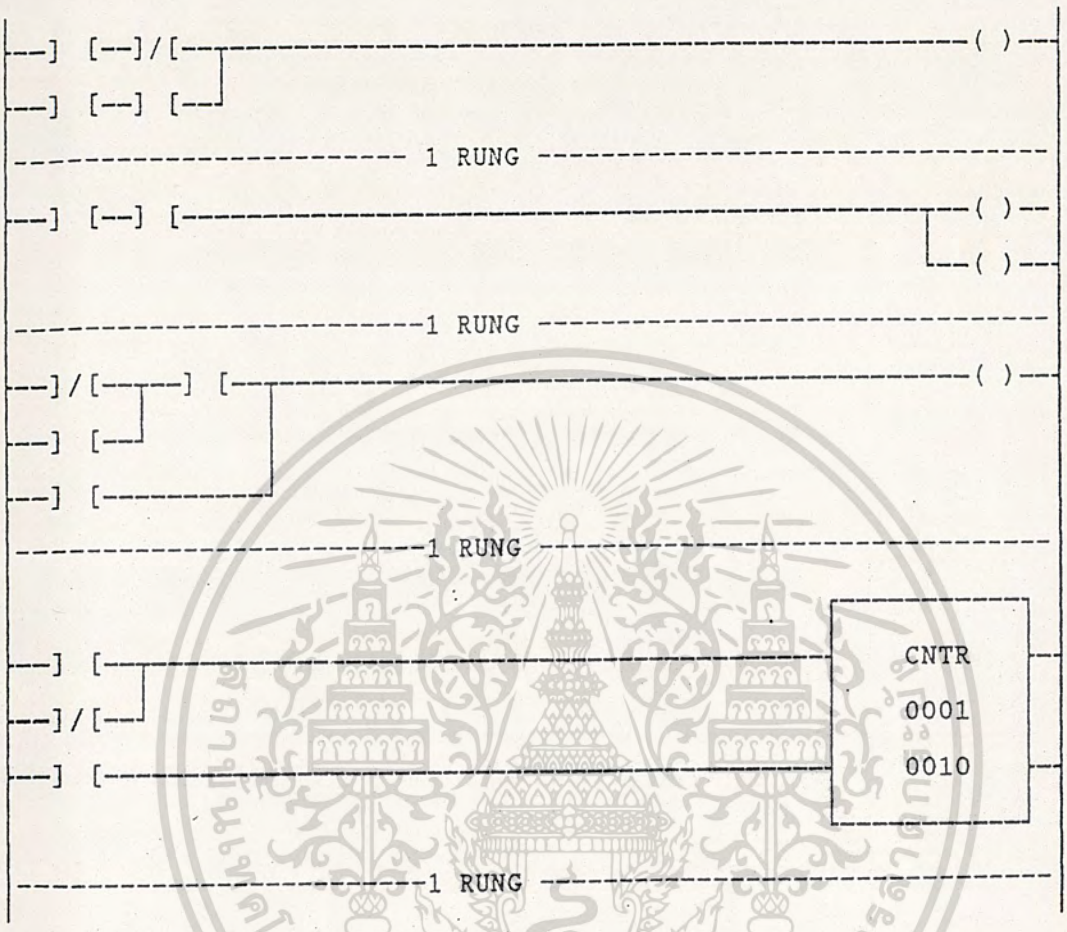
นอกจากในหัวข้อ 2.2.1 และ 2.2.2 แล้วหน่วยอินพุทเอาท์พุทยังมีอีกหลายชนิด เช่น หน่วยอินพุทชนิดความเร็วสูงสำหรับอุปกรณ์อินพุทที่มีความเร็วสูงจนหน่วยอินพุทของเครื่องควบคุมไม่สามารถตรวจจับได้ ตัวนับชนิดความเร็วสูงสำหรับการนับค่าจากอินพุทที่มีความเร็วสูงไม่สามารถใช้ตัวนับในเครื่องควบคุมได้ หรือหน่วยเชื่อมต่อคอมพิวเตอร์ (COMPUTER LINK) เพื่อใช้ในการรับส่งข้อมูลหรือโปรแกรมกับเครื่องคอมพิวเตอร์ด้วยพอร์ทมาตรฐานเช่น RS-422 RS-232 เป็นต้น อุปกรณ์เหล่านี้มักจะมีไมโครโปรเซสเซอร์เพื่อการประมวลผลเป็นอิสระมิฉะนั้นเครื่องควบคุมอาจจะทำงานผิดพลาดได้ และจะทำให้การควบคุมมีประสิทธิภาพสูงขึ้นด้วย

2.2 ภาษาที่ใช้ในการโปรแกรม

ภาษาที่ใช้ในการโปรแกรมเครื่องควบคุมที่โปรแกรมได้มีอยู่ด้วยกันหลายชนิด เช่น โปรแกรมภาษาคำสั่งแบบบูลลีน (BOOLEAN INSTRUCTION PROGRAMING) แบบแลดเดอร์ไดอะแกรม (LADDER DIAGRAM PROGRAMING) และอื่น ๆ ในเครื่องควบคุมส่วนใหญ่จะโปรแกรมด้วยภาษาคำสั่งแบบบูลลีน และแบบแลดเดอร์ไดอะแกรม

2.2.1 โปรแกรมภาษาคำสั่งแบบแลดเดอร์ไดอะแกรม

เป็นการโปรแกรมที่มีประสิทธิภาพเนื่องจากมีลักษณะคล้ายกับรีเลย์ไดอะแกรม (RELAY DIAGRAM) ดังนั้นการเขียนโปรแกรมจากรีเลย์ไดอะแกรมจึงทำได้ง่าย แลดเดอร์ไดอะแกรมจะแบ่งออกเป็นส่วน ๆ แต่ละส่วนคือเอาท์พุท 1 จุดหรือเอาท์พุทมากกว่า 1 จุด แต่เอาท์พุทเหล่านี้จะมีลอจิกเหมือนกัน แต่ละส่วนเหล่านี้เรียกว่า "รังค์ (RUNG)"



รูปที่ 2.9 การแบ่งแลดเดอร์ไดอะแกรมออกเป็นรังค์

การโปรแกรมด้วยแลดเดอร์ไดอะแกรมจะสะดวกและมีประสิทธิภาพมากแต่ข้อเสียของการโปรแกรมแบบนี้คือ อุปกรณ์ที่ใช้ในการเขียนโปรแกรมจะต้องมีหน่วยแสดงผลที่สามารถแสดงอักขระหรือไดอะแกรมได้ครั้งละหลาย ๆ บรรทัด ดังนั้นส่วนใหญ่แล้วอุปกรณ์ที่ใช้ในการโปรแกรมแบบนี้มักจะใช้จอซีอาร์ที (CRT : CATHODE RAY TUBE) เป็นหน่วยแสดงผล ทำให้มีขนาดใหญ่ไม่สะดวกในการทำงาน แต่ในปัจจุบันได้มีการออกแบบและพัฒนาหน่วยแสดงผลที่มีขนาดเล็กเพื่อความสะดวก แต่ก็ยังมีราคาแพงและขนาดของจอภาพก็ยังมีขนาดเล็กไม่สามารถแสดงแลดเดอร์ไดอะแกรมครั้งละหลาย ๆ บรรทัด บางครั้งก็อาจจะใช้คอมพิวเตอร์หรือ

ไมโครคอมพิวเตอร์ ทำการเชื่อมต่อกับเครื่องควบคุมเพื่อใช้ในการโปรแกรมด้วยแลดเดอร์ไดอะแกรม โดยคอมพิวเตอร์ทำหน้าที่ในการสร้างและแก้ไขแลดเดอร์ไดอะแกรมแล้วทำการแปลเป็นชุดคำสั่งของเครื่องควบคุมก่อน จากนั้นก็จะส่งชุดคำสั่งดังกล่าวมายังเครื่องควบคุมเพื่อให้ทำงานตามที่ต้องการต่อไป นอกจากนั้นแล้วขณะที่เครื่องควบคุมทำงานก็สามารถตรวจสอบการทำงานได้โดยการแสดงผล ซึ่งจะแสดงเป็นแลดเดอร์ไดอะแกรมและเมื่ออินพุทหรือเอาท์พุทตัวใดมีการเปลี่ยนแปลงสภาวะ ก็จะมีการเปลี่ยนสีตามสภาวะของการเปลี่ยนแปลงของอินพุทหรือเอาท์พุทตัวนั้น ๆ

2.2.2 โปรแกรมภาษาคำสั่งแบบบูลลีน

การโปรแกรมชนิดนี้มีลักษณะคล้ายกับสัญญลักษณ์ของพีชคณิตบูลลีน เช่น

LD	0010
OR	0011
AND NOT	0012
OUT	1000

นอกจากจะเป็นสัญญลักษณ์ของพีชคณิตแล้วยังมีการใช้ฟังก์ชั่นพิเศษต่าง ๆ อีกมากเช่น ตัวนับ ตัวตั้งเวลา เป็นต้น

นอกจากการโปรแกรมทั้งสองแบบที่กล่าวมาแล้วก็ยังมีอีกวิธีหนึ่งที่มีใช้กันอยู่พอสมควร คือการใช้ภาษาชั้นสูง (HIGH LEVEL LANGUAGE) คือการใช้ภาษาในการโปรแกรมคอมพิวเตอร์เช่น ภาษาปาสคาล (PASCAL) ภาษาฟอร์แทรน (FORTRAN) เป็นต้น แต่การใช้วิธีนี้มีข้อจำกัดอยู่ว่าจะทำงานได้ช้า และวิธีนี้ก็เหมาะสำหรับการโปรแกรมที่มีการทำงานซับซ้อนมากหรือมีการคำนวณทางวิทยาศาสตร์ดังนั้นวิธีนี้มักจะใช้กับเครื่องควบคุมหรือระบบควบคุมกระบวนการขนาดใหญ่ วิธีที่มีประสิทธิภาพและใช้กันอย่างแพร่หลายโดยทั่วไปก็คือการใช้แลดเดอร์ไดอะแกรมเนื่องจากแลดเดอร์ไดอะแกรมใกล้เคียงกับรีเลย์ไดอะแกรมมากที่สุด ตลอดจนการตรวจสอบการทำงานของโปรแกรมหรือการควบคุมก็ทำได้โดยสะดวก ส่วนฟังก์ชั่นทางคณิตศาสตร์นั้นสามารถนำมาใช้ร่วมกับแลดเดอร์ไดอะแกรมได้โดยง่าย ซึ่งรายละเอียดจะได้กล่าวต่อไป

บทที่ 3
โครงสร้างของเครื่องควบคุมที่โปรแกรมได้

บทนี้จะกล่าวถึงคุณลักษณะของโครงสร้างทางฮาร์ดแวร์ของเครื่องควบคุมที่โปรแกรมได้ ที่นำเอาบอร์ด ANT -32 ซึ่งเป็นระบบไมโครคอนโทรลเลอร์มาใช้เป็นหน่วยประมวลผลกลางต่อร่วมกับหน่วยอินพุทเอาท์พุทแบบต่าง ๆ หน่วยเชื่อมต่อกับคอมพิวเตอร์เพื่อใช้สำหรับทำการรับส่งข้อมูลหรือโปรแกรมหรือแสดงค่าสภาวะขณะทำการควบคุม โดยแบ่งอธิบายเป็นหัวข้อดังนี้

- รู้จักกับ ANT-32
- หน่วยประมวลผลกลาง
- หน่วยอินพุทเอาท์พุท
- หน่วยเชื่อมต่อกับเครื่องไมโครคอมพิวเตอร์

3.1 รู้จักกับ ANT-32

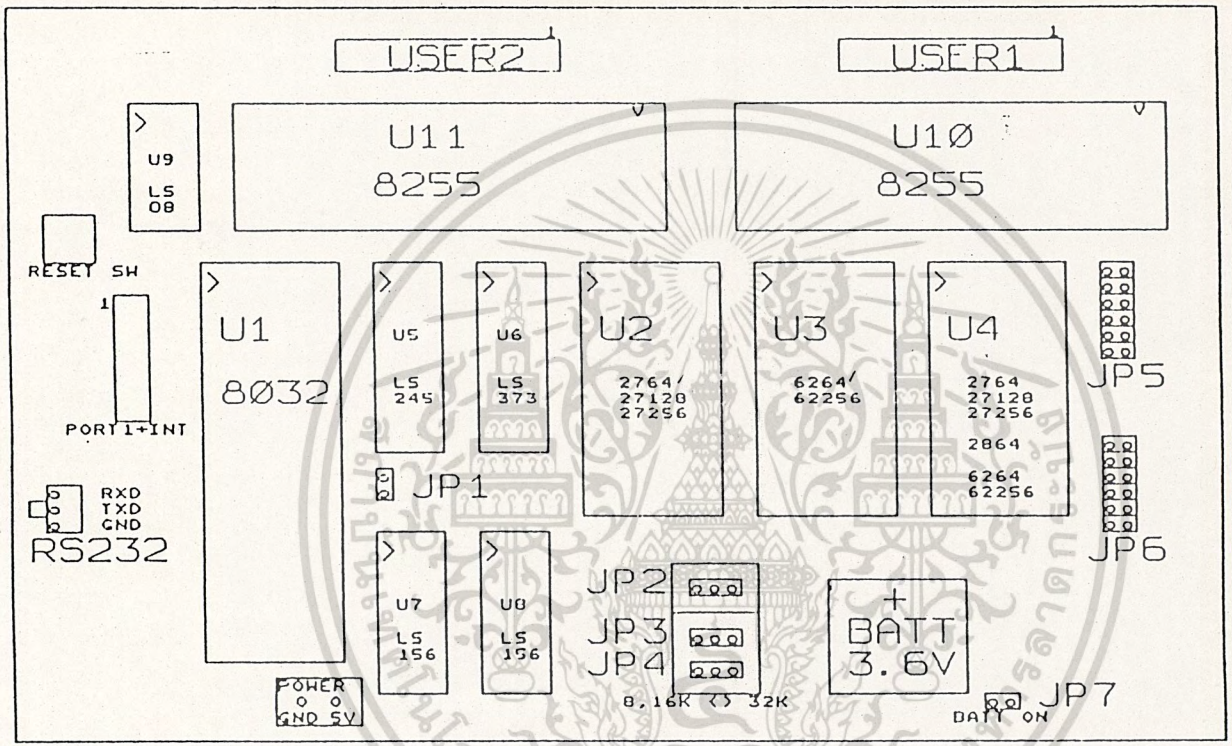
ANT-32 คือบอร์ดไมโครคอนโทรเลอร์ขนาดเล็ก ที่สามารถใช้ทั้ง ภาษาเบสิก และแอสเซมบลี้ สำหรับการพัฒนาระบบควบคุมอัตโนมัติ และใช้งานในลักษณะเป็น EMBEDDED CONTROLLER โดยใช้ชิพไมโครคอนโทรลเลอร์ เบอร์ 8032 ซึ่งอยู่ในตระกูล MCS-51 ของอินเทล ทำหน้าที่เป็น CPU

ในการพัฒนาโปรแกรม ANT-32 สามารถใช้เครื่องคอมพิวเตอร์ PC ทั่วไป เป็นเครื่องมือช่วยในการพัฒนา ซึ่งจะทำให้งานสะดวกเป็นอย่างมาก มีความรวดเร็วและง่ายต่อการทดสอบแก้ไข สามารถมองเห็นโปรแกรม และข้อมูลในหน่วยความจำบนจอภาพ และยังสามารถเขียนโปรแกรมสั่งงานภาษาเบสิก ด้วยคีย์บอร์ดของเครื่องคอมพิวเตอร์ PC ทั้งนี้การพัฒนาโปรแกรมจะกระทำโดยผ่านพอร์ตสื่อสารอนุกรม RS232C โดยที่เครื่องคอมพิวเตอร์ PC จะใช้ COMMUICATION SOFTWARE ทั่วไปเช่น PROCOM, CROSSTALK จัดการให้เครื่องคอมพิวเตอร์ PC ทำหน้าที่เสมือนเป็น จอภาพ และคีย์บอร์ด ของ ANT-32

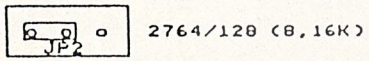
ANT-32 มีภาษาเบสิค เป็น MONITOR PROGRAM ทำให้สามารถเขียน แก้ไข หรือทดสอบโปรแกรมภาษาเบสิก ซึ่งเป็นภาษาระดับสูง ง่ายต่อการเรียนรู้ และใช้งานอีกทั้งใน BASIC MONITOR นี้ ยังสามารถทำการ DOWNLOAD และ UPLOAD ข้อมูลหรือโปรแกรมภาษาเบสิก ระหว่าง ANT-32 และเครื่องคอมพิวเตอร์ PC ทั่วไปได้

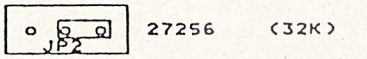
ANT-32 SPECIFICATION

CPU :	8032
CPU CLOCK :	11.0582 MHz
INTERNAL MEMORY :	256 BYTE (RAM)
EXT.CODE MEMORY :	(U2) 8-32K SELECT 2764,27128,27256(ROM)
EXT.DATA MEMORY :	(U3) 8-32K SELECT 6264,62256 (RAM)
EXT.CODE & DATA MEMORY :	(U4) 8-30K SELECT 2764,27128,27256 (ROM)
	2864 (EEPROM)
	6264, 62256 (RAM)
INTERNAL PORT :	12 BIT I/O
EXTERNAL PORT :	USER1 8255 PORT I/O 24 BIT
	USER2 8255 PORT I/O 24 BIT
BACKUP :	DATA MEMORY (U3) 52 HOUR
CHARGE TIME :	48 HOUR
LANGUAGE :	MCS BASIC-52
	ASSEMBLY (BY DOWNLOAD HEX FILE)
CONNECTOR :	16P INTERNAL PORT
	26P USER1 PORT
	26P USER2 PORT
	2P POWER SUPPLY
	3P SERIAL INTERFACE (RS232)
SERIAL INTERFACE :	RS232C
POWER :	5V DC 290mA (U4 NOT INCLUDE)
SIZE :	5.25" x 3.9"
OPTION :	BATTERY NI-CAD 3.6V 60mA

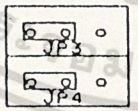


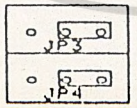
SELECT U2 BY JP2





SELECT U3 BY JP3,JP4



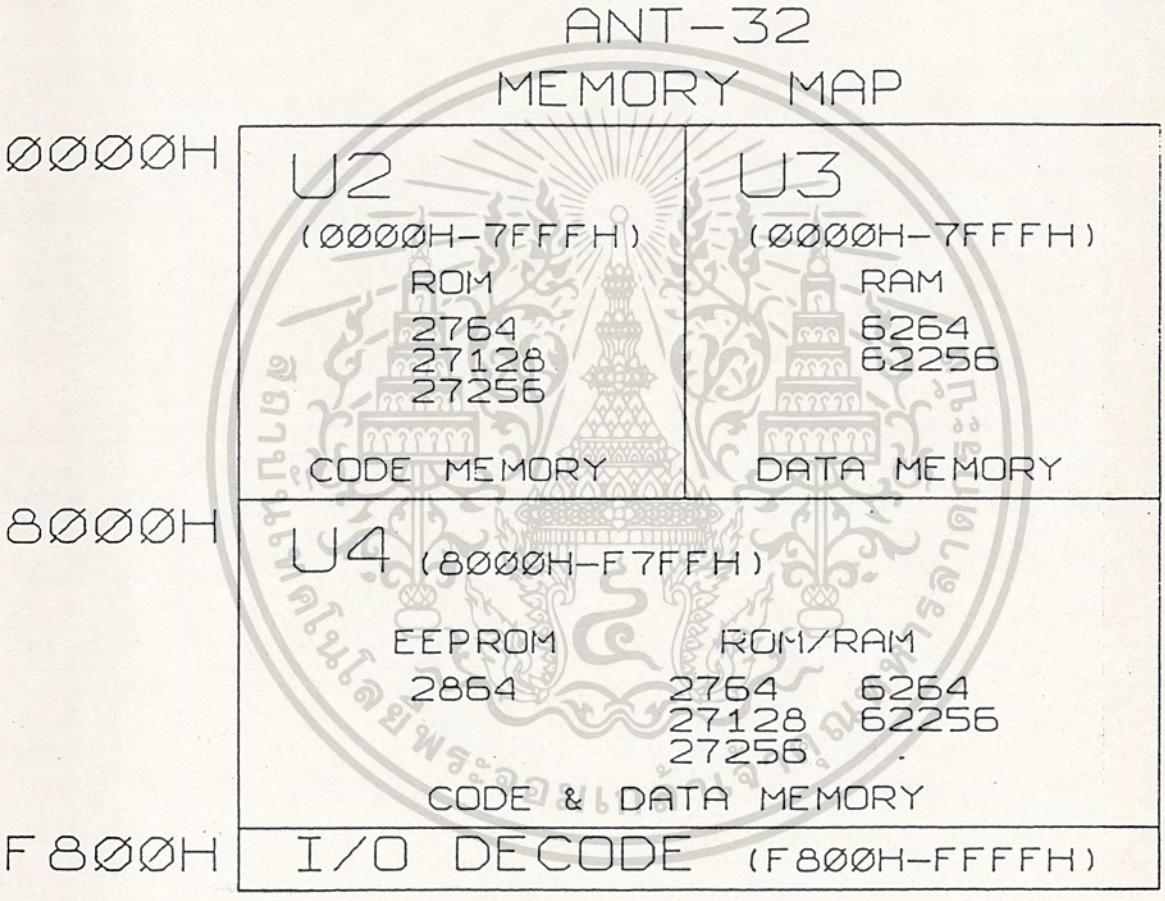


SELECT U4 BY JP5,JP6

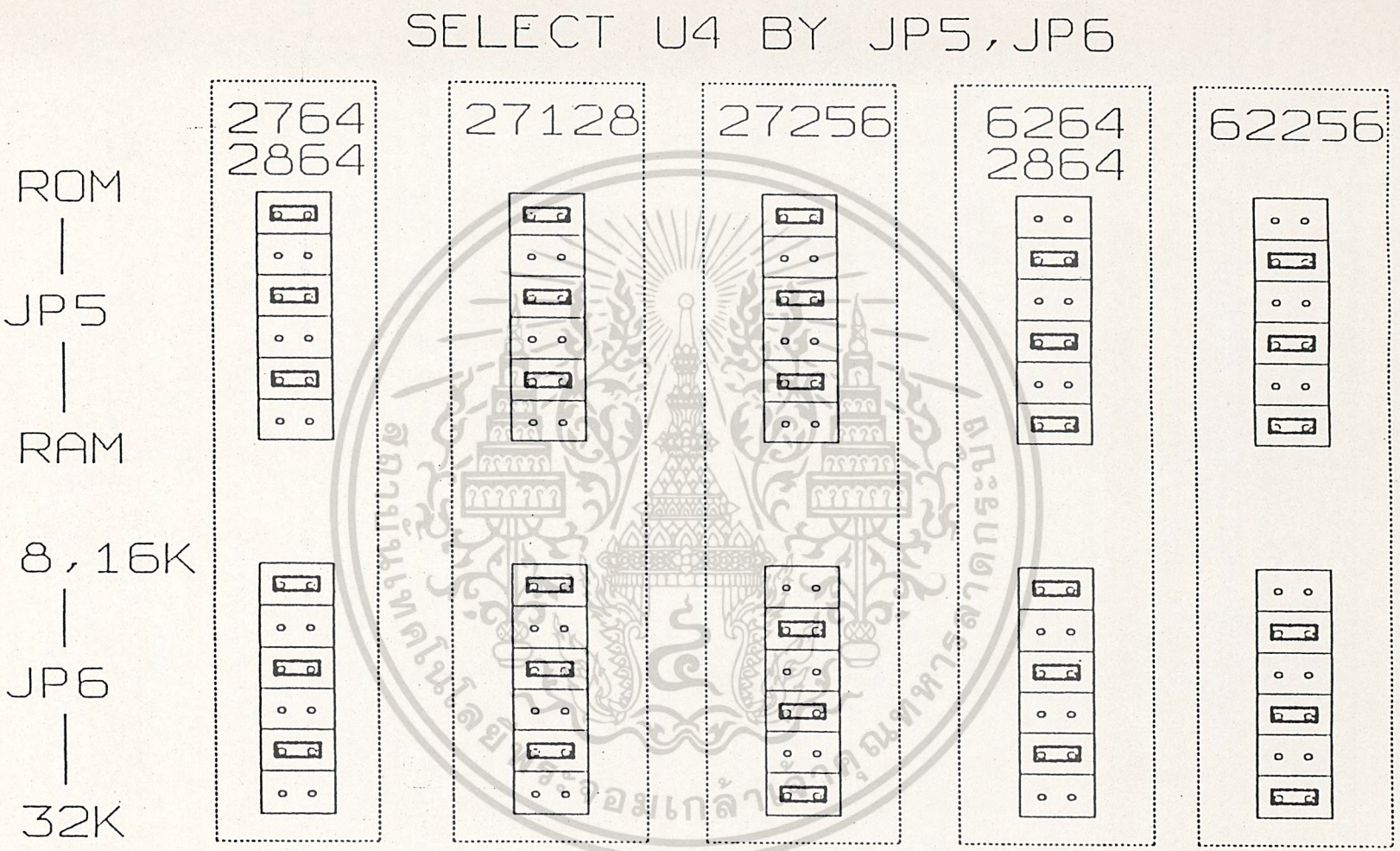
SEE NEXT PAGE

ANT-32 LAYOUT		
Size A	Document Number SILA RESEARCH CO.,LTD.	REV 001
Date: July 10, 1990	Sheet 1 of 2	

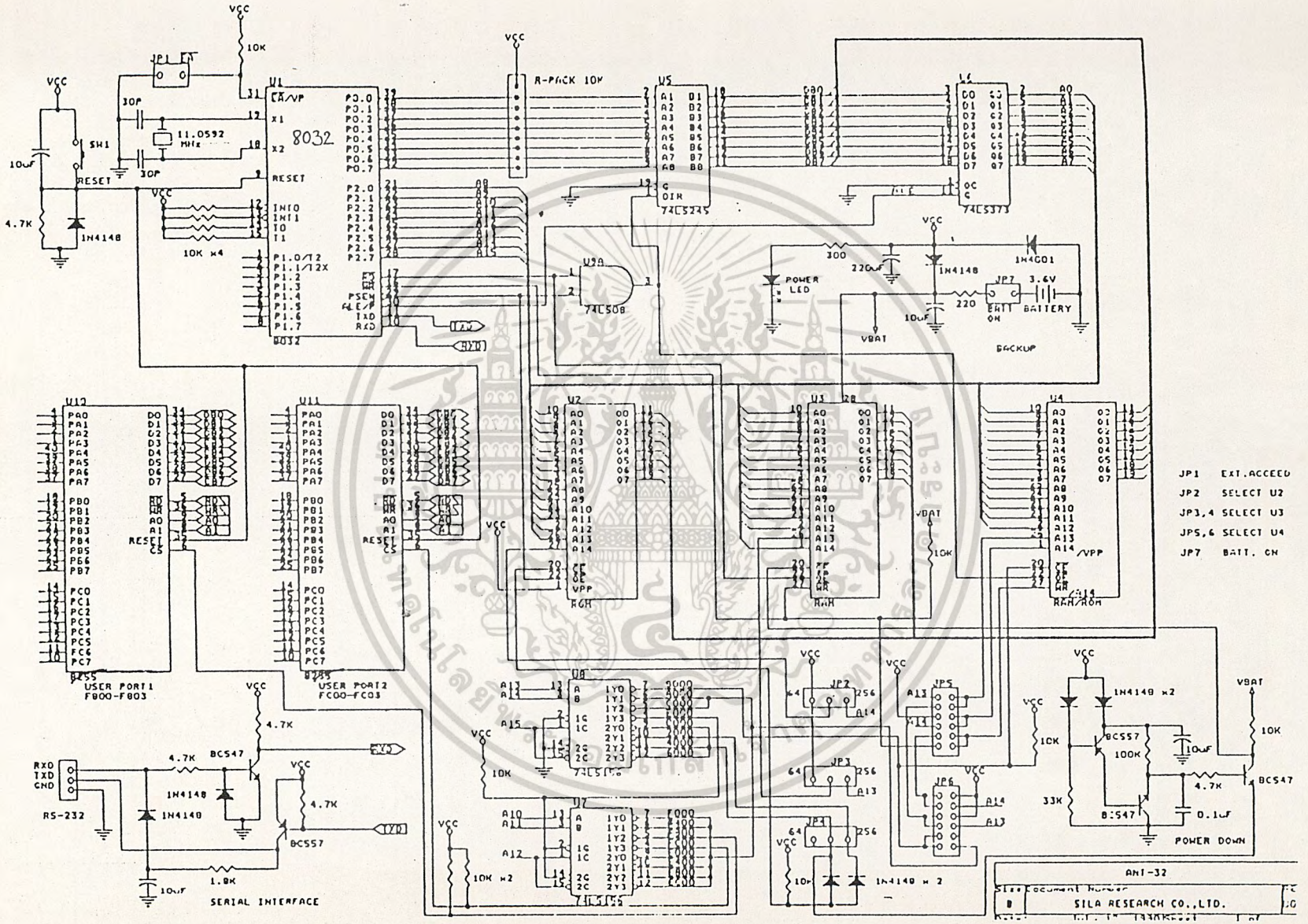
รูปที่ 3.1



รูปที่ 3.2


SELECT U4 BY JP5,JP6
ROM
JP5
RAM
8,16K
JP6
32K
2764
2864
27128
27256
6264
2864
62256
ANT-32 LAYOUT
Size
A
Document Number
SILA RESEARCH CO.,LTD.
REV
001
Date:
July 10, 1990
Sheet
2 of
2

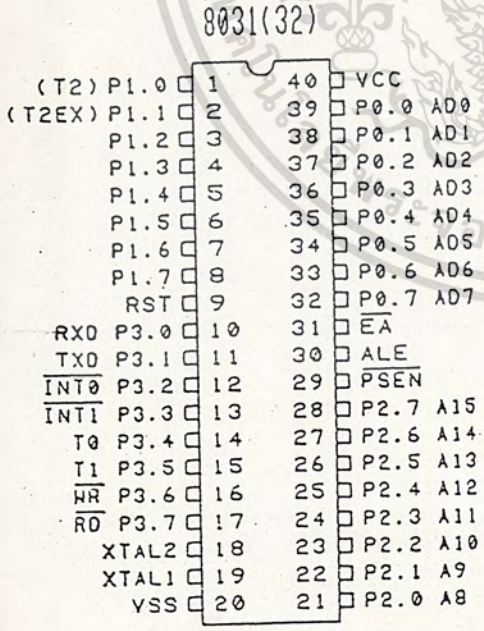
รูปที่ 3.3


8032
74LS245
74LS373
74LS08
R-PACK 10K
RESET
SW1
11.0592 MHz
30P
10K x4
4.7K
1N4148
POWER LED
220
BATTERY
BACKUP
USER PORT1
F800-F803
USER PORT2
FC00-FC03
RS-232
SERIAL INTERFACE
BC547
BC557
1.8K
10K x2
1N4148 x2
100K
33K
0.1uF
POWER DOWN
JP1 EXT.ACCEED
JP2 SELECT U2
JP3,4 SELECT U3
JP5,6 SELECT U4
JP7 BATT. ON
ANT-32
SILA RESEARCH CO.,LTD.

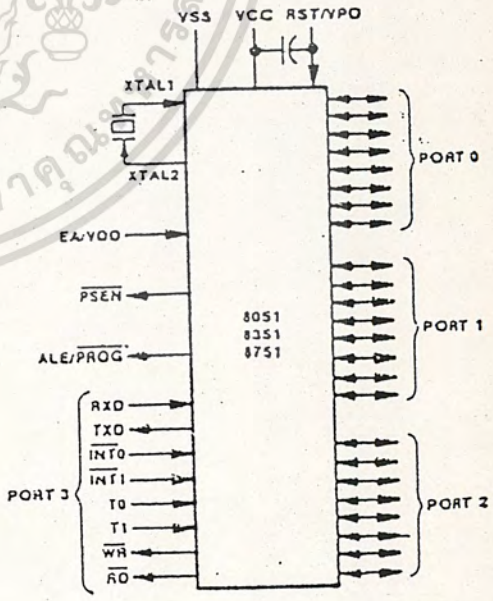
รูปที่ 3.4

3.2 หน่วยประมวลผลกลาง (CENTRAL PROCESSING UNIT : CPU)

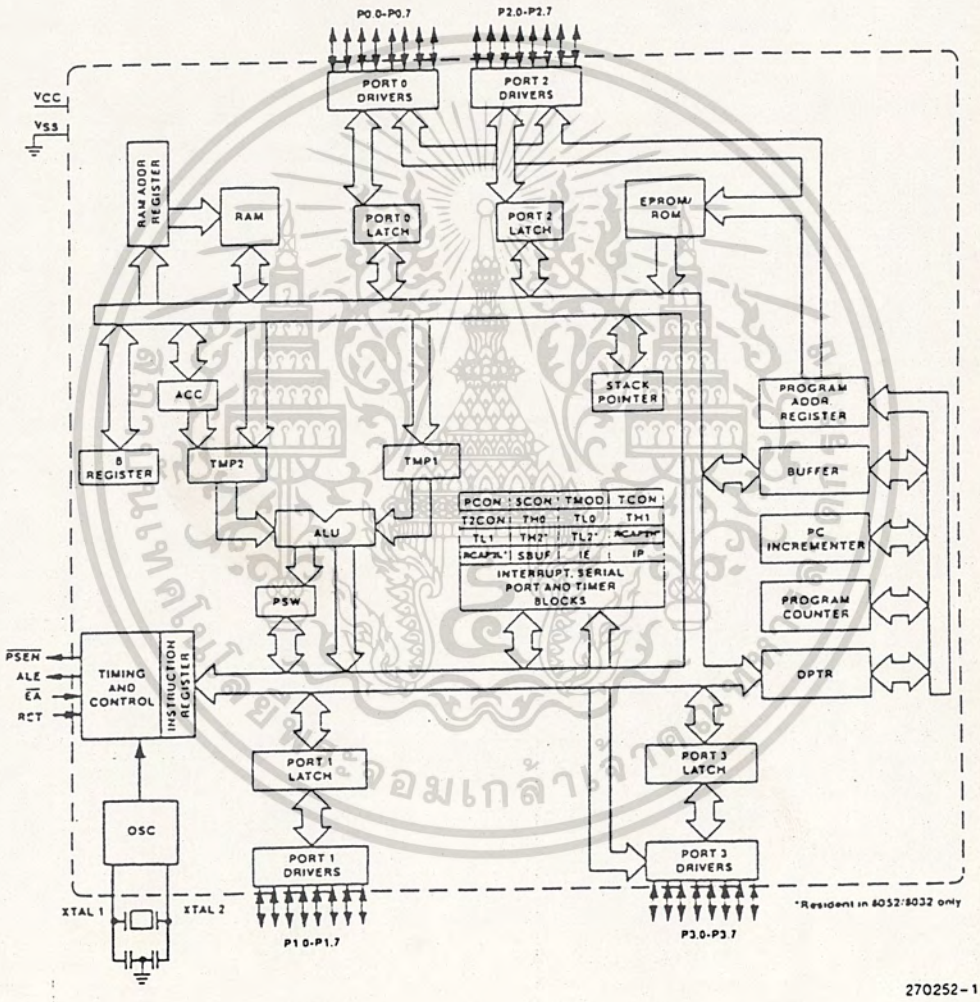
หน่วยประมวลผลกลางเป็นระบบไมโครโปรเซสเซอร์ซึ่งได้ออกแบบมาเพื่อทำการควบคุมกระบวนการโดยรับค่าสภาวะมาทางหน่วยอินพุทและทำการประมวลผลข้อมูลอินพุทตามโปรแกรมที่ผู้ใช้กำหนดและส่งข้อมูลไปยังหน่วยเอาท์พุทโดยมีโปรแกรมควบคุมระบบเป็นตัวจัดการ ข้อมูลอินพุทจะได้รับมาจากค่าสภาวะของกระบวนการและข้อมูลเอาท์พุทก็คือข้อมูลที่ส่งออกไปควบคุมกระบวนการ ส่วนโปรแกรมที่ผู้ใช้เขียนขึ้นนั้นเป็นชุดคำสั่งการประมวลผลทั้งทางด้านลอจิก คณิตศาสตร์ และการเคลื่อนย้ายข้อมูล ปริญญานิพนธ์ฉบับนี้ได้พัฒนาโดยใช้ไมโครคอนโทรลเลอร์อินเทล (INTEL) เบอร์ 8032 ซึ่งเป็นไมโครคอนโทรเลอร์ในตระกูล MCS-51 สมรรถนะสูงและมีชุดคำสั่งที่ง่ายต่อการใช้งาน โครงสร้างของ MCS-51 จะมีลักษณะดังรูปที่ 3.5 และรูปที่ 3.6



รูปที่ 3.5 ลักษณะขาภายนอกของ MCS-51



รูปที่ 3.6 สัญญลักษณ์ทางตรรกของ MCS-51



รูปที่ 3.7 สถาปัตยกรรม บล๊อคไดอะแกรม ของ MCS-51

ซีพียูเป็นมันสมองของระบบไมโครคอมพิวเตอร์ การอ่านโปรแกรม และทำงานตามคำสั่งโปรแกรมจะกระทำที่ส่วนนี้ โดยการใช้ส่วนคณิตศาสตร์ และตรรกศาสตร์ทำงานร่วมกับเรจีสเตอร์ A, B, PSW (Program Status Word), SP (Stack Pointer) ตัวนับโปรแกรม (PC : Program Counter) ขนาด 16 บิต และตัวชี้ตำแหน่งข้อมูล (DPTR : Data Pointer) ส่วนคณิตศาสตร์และตรรกศาสตร์ (ALU:Arithmetic Logic Unit) ALU นี้ทำงานในฟังก์ชั่นทางการทำงานทางคณิตศาสตร์และตรรกศาสตร์ด้วยตัวแปรต่าง ๆ ขนาด 8 บิท ที่มีลักษณะการทำงานทางคณิตศาสตร์เป็น บวก ลบ คูณ หาร รวมทั้งทางตรรกศาสตร์ เช่น AND OR XOR รวมทั้งการเลื่อนและวนรอบบิต การเคลียร์ค่าและกลับค่า (Complement) เป็นต้น ALU ยังสามารถที่จะตัดสินใจในการให้กระโดดไปทำคำสั่งของโปรแกรมในส่วนอื่น ๆ ตามเงื่อนไขที่ตั้งขึ้น และยังแบ่งเรจีสเตอร์ชั่วคราวใช้สำหรับเป็นทางผ่านชั่วคราวของข้อมูลในการถ่ายเทภายในระบบ คำสั่งอื่นที่มีการใช้ ALU ยังมีความสามารถที่จะเพิ่มค่าในเรจีสเตอร์ในลักษณะการบวกด้วยหนึ่ง (Increment) หรือ คำนวณเลขที่อยู่ของข้อมูลที่จะนำไปเก็บหรือการลดค่าลงครั้งละหนึ่ง ในลักษณะการลบด้วยค่าหนึ่ง (Decrement) โดยอัตโนมัติ หรือใช้ในการเปรียบเทียบค่าของตัวแปรทั้งสอง

สิ่งสำคัญในการทำงานทางสถาปัตยกรรมของ MCS-51 คือ ความสามารถในการทำงานสำหรับข้อมูลขนาด 8 บิต และความสามารถเช่นนี้เหมาะสำหรับใช้ในงานควบคุมของสัญญาณเข้าและออกที่มีการคิดและการออกแบบทางตรรกด้วยพีชคณิต Boolean ซึ่งโดยปกติทำได้ลำบากสำหรับไมโครโพรเซสเซอร์ทั่ว ๆ ไป งานในลักษณะเช่นนี้จึงได้ชื่ออีกอย่างหนึ่งว่า ตัวประมวลผลบูลลีน (Boolean Processor)

1. แอกคิวมิวเลเตอร์ (Accumulator : ACC)

MCS-51 ก็เช่นเดียวกับ MCS-48 ที่ใช้ ACC ที่มีขนาด 8 บิต เป็นเอกคิวมิวเลเตอร์หลัก คำสั่งส่วนใหญ่จะอ้างถึงตัวรีจีสเตอร์นี้ โดยถือค่าภายในเป็นค่าตัวตั้ง และรับค่าผลลัพธ์ที่ได้จากคำสั่งทางคณิตศาสตร์ เช่น บวก ลบ คูณ หาร เข้ามาเก็บไว้ ตัว ACC ยังสามารถใช้เป็นตัวแหล่งกระทำหรือถูกกระทำในการทำงานทางตรรก และใช้เป็นตัวกลางในการถ่ายเทข้อมูลในการติดต่อกับอุปกรณ์ภายนอกไอโอ และหน่วยความจำภายนอก รวมถึงการตรวจสอบตารางข้อมูล

2. เรจีสเตอร์ B

เป็นเรจีสเตอร์พิเศษที่ใช้งานสำหรับคำสั่งของการคูณและหาร โดยใช้เป็นที่เก็บตัวคูณหรือตัวหาร และเป็นที่เก็บผลลัพธ์ตัวที่สอง หลังการคูณและเศษหลังการหาร

3. เรจีสเตอร์คำแสดงสถานะโปรแกรม (Program Status Word : PSW)

เรจีสเตอร์ PSW เป็นเรจีสเตอร์ที่แสดงผลที่ได้หลังจากการใช้คำสั่งต่าง ๆ และใช้เป็นตัวเลือกกลุ่มการทำงานของเรจีสเตอร์กลุ่มต่าง ๆ ซึ่งมีรายละเอียดดังรูปข้างล่าง

(MSB) (LSB)

CY	AC	F0	RS1	RS0	OV	-	P

สัญญลักษณ์	ตำแหน่ง	ข้อกำหนดการทำงาน
CY	PSW7	แฟลกตัวทด จะเซต/เคลียร์ด้วยฮาร์ดแวร์หรือซอฟต์แวร์ ระหว่างผลลัพธ์หลังการใช้คำสั่งทางคณิตศาสตร์ หรือ ตรรกศาสตร์ที่แน่นอน
AC	PSW6	แฟลกตัวทดของ Auxiliary จะเซต/เคลียร์ด้วยฮาร์ดแวร์ ระหว่างการบวกและลบ ที่แสดงผลจากการทดหรือยืมจากบิตที่ 3 ของ ACC
F0	PSW5	แฟลก 0 จะเซต/เคลียร์ด้วยของซอฟต์แวร์ที่ผู้ใช้กำหนดสถานะแฟลกนี้เอง
RS1	PSW4	เรจีสเตอร์ตัวควบคุมการเลือกแบงค์ ด้วยค่า RS1 และ RS0
RS0	PSW3	จะเซต/เคลียร์ด้วยซอฟต์แวร์ เพื่อเลือกกลุ่มเรจีสเตอร์ทำงานในแต่ละแบงค์ โดยปรับค่าใน RS1 และ RS0 ให้อีนาเบิ้ลคลุมลักษณะการเลือกแบงค์ต่อไปนี้

RS1	RS0	เลือกแบงค์	ค่าแอดเดรส
0	0	แบงค์ 0	00H-07H
0	1	แบงค์ 1	08H-0FH
1	0	แบงค์ 2	10H-17H
1	1	แบงค์ 3	18H-1FH

OV	PSW2	แฟลก Overflow จะเซต/เคลียร์ด้วยฮาร์ดแวร์ระหว่างการใช้คำสั่งที่แสดงผลถึงการเกิดลักษณะ Overflow ทางคณิตศาสตร์
-	PSW1	บิตสำรอง จะไม่สามารถเซต/เคลียร์ด้วยผู้ใช้ เพราะสำรองไว้สำหรับโรงงานผู้สร้าง
P	PSWO	แฟลกพาริตี้ จะเซต/เคลียร์ด้วยฮาร์ดแวร์ในแต่ละวัฏจักรคำสั่ง แสดงถึงตัวเลขค่า "1" ในแต่ละบิตของแอกคิวมิวเลเตอร์เช่น "1" มี 6 ตัว จะเป็นพาริตี้คู่ P บิตจะเท่ากับ 0

4. ตัวชี้สแต็ก (Stck Pointer : SP)

MCS-51 จะรวมเอาสแต็กทางฮาร์ดแวร์ที่ใช้ RAM ภายในสำหรับการเชื่อมต่อระหว่างโปรแกรมหลัก สแต็กการผ่านพารามิเตอร์ระหว่างงานในแต่ละส่วนโปรแกรม และสแต็กเก็บตัวแปรข้อมูลชั่วคราว หรือสแต็กการเก็บสถานะระหว่างการบริการอินเตอร์รัพต์ไว้ภายในชิป โดยที่ SP จะมีขนาด 8 บิต จะเพิ่มค่าขึ้นอัตโนมัติก่อนที่ข้อมูลจะนำมาเก็บในหน่วยความจำระหว่างการใช้คำสั่ง PUSH และ CALL และจะลดค่าของ SP ลงหลังจากที่ได้ถ่ายเทข้อมูลออกไปแล้วในคำสั่ง POP หรือ RETURN โดยทฤษฎีทางสถาปัตยกรรม MCS-51 สามารถใช้สแต็กให้มีเนื้อที่ถึง 128 ไบต์ แต่ในทางปฏิบัติสำหรับโปรแกรมทั่วไปและใช้น้อยกว่านี้ SP จะเริ่มที่ตำแหน่ง 07H ดังนั้น สแต็กจะเริ่มบรรจุข้อมูลที่ตำแหน่ง 08H MCS-51 สามารถเปลี่ยนแปลงค่าใน SP ได้ ซึ่งจะเป็นการเปลี่ยนแปลงตำแหน่งสแต็กไปยังที่ใด ๆ ของ RAM ภายในชิป

5. ตัวชี้ข้อมูล (Data Pointer : DPTR)

DPTR เรจิสเตอร์ขนาด 16 บิต ที่ประกอบด้วยไบต์สูง (DPH) และไบต์ต่ำ (DPL) ที่เราสามารถเลือกแบ่งออกเป็น เรจีสเตอร์ 8 บิต สองตัวที่ใช้ได้อย่างอิสระ หรือจะใช้รวมกันทั้ง 16 บิต ก็ได้ ในการ Increment หรือ Decrememt เพื่อประโยชน์ในการใช้เป็นฐานของเลขที่อยู่ในเรจีสเตอร์ในการกระโดดโดยทางอ้อมในการใช้คำสั่งเกี่ยวกับตารางข้อมูลและชี้ตำแหน่งของหน่วยความจำภายนอก

6. พอร์ต 0 ถึง 3

เรจีสเตอร์ P0, P1, P2 และ P3 ของกลุ่มเรจีสเตอร์ฟังก์ชั่นพิเศษ (Special Function Register : SFR) จะเป็นตัวเรจีสเตอร์ที่แลทช์ค่าของพอร์ต 0, 1 , 2 และ 3 ตามลำดับ ในขณะใช้งาน

7. บัฟเฟอร์ข้อมูลอนุกรม (Serial Data Buffer : SBUF)

บัฟเฟอร์ข้อมูลอนุกรมแบ่งออกเป็นรีจีสเตอร์สองตัว ตัวหนึ่งเป็นบัฟเฟอร์การรับ เมื่อข้อมูลถ่ายเทเข้า SBUF มันจะถ่ายเข้าบัฟเฟอร์ซึ่งเป็นตัวจัดการส่งข้อมูลอนุกรม วิธีการเคลื่อนย้ายเข้า SBUF ขึ้นอยู่กับการเริ่มแรก (Initial) การส่งเมื่อข้อมูลย้ายออกจาก SBUF จะเป็นการรับข้อมูลจากบัฟเฟอร์ตัวรับ

8. เรจีสเตอร์ Capture

IC เบอร์ 8032/8052 จะมีคู่รีจีสเตอร์ (RCAP2H,RCAP2L) เพิ่มเติมเป็นรีจีสเตอร์แคปเจอร์สำหรับตัวจับเวลาหมายเลข 2 ในโหมดการใช้งานของรีจีสเตอร์คู่นี้จะรับการเปลี่ยนแปลงที่เข้ามาที่ขา T2EX คู่ TH2 และ TL2 จะลอกข้อมูลเข้าไปในรีจีสเตอร์คู่ RCAP2H และ RCAP2L ด้วยการใช้การจับเวลา จะมีโหมดการบรรจุอัตโนมัติขนาด 16 บิท สำหรับการใช้ตัวจับเวลา/ตัวนับสอง

9. เรจีสเตอร์ควบคุม (Control Register)

กลุ่ม SFR ที่เป็น IP, IE, TMOD, TCON, T2CON, SCON และ PCON จะประกอบด้วยบิตที่ใช้ในการควบคุม และแสดงสถานะของการใช้งานในระบบอินเตอร์รัพต์ ตัวจับเวลา/ตัวนับ และพอร์ตอนุกรม

3.3 หน่วยอินพุตเอาท์พุท (INPUT OUTPUT UNIT)

หน่วยอินพุทเอาท์พุท มีหน้าที่เชื่อมต่อกับอุปกรณ์ภายนอก ทั้งการรับค่าสภาวะหรือค่าที่วัดจากอุปกรณ์ภายนอก เช่นการเปิดปิดของสวิทช์ต่าง ๆ ระดับของเหลว อุณหภูมิ ความดัน

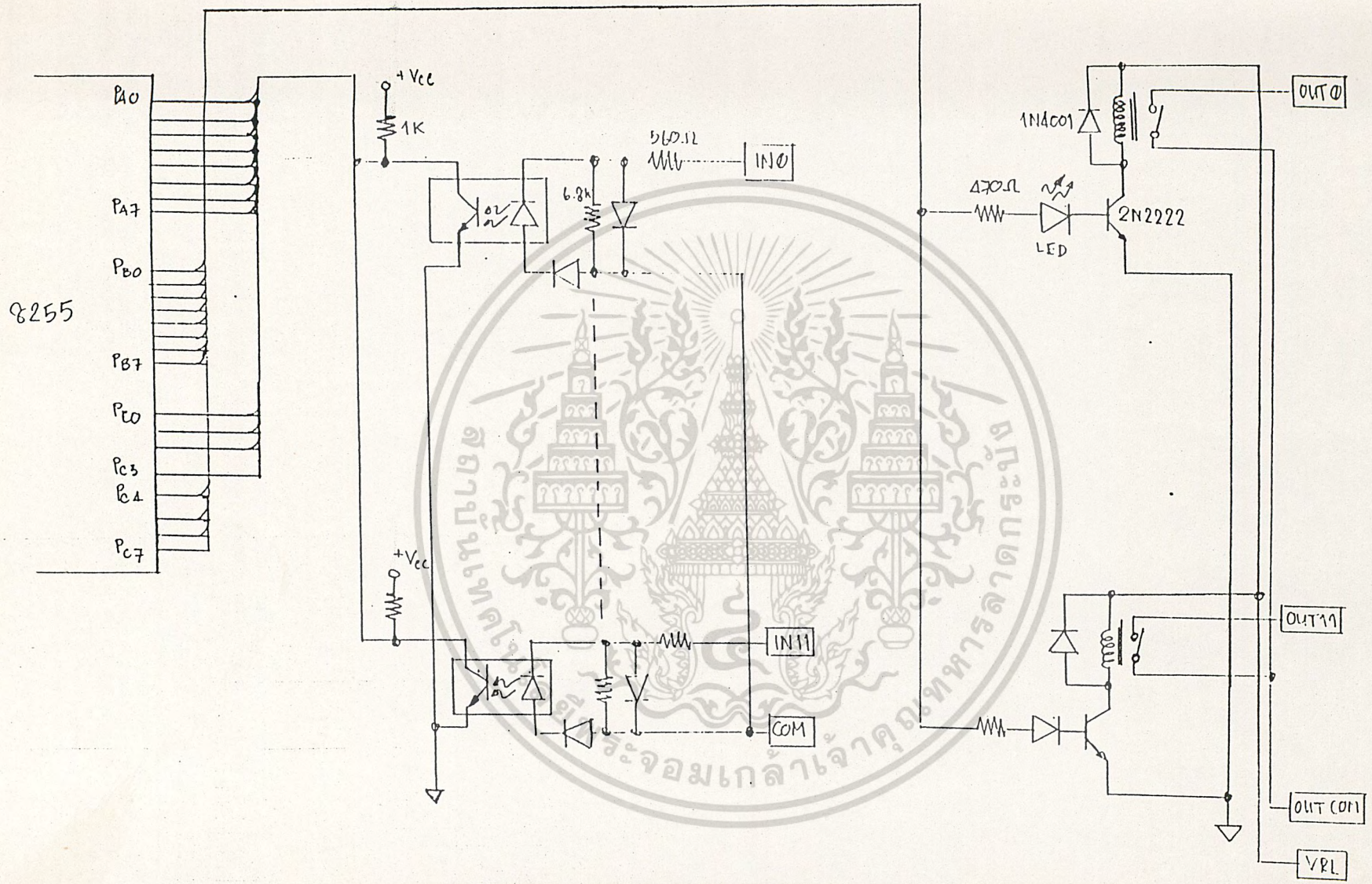
ระดับแรงดันและกระแสไฟฟ้า หรือส่งค่าสภาวะออกไปยังอุปกรณ์ควบคุม หรืออุปกรณ์ที่ต้องการควบคุม เช่น รีเลย์ มอเตอร์ไฟฟ้า ปั๊ม วาลว์ เป็นต้น แบ่งออกดังนี้

3.3.1 หน่วยอินพุทและสภาวะลอจิก ทำหน้าที่รับค่าสภาวะของอุปกรณ์ภายนอกที่มีการเปลี่ยนแปลงสภาวะเป็นแบบลอจิกที่มีแรงดันไฟฟ้าขนาด 12 โวลท์ โดยเปลี่ยนให้เป็นสภาวะทางลอจิกจากนั้นหน่วยประมวลผลจะทำการอ่านและนำเก็บสภาวะลอจิกดังกล่าวไว้ใน INTERNAL MEMORY

จากรูปที่ 3.8 แรงดันไฟฟ้าที่จุด IN0-IN11 จะต้องมีขนาดแรงดัน 12 โวลท์เมื่อเทียบกับจุด COM เพื่อที่จะทำให้ทรานซิสเตอร์ (TRANSISTOR) ที่อยู่ภายในอุปกรณ์เชื่อมต่อด้วยแสงหรือออปโตไอโซเรเตอร์ (OPTO ISOLATOR) สามารถนำกระแส (CONDUCT) ได้ ออปโตไอโซเรเตอร์ทำหน้าที่เชื่อมต่อการส่งสัญญาณไฟฟ้าด้วยแสงเพื่อแยกออกจากกันอย่างเด็ดขาดทางไฟฟ้าระหว่างอุปกรณ์ภายนอกกับเครื่องควบคุม เพื่อป้องกันความเสียหายซึ่งอาจจะเกิดขึ้นเนื่องมาจากการลัดวงจรของอุปกรณ์ภายนอก การนำกระแสของทรานซิสเตอร์ก็คือสภาวะลอจิกที่มีแรงดันตามมาตรฐานทีทีแอล (TTL) ซึ่งสภาวะลอจิกดังกล่าวก็คือข้อมูลสภาวะอินพุทนั่นเอง ข้อมูลนี้จะถูกส่งไปยัง U11 (8255 PPI) U11 จะทำหน้าที่รับข้อมูลเหล่านี้ส่งผ่านให้หน่วยประมวลผลกลาง

3.3.2 หน่วยเอาท์พุทแบบหน้าสัมผัส ทำหน้าที่นำข้อมูลจากหน่วยประมวลผลกลางส่งออกไปยังอุปกรณ์ควบคุมซึ่งเชื่อมต่อให้อยู่ในสภาวะตามต้องการ โดยข้อมูลดังกล่าวได้จาก INTERNAL MEMORY แล้วใช้หน้าสัมผัสเป็นตัวควบคุมเปิดปิดวงจรไฟฟ้า สามารถควบคุมอุปกรณ์ไฟฟ้าทั้งชนิดกระแสตรงหรือกระแสสลับได้ ดังแสดงในรูปที่ 3.8

หน่วยประมวลผลจะส่งสภาวะลอจิกให้กับพอร์ท U11 ทำให้มีการเปลี่ยนแปลงของระดับไฟฟ้า เป็นผลให้ทรานซิสเตอร์เกิดการนำกระแสไหลผ่านขดลวดของรีเลย์และทำให้หน้าสัมผัสทำงาน ดังนั้นอุปกรณ์ควบคุมที่ต่ออยู่กับเอาท์พุทตำแหน่งดังกล่าวก็จะทำงานตามสภาวะที่ต้องการ



รูปที่ 3.8 วงจร INPUT และ OUTPUT

โครงสร้างพอร์ตและการทำงาน

ใน MCS-51 มีพอร์ต 4 พอร์ต และทั้งสี่พอร์ตเป็นแบบสองทิศทาง แต่ละพอร์ตจะประกอบด้วยแลทช์เป็น P0 ถึง P3 ของ SFR จะมีตัวขับเอาท์พุตและบัฟเฟอร์อินพุต ตัวขับเอาท์พุตของพอร์ต 0 และ 2 และบัฟเฟอร์อินพุตของพอร์ต 0 จะใช้งานสำหรับการเข้าถึงหน่วยความจำภายนอก ในการใช้งานลักษณะนี้เอาท์พุตพอร์ต 0 จะทำหน้าที่เป็นตัวกำหนดไบต์ต่ำของแอดเดรสหน่วยความจำภายนอก โดยที่ค่าแอดเดรสและค่าข้อมูลจะถูกมัลติเพลกซ์ด้วยช่วงจังหวะการเฟทช์และการอ่านหรือเขียนข้อมูล ส่วนเอาท์พุทพอร์ต 2 จะทำหน้าที่เป็นตัวกำหนดส่งไบต์สูงของแอดเดรสในการเข้าถึงหน่วยความจำภายนอก โดยที่ค่าแอดเดรส และค่าข้อมูลจะถูกมัลติเพลกซ์ด้วยช่วงจังหวะการเฟทช์และการอ่านหรือเขียนข้อมูล ส่วนเอาต์พุตพอร์ต 2 จะทำหน้าที่เป็นตัวกำหนดส่งไบต์สูงของแอดเดรสในการเข้าถึงหน่วยความจำภายนอก

บางขาของตัวขับเอาต์พุตและบัฟเฟอร์อินพุตของขา 1.0, 1.1 และ พอร์ต 3 ทั้งหมดสามารถนำไปใช้งานเป็นแบบหลายฟังก์ชั่น (Multifunction) ได้ดังนี้

ขาพอร์ต	การใช้งานตามฟังก์ชั่น
* P1.0	T2 (Timer/Counter 2 สัญญาณอินพุตจากภายนอก)
* P1.1	T2RST (Timer/Counter 2 สัญญาณอินพุตการรีเซตภายนอก)
P3.0	R x D (พอร์ตรับข้อมูลอนุกรม)
P3.1	T x D (พอร์ตส่งข้อมูลอนุกรม)
P3.2	INT0 (การใช้อินเตอร์รัพต์ภายนอกตัวที่ 1)
P3.3	INT1 (การใช้อินเตอร์รัพต์ภายในนอกตัวที่ 2)
P3.4	T0 (Timer/Counter 0 สัญญาณอินพุตภายนอก)
P3.5	T1 (Timer/Counter 1 สัญญาณอินพุตภายนอก)
P3.6	WR (สโตรบการเขียนหน่วยความจำภายนอก)
P3.7	RD (สโตรบการอ่านหน่วยความจำภายนอก)

ตัวขับเอาต์พุตแลทช์ในการที่จะให้ทำงานตามตารางบน จะต้องเริ่มโปรแกรมด้วยการเซตค่า "1" เก็บในแลทช์ก่อน เครื่องหมาย * แสดงถึงการใช้ตัว Timer/Counter 2 ซึ่ง

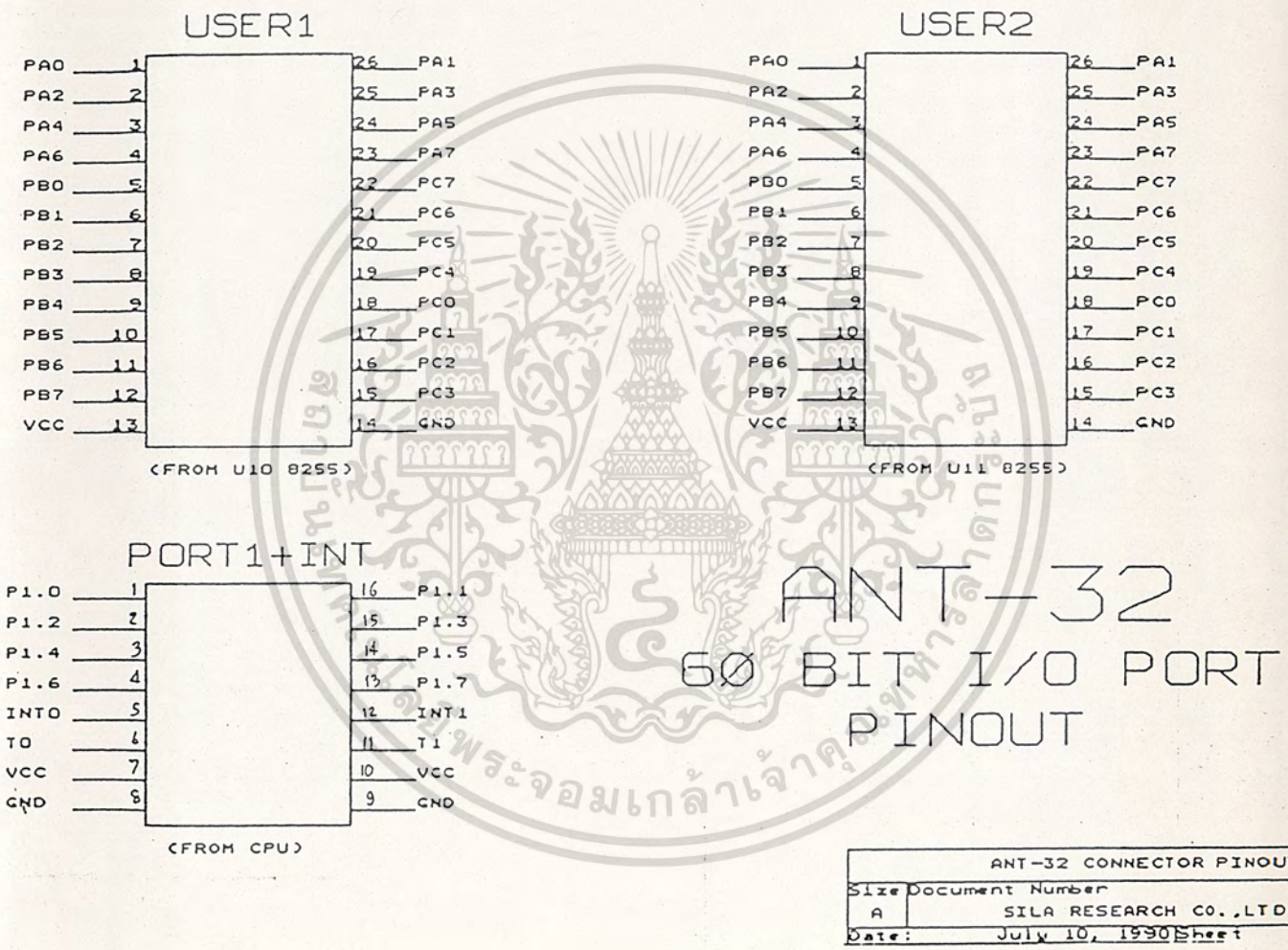
มีเฉพาะในเบอร์ 8032/8052 เท่านั้น ทั้งโหมดตัวจับเวลาหรือตัวนับ การโหลดใหม่แบบอัตโนมัติของ Timer/Counter 2 ที่เรจิสเตอร์ RLDH และ RLDL จะเกิดขึ้นถ้าการโหลดใหม่แบบอัตโนมัติถูกเลือกใช้งานด้วยการกำหนดในบิต CP/RL 2 = 0

3.4 หน่วยเชื่อมต่อกับคอมพิวเตอร์

หน่วยเชื่อมต่อกับคอมพิวเตอร์ทำหน้าที่ส่งชุดคำสั่งของโปรแกรมผู้ใช้ซึ่งจะเป็นโปรแกรมที่กำหนดลำดับการทำงานและการประมวลผลของการควบคุม ไปยังเครื่องควบคุมและรับค่าสภาวะอินพุทเอาท์พุทจากเครื่องควบคุมมายังคอมพิวเตอร์ เพื่อใช้ในการตรวจสอบและติดตามการเปลี่ยนแปลงสภาวะของกระบวนการ การเชื่อมต่อกับคอมพิวเตอร์กระทำผ่านพอร์ทอนุกรมมาตรฐาน RS-232C ดังแสดงในรูปที่ 3.4

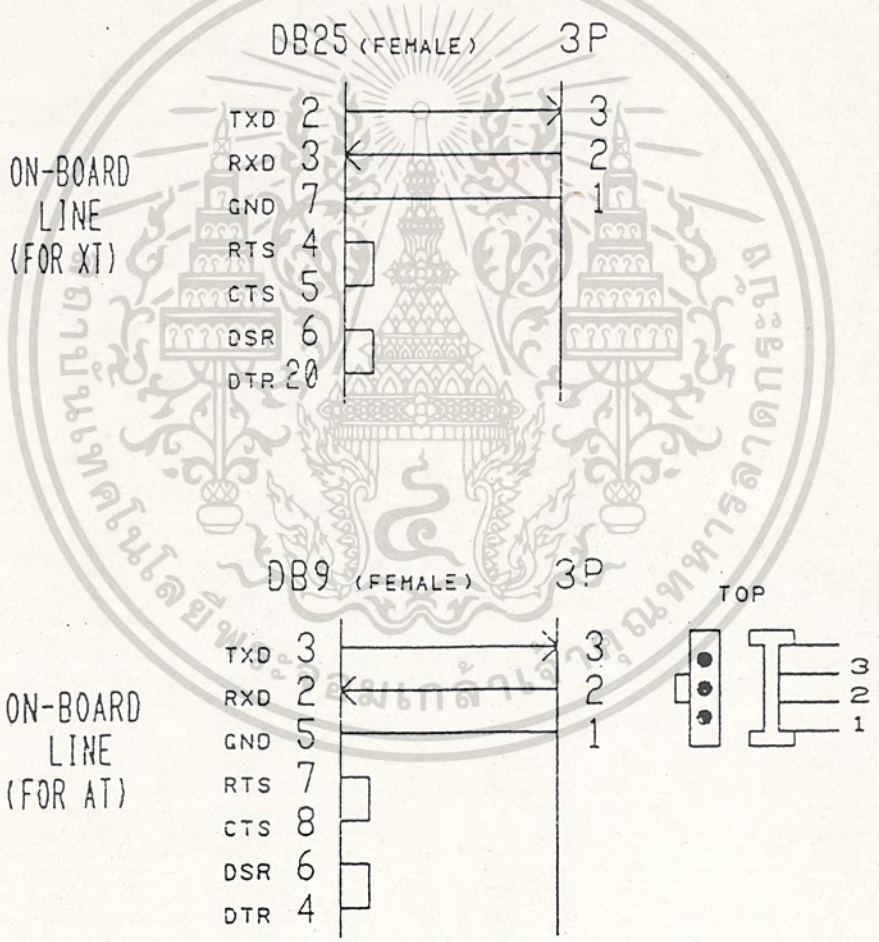
การรับและส่งข้อมูลแบบอนุกรมระหว่างคอมพิวเตอร์กับเครื่องควบคุมจะทำให้ระยะทางในการสื่อสารข้อมูลได้ระยะทางไกลและค่าใช้จ่ายต่ำ เนื่องจากระดับแรงดันของสัญญาณถูกทำให้มีขนาดสูงถึง -12 และ +12 โวลท์เมื่อข้อมูลเป็น 0 และ 1 ตามลำดับ ส่วนความเร็วในการรับส่งโดยปกติแล้วจะใช้ความเร็วในการรับส่งขนาด 9600 บิทต่อวินาที

การต่อสาย RS-232 ดังแสดงในรูปที่ 3.10

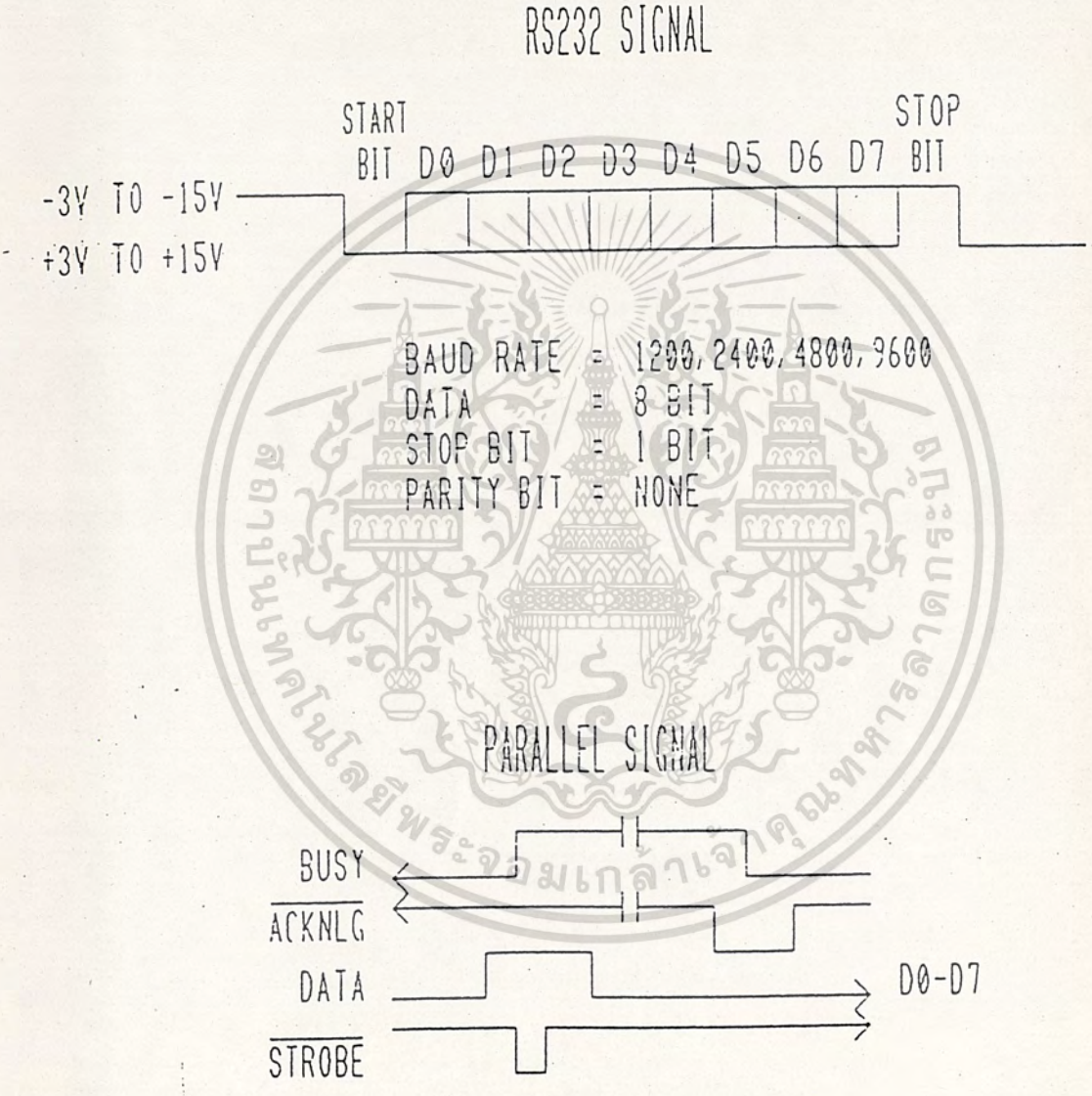


รูปที่ 3.9

การต่อสาย RS232



รูปที่ 3.10



รูปที่ 3.11

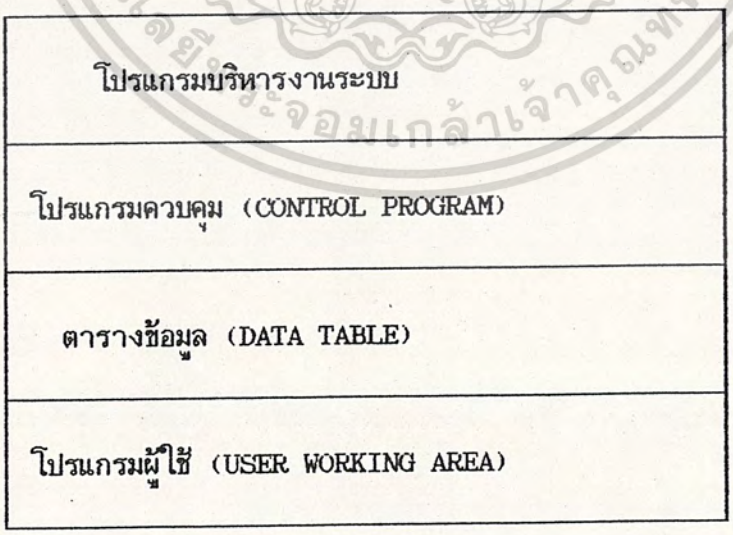
บทที่ 4
ระบบจัดการของเครื่องควบคุมที่โปรแกรมได้

ระบบการจัดการของเครื่องควบคุมหมายถึงโปรแกรมคำสั่งที่ใช้ในการควบคุมการทำงานการจัดการหน่วยความจำและโครงสร้างข้อมูล การติดต่อกับหน่วยงานอื่น ๆ ทำการแปลคำสั่งควบคุมให้เป็นรหัสที่หน่วยประมวลผลกลางสามารถปฏิบัติได้ และการทำงานของฟังก์ชั่นควบคุม โดยจะอธิบายเป็นลักษณะของอัลกอริทัม แบ่งเป็นหัวข้อดังนี้

- การแบ่งพื้นที่ใช้งานของหน่วยความจำ
- การจัดข้อมูลของโปรแกรมผู้ใช้เป็นภาษาเครื่อง
- การทำงานของฟังก์ชั่นควบคุม

4.1 การแบ่งพื้นที่ใช้งานของหน่วยความจำ

จากบทที่ 3 ในโครงสร้างทางฮาร์ดแวร์ของหน่วยประมวลผลกลางมีหน่วยความจำที่ใช้ในการประมวลผลขนาด 24 กิโลไบท์ที่ถูกกำหนดขึ้นสำหรับเครื่องควบคุม แบ่งเป็นพื้นที่ใช้งานดังรูปที่ 4.1

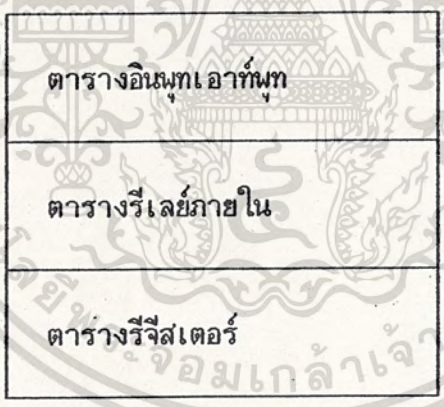


รูปที่ 4.1 แสดงการแบ่งพื้นที่ใช้งานของหน่วยความจำ

4.1.1 โปรแกรมบริหารงานระบบ ใช้พื้นที่หน่วยความจำร่วมกับข้อมูลระบบ มีขนาด 8 กิโลไบท์ ถูกใช้เป็นโปรแกรมจัดวางระบบข้อมูลโปรแกรมผู้ใช้ แปลชุดคำสั่งของโปรแกรมผู้ใช้เป็นภาษาเครื่อง ติดต่อสื่อสารกับหน่วยป้อนโปรแกรมและอุปกรณ์ร่วมอื่น ๆ พื้นที่ส่วนนี้ผู้ใช้ไม่สามารถที่จะเปลี่ยนแปลงแก้ไขข้อมูลในหน่วยความจำได้ ในบอร์ด ANT-32 คือ ROM U2 (ดูรูปที่ 3.2)

4.1.2 โปรแกรมควบคุม เป็นพื้นที่หน่วยความจำของโปรแกรมในการประมวลผลฟังก์ชั่นควบคุมที่ใช้ผู้ใช้ทำการโปรแกรม เข้ามาในหน่วยความจำของพื้นที่ส่วนโปรแกรมผู้ใช้

4.1.3 ตารางข้อมูล เป็นพื้นที่หน่วยความจำที่ผู้ใช้สามารถเปลี่ยนแปลงแก้ไขได้ภายในตารางข้อมูล จะถูกเก็บข้อมูลที่มีลักษณะ "1" หรือ "0" ใช้แทนสภาวะ "ON" และ "OFF" ทางไฟฟ้า และข้อมูลแทนค่าตัวเลขที่ใช้ในการคำนวณทางคณิตศาสตร์ ข้อมูลของการตั้งค่าเวลาและค่าการนับ ซึ่งตารางข้อมูลได้ถูกแบ่งออกเป็น 3 ส่วนดังรูปที่ 4.2



รูปที่ 4.2 แสดงการแบ่งพื้นที่ใช้งานในตารางข้อมูล

ตารางอินพุทเอาท์พุทและตารางรีเลย์ภายในถูกกำหนดขึ้นตารางละ 8 บิท ดังรูปที่ 4.3

	07	06	05	04	03	02	01	00	
address 20	x	x	x	x	x	x	x	x	INPUT PORT A FC00
address 22	x	x	x	x	x	x	x	x	OUTPUT PORT B FC01
Address 24	x	x	x	x	x	x	x	x	INTERNAL RELAY

รูปที่ 4.3 ตารางอินพุท เอาท์พุท และตารางรีเลย์ภายใน

4.1.4 โปรแกรมผู้ใช้ หน่วยความจำส่วนนี้เก็บโปรแกรมควบคุม ที่ผู้ใช้เป็นผู้เขียนขึ้นโดยใช้หน่วยป้อนโปรแกรมซึ่งจะถูกเปลี่ยนให้เป็นรหัสของชุดคำสั่งนั้น ๆ ข้อมูลในหน่วยความจำของโปรแกรมผู้ใช้ จะเป็นรหัสของชุดคำสั่งซึ่งเป็นคำสั่งทางลอจิก คำสั่งทางคณิตศาสตร์ คำสั่งการเคลื่อนย้ายข้อมูล ควบคู่กับตำแหน่งของอินพุทหรือเอาท์พุท เช่น

LD 01 คำสั่งคือ LD ตำแหน่งอินพุท 01

OUT 22 คำสั่งคือ OUT ตำแหน่งเอาท์พุท 22

4.2 การจัดข้อมูลของโปรแกรมผู้ใช้

ข้อมูลของโปรแกรมผู้ใช้จะมีลักษณะเป็นชุดคำสั่งแต่ละชุดคำสั่งจะประกอบด้วย CODE และ DATA

ฟิลด์ CODE ใช้กำหนดรหัสของคำสั่งที่ผู้ใช้ทำการโปรแกรมขึ้นเพื่อให้เครื่องควบคุมทำงานตามฟังก์ชั่นการควบคุม คำสั่งที่ได้สร้างขึ้นมี ดังรายละเอียดต่อไปนี้

CODE	คำสั่ง	ความหมาย
00	LD	นำค่าสภาวะจากตารางข้อมูลมาเป็นผลลัพธ์
01	LD NOT	นำค่าสภาวะที่ตรงข้ามจากตารางข้อมูลมาเป็นผลลัพธ์
02	AND	กระทำลอจิก AND กับสภาวะที่นำจากตารางข้อมูล
03	AND NOT	กระทำลอจิก AND กับสภาวะตรงข้ามที่นำมาจากตารางข้อมูล
04	OR	กระทำลอจิก OR กับสภาวะที่นำจากตารางข้อมูล
05	OR NOT	กระทำลอจิก OR กับสภาวะตรงข้ามที่นำมาจากตารางข้อมูล
06	AND LD	กระทำลอจิก AND กับสภาวะที่นำจาก STACK
07	OR LD	กระทำลอจิก OR กับสภาวะที่นำจาก STACK
08	OUT	นำผลลัพธ์ทางลอจิกไว้ในตารางข้อมูล
09	OUT NOT	นำผลลัพธ์ทางลอจิกแล้วทำตรงข้ามไว้ในตารางข้อมูล
10	END	สิ้นสุดคำสั่งให้เริ่มทำรอบใหม่

บทที่ 5

โครงงาน และ การทดลองโครงงาน

ปริญญานิพนธ์ฉบับนี้ ใช้เครื่อง PERSONAL COMPUTER มาช่วยในการประมวลผล และเชื่อมต่อกับเครื่องควบคุมที่โปรแกรมได้ ทางพอร์ทแบบอนุกรม RS-232C

การพัฒนา PROGRAM ใช้ PROGRAM WORD PROCESSOR ช่วยในการเขียน โดยใช้คำสั่ง INSTRUCTION SET สำหรับ MCS-51 (ดังตาราง INSTRUCTION SET ในภาคผนวก 1) และแบ่ง MEMORY ออกเป็น 3 ส่วน โดยเริ่มต้นจาก ADDRESS ที่ 8000 H ถึง 89FF H ช่วงที่ 2 เริ่มจาก 8A00 H ถึง 8BFF H และช่วงที่ 3 เริ่มจาก 8C00 H ถึง 9000 H ดังรูป 5.1

5.1 การพัฒนาโปรแกรมด้วยภาษาแอสเซมบลี้ มีหลักการดังนี้

5.1.1 การเขียนโปรแกรมแอสเซมบลี้ให้ใช้โปรแกรมเอดิเตอร์ทั่วไป หรือ ใช้โปรแกรม ED.COM ที่มีอยู่ในดิสค์ ANT-32 UTILITY และจุดเริ่มต้นของโปรแกรม (origin) ต้องอยู่ที่แอดเดรส 8000 H

5.1.2 โปรแกรมแอสเซมบลี้ที่เขียนเสร็จแล้ว นำมาแปลเป็นอ็อบเจ็คโค้ด โดยใช้โปรแกรมตัวแปลภาษาแอสเซมบลี้ 8051 ทั่วไป หรือ ใช้โปรแกรม SXA51.EXE ที่มีอยู่ในดิสค์ ANT-32 UTILITY โดยไฟล์ที่แปลได้จะอยู่ในลักษณะของ INTEL HEX FILE

5.1.3 ต่อสาย RS232 ระหว่างเครื่องคอมพิวเตอร์ PC กับ ANT-32 ให้เรียบร้อย

5.1.4 เรียกใช้โปรแกรม ANT32.BAT ซึ่งอยู่ในแผ่นดิสค์ "ANT-32 UTILITY"

A> ANT32 (กดคีย์ Enter)

เครื่องจะเข้าโปรแกรม CROSSTALK และอยู่ในสภาวะที่พร้อมจะทำการสื่อสารโดยเครื่องคอมพิวเตอร์ PC จะทำหน้าที่ เสมือนเป็นจอภาพ และ คีย์บอร์ด ของ ANT-32 (การใช้งานโปรแกรม XTALK ดูได้จากภาคผนวก 2)

5.1.5 เปิดเครื่อง ANT-32 และกด SPACE ที่เครื่องคอมพิวเตอร์ PC, ANT-32 จะทำการคำนวนค่าอัตราการรับส่งข้อมูล ให้โดยอัตโนมัติ และถ้าระบบเรียบร้อยดี จะมีข้อความต่อไปนี้ ปรากฏบนจอภาพ

```
* MCS-51 (tm) BASIC V1.1*
  READY
  >
```

5.1.6 ใช้คำสั่ง XDOWN ทำการ DOWN LOAD ไฟล์โปรแกรมแอสเซมบลี้ที่แปลแล้วที่อยู่ในลักษณะ INTEL HEX FILE ลงมาที่หน่วยความจำของ ANT-32

XDOWN [OFFSET]

สำหรับการ DOWN LOAD ข้อมูลในลักษณะ INTEL-HEX FILE ลงใน RAM ทั้งนี้เพื่อโหลดส่วนที่เป็นข้อมูล หรือส่วนที่แปลมาจาก COMPILER ตัวอื่น ๆ ได้ คำสั่งนี้จะรับข้อมูลและบรรจุลงในหน่วยความจำข้อมูล (DATA MEMORY) ตามค่า ADDRESS ที่อยู่ใน FILE เมื่อใช้คำสั่งนี้ ANT-32 จะรอรับข้อมูลทางสาย RS232 ให้ผู้ใช้ส่งข้อมูลจากเครื่องคอมพิวเตอร์ PC ไปยัง ANT-32 โดยใช้คำสั่ง SE ในโปรแกรม XTALK ได้ทันที ดังตัวอย่างต่อไปนี้

```
>XDOWN                    (กดคีย์ ENTER)
Wait for DOWNLOAD Intel Hex file.....
```

จากนั้นกด Ctrl-A เพื่อเข้า COMMAND MODE ของโปรแกรม XTALK แล้วใช้คำสั่ง SE เพื่อส่งข้อมูลโดยที่ ANT-32 จะรอรับ HEX FILE จากเครื่องคอมพิวเตอร์ PC และเมื่อรับจนจบไฟล์แล้วข้อมูลจะถูกเก็บไว้ใน หน่วยความจำข้อมูล (DATA MEMORY) ตามตำแหน่งที่ระบุใน HEX FILE นั้น

ถ้าต้องการให้ข้อมูลเก็บที่แอดเดรสอื่น ๆ ที่ไม่ตรงกับค่าแอดเดรสที่ระบุใน HEX FILE นั้น ให้ใช้คำสั่ง XDOWN โดยระบุค่า OFFSET วิธีการคำนวณค่า OFFSET มีดังนี้

OFFSET = แอดเดรสเริ่มต้นที่ต้องการ LOAD-แอดเดรสเริ่มต้นของ HEX FILE

เช่นที่ HEX FILE ระบุแอดเดรสเริ่มต้นเป็น 2000H และต้องการ DOWNLOAD ข้อมูลมาเก็บไว้ที่ ANT-32 โดยมีแอดเดรสเริ่มต้นเป็น 8000H จะคำนวนค่า OFFSET ดังนี้

OFFSET = 8000H - 2000H = 6000H

```
>XDOWN 6000                    (กดคีย์ Enter)
Wait for DOWNLOAD Intel Hex file.....
```

5.1.7 เมื่อต้องการทดสอบโปรแกรม ให้ใช้คำสั่ง CALL ของเบสิค เรียกโปรแกรมแอสเซมบลี้ให้ทำงาน โดยใช้ CALL 8000H

5.2 LADDER DIAGRAM ที่ทดลอง

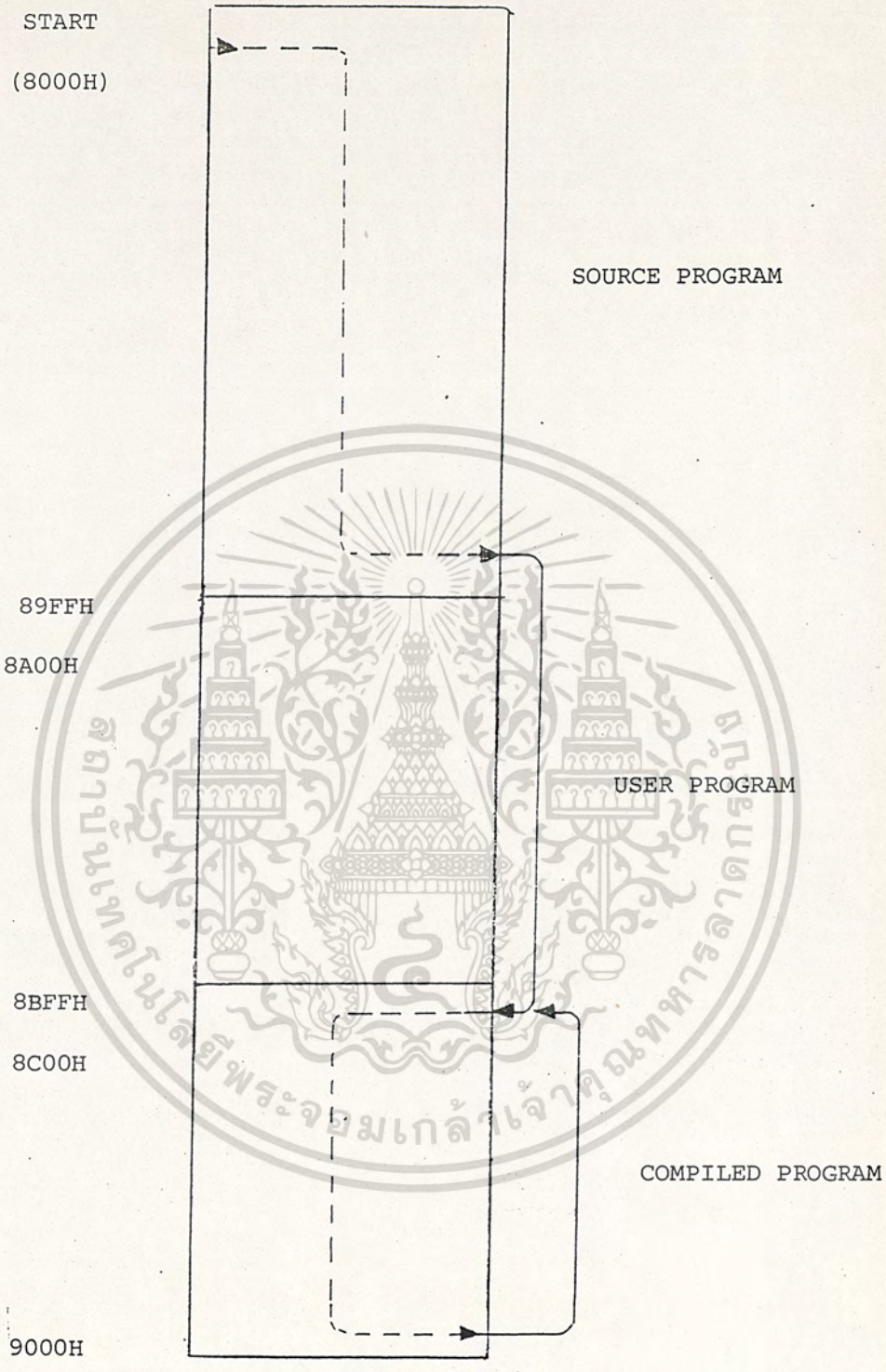
แลดเดอร์ ไดอะแกรม ที่ใช้ในการทดลอง เป็นตัวอย่างในการอธิบาย ดังรูปที่ 5.2

5.3 การทำงานของโปรแกรมที่พัฒนาเป็นเครื่องควบคุมที่โปรแกรมได้

(ดูรายละเอียดของโปรแกรม จากภาคผนวก 3)

เริ่มต้นการทำงานที่ address 8000 H เป็นการ SET PORT ตามโปรแกรมข้างล่างนี้

```
ORG     8000H
MOV     DPH,#0FCH       ; SET PORT
MOV     DPL,#03H
MOV     A,#98H
MOVX    @DPTR,A
```

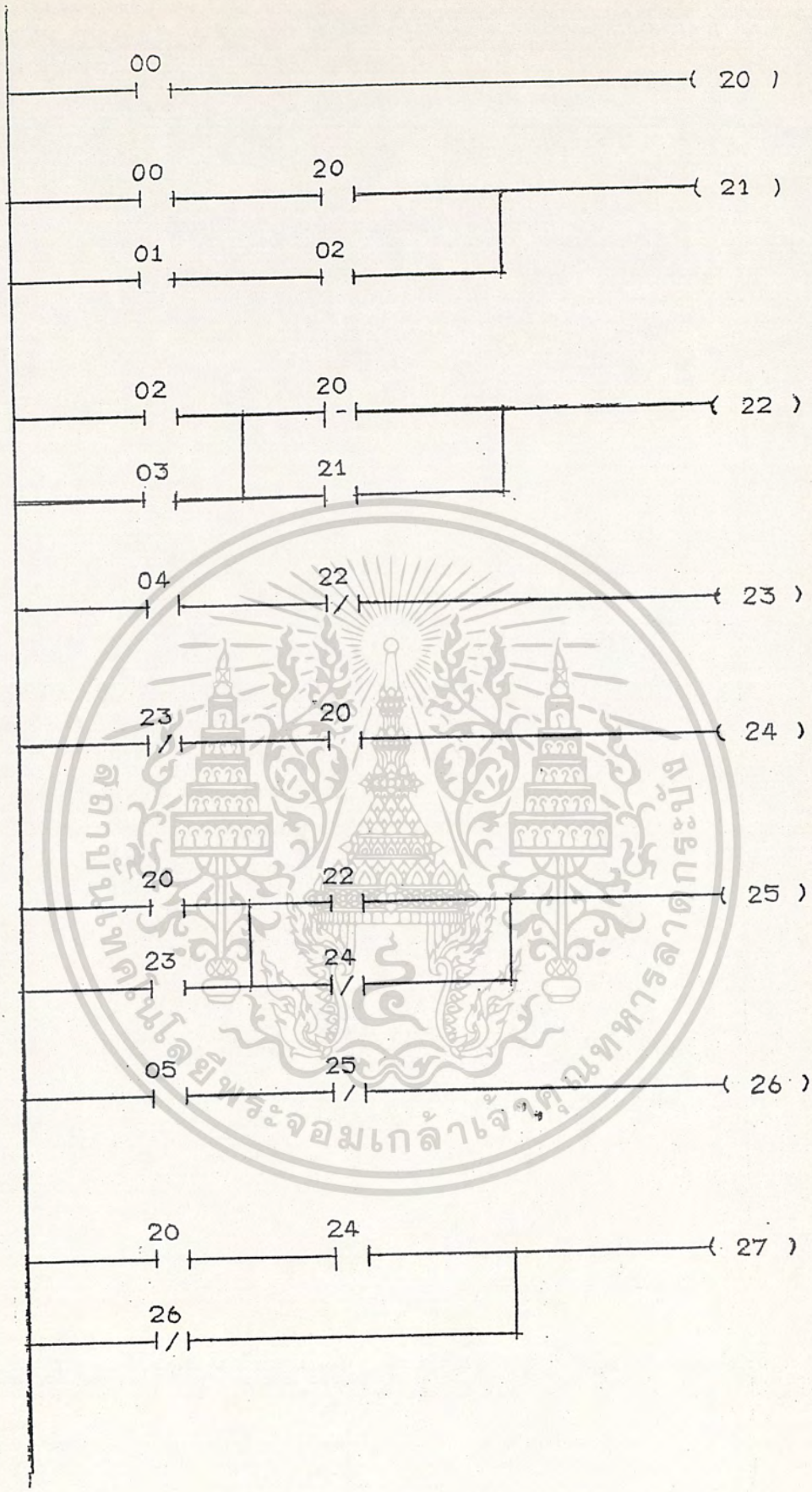


รูปที่ 5.1 MEMORY MAP

```
MOV         77H,#8AH
MOV         76H,#00H
MOV         75H,#8CH
MOV         74H,#11H
```

จากนั้นเป็นการ check COMMAND CODE ว่าเป็น CODE อะไร จะสั่งให้เครื่องควบคุมที่โปรแกรมได้ ทำงานตามฟังก์ชั่นไหน ซึ่ง COMMAND CODE เหล่านี้จะเก็บไว้ใน MEMORY ส่วนที่ 2 ที่เป็น USER PROGRAM ตั้งแต่ address ที่ 8A000H ถึง 8BFFH ดังตัวอย่าง USER PROGRAM ข้างล่างนี้

```
ORG         8A00H
DB          00H         ; LD 00
DB          00H
DB          08H         ; OUT 20
DB          20H
DB          00H         ; LD 00
DB          00H
DB          02H         ; AND 20
DB          20H
DB          00H         ; LD 01
DB          01H
DB          02H         ;AND 02
DB          02H
DB          07H         ;OR LD
DB          08H         ;OUT 21
DB          10H         ;END
END
```



รูปที่ 5.2 LADDER DIAGRAM

(ดูตารางของ COMMAND CODE ว่า CODE อะไรคือ COMMAND อะไร ได้จากภาคผนวกที่ 4)

ในการ check ว่าเป็น CODE อะไรนั้น จะเริ่มจาก การ check ว่าเป็น CODE 00 หรือไม่

ถ้าใช่ก็จะทำงานต่อลงไป โดยจะ JUMP ไปทำงานใน SUBROUTINE เช่น LOOP1, LOOPA, LOOP2 เพื่อสร้างข้อมูลและคำสั่งใหม่ เก็บไว้ใน MEMORY ช่วงที่ 3 (ดูรูป 5.1) ตั้งแต่ address 8C00H ถึง 9000H เสร็จแล้วจะย้อนกลับมาทำงานที่ AGAIN อีก เพื่อ check COMMAND ต่อไป ดังตัวอย่างข้างล่างนี้

```
AGAIN:      MOV     DPH,77H     ; CHECK COMMAND CODE
            MOV     DPL,76H
            MOVX    A, @DPTR
            CJNE    A, #00H,NOT_EQ0
            LCALL   LOOP1
            MOV     DPTR,#LDAT
            LCALL   LOOPA
            LCALL   LOOP2
            LJMP    AGAIN
```

แต่ถ้าการ check ว่าเป็น CODE 00 หรือไม่ ปรากฏว่าไม่ใช่ CODE 00 จะ JUMP ไปที่ NOT EQUAL 0 เพื่อ check ว่าเป็น COMMAND CODE 01 หรือไม่

```
NOT_EQ0:    CJNE    A,#02H,NOT_EQ1
            LCALL   LOOP1
            MOV     DPTR,#ANDE
            LCALL   LOOPA
            LCALL   LOOP2
            LJMP    AGAIN
```

```
NOT_EQ1:    CJNE    A,#04H,NOT_EQ2
            LCALL   LOOP1
            MOV     DPTR,#ORAT
            LCALL   LOOPA
            LCALL   LOOP2
            LJMP    AGAIN
```

การทำงานต่อมาจะมีลักษณะเช่นเดียวกับการ CHECK CODE 00 และจะ LOOP กลับไปทำงานที่ AGAIN เพื่อ check CODE อื่น ๆ ต่อไป

5.4 ผลของการทดลอง

การทดลองตามตัวอย่างของ LADDER DIAGRAM ตามรูปที่ 5.2 สามารถทำงานได้ เช่น เมื่อมี INPUT เข้ามาที่ 00 จะได้ สภาวะของ OUTPUT ที่ 20 ตามสภาวะของ INPUT 00

การทดลองนี้ได้ใช้ COMMAND ต่าง ๆ ตาม COMMAND CODE ของภาคผนวก 4 และใช้ OUTPUT ทั้งหมด 8 OUTPUT ซึ่งก็สามารถทำงานตามสภาวะของ INPUT ได้

บทที่ 6

วิจารณ์และสรุปผลการทดลอง

เครื่องควบคุมที่โปรแกรมได้ เป็นการประยุกต์ใช้ไมโครโปรเซสเซอร์ในการควบคุมกระบวนการที่มีประสิทธิภาพวิธีหนึ่ง โครงงานนี้จึงได้พัฒนาโปรแกรมเพื่อใช้กับไมโครคอนโทรลเลอร์ 8032 ซึ่งมีประสิทธิภาพสูง มีความแน่นอนและทำงานได้รวดเร็ว เพื่อให้เป็นเครื่องควบคุมที่โปรแกรมได้ ซึ่งสามารถเปลี่ยนแปลงการควบคุมโดยอาศัยคอมพิวเตอร์ส่วนบุคคล เข้าช่วย

โครงงานนี้พัฒนาโปรแกรมโดยใช้ภาษาแอสเซมบลี้ ซึ่งถ้าเปรียบเทียบระหว่าง 8032 กับ Z80 แล้ว การเขียน SOFTWARE จะลำบากกว่าของ Z80 เนื่องจากความไม่คล่องตัวของ INSTRUCTION หลายตัว โดยเฉพาะการประมวลผลที่อ้าง ADDRESS ขนาด 16 บิท Z80 จะทำได้คล่องตัวกว่า

นอกจากนี้การพัฒนายังได้ฟังก์ชั่นไม่มากนัก เพราะเครื่องควบคุมที่โปรแกรมได้จะมีฟังก์ชั่นพิเศษต่าง ๆ อีกมาก

อย่างไรก็ตาม การพัฒนาโดยการใช้ 8032 จะได้ เสถียรภาพ ในการทำงานที่แน่นอนกว่า

กิตติกรรมประกาศ

ปริญญานิพนธ์ฉบับนี้ สำเร็จลงได้ก็ด้วยความร่วมมือหลายฝ่ายด้วยกันทางคณะผู้จัดทำจึงขอขอบพระคุณ อาจารย์ วิริยะ กองรัตน์ ซึ่งเป็นอาจารย์ที่ปรึกษา และ อาจารย์หลายท่านในภาควิชาเทคโนโลยีการวัดคุมทางอุตสาหกรรมที่ได้ช่วยแนะนำ จนโครงงานนี้สำเร็จลุล่วงได้

เอกสารอ้างอิง

REFERENCES

1. วิริยะ กองรัตน์, "เครื่องควบคุมที่โปรแกรมได้", วิทยานิพนธ์ สำหรับปริญญาวิศวกรรมศาสตรมหาบัณฑิต, สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง, 2531.
2. สุพรรณ กุลพาณิชย์, PROGRAMMABLE CONTROLLER เทคนิคและการใช้งานเบื้องต้น เล่ม 1, โรงพิมพ์ทิพยวิสุทธิ์, 2533.
3. Intel, "MCS-51 MICROCONTROLLERS DATA BOOK", Intel Corporation, 1988.
4. รศ. ยืน ภู่วรวรรณ & ดร.ชัยยงค์ วงค์ชัยสุวัฒน์ & น.ท. ดร. ไพศาล สงวนหมู่, "เทคโนโลยีไมโครคอมพิวเตอร์ 16 บิท", บริษัท ซีเอ็ดยูเคชั่น จำกัด, 2530
5. THOMAS E. KISSELL, "UNDERSTANDING AND USING PROGRAMMABLE CONTROLLERS", Prentice-Hall International , Inc., 1986
6. Omron, Sysmac Programmable Controllers C20P / C28P / C40P Installation Guide, Omron Tateisi Electronics Co.
7. Gould, Gould Introduction to Programmable Control Reference Manual, Gould Electronics
8. Modicon, Modicon Programmable Logic Controller, Modicon Inc., 1990

ภาคผนวก 1

MCS®-51 INSTRUCTION SET

Table 10. 8051 Instruction Set Summary

Interrupt Response Time: Refer to Hardware Description Chapter.

Instructions that Affect Flag Settings(1)

Instruction	Flag			Instruction	Flag		
	C	OV	AC		C	OV	AC
ADD	X	X	X	CLR C	0		
ADDC	X	X	X	CPL C	X		
SUBB	X	X	X	ANL C,bit	X		
MUL	0	X		ANL C,/bit	X		
DIV	0	X		ORL C,bit	X		
DA	X			ORL C,bit	X		
RRC	X			MOV C,bit	X		
RLC	X			CJNE	X		
SETB C	1						

(1)Note that operations on SFR byte address 208 or bit addresses 209-215 (i.e., the PSW or bits in the PSW) will also affect flag settings.

Note on instruction set and addressing modes:

Rn — Register R7–R0 of the currently selected Register Bank.

direct — 8-bit internal data location's address. This could be an Internal Data RAM location (0–127) or a SFR [i.e., I/O port, control register, status register, etc. (128–255)].

@Ri — 8-bit internal data RAM location (0–255) addressed indirectly through register R1 or R0.

#data — 8-bit constant included in instruction.

#data 16 — 16-bit constant included in instruction.

addr 16 — 16-bit destination address. Used by LCALL & LJMP. A branch can be anywhere within the 64K-byte Program Memory address space.

addr 11 — 11-bit destination address. Used by ACALL & AJMP. The branch will be within the same 2K-byte page of program memory as the first byte of the following instruction.

rel — Signed (two's complement) 8-bit offset byte. Used by SJMP and all conditional jumps. Range is −128 to +127 bytes relative to first byte of the following instruction.

bit — Direct Addressed bit in Internal Data RAM or Special Function Register.

Mnemonic		Description	Byte	Oscillator Period
ARITHMETIC OPERATIONS				
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
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 direct 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

Table 10. 8051 Instruction Set Summary (Continued)

Mnemonic		Description	Byte	Oscillator Period
ARITHMETIC OPERATIONS (Continued)				
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
LOGICAL OPERATIONS				
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
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

Mnemonic		Description	Byte	Oscillator Period
LOGICAL OPERATIONS (Continued)				
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
DATA TRANSFER				
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

Table 10. 8051 Instruction Set Summary (Continued)

Mnemonic		Description	Byte	Oscillator Period
DATA TRANSFER (Continued)				
MOV	@Ri,direct	Move direct byte to indirect RAM	2	24
MOV	@Ri, # data	Move immediate data to indirect RAM	2	12
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
BOOLEAN VARIABLE MANIPULATION				
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
PROGRAM BRANCHING				
ACALL	addr11	Absolute Subroutine Call	2	24
LCALL	addr16	Long Subroutine Call	3	24
RET		Return from Subroutine	1	24
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

Table 10. 8051 Instruction Set Summary (Continued)

Mnemonic		Description	Byte	Oscillator Period
PROGRAM BRANCHING (Continued)				
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	2	24
DJNZ	direct,rel	Decrement direct byte and Jump if Not Zero	3	24
NOP		No Operation	1	12

Table 11. Instruction Opcodes in Hexadecimal Order

Hex Code	Number of Bytes	Mnemonic	Operands
00	1	NOP	
01	2	AJMP	code addr
02	3	LJMP	code addr
03	1	RR	A
04	1	INC	A
05	2	INC	data addr
06	1	INC	@R0
07	1	INC	@R1
08	1	INC	R0
09	1	INC	R1
0A	1	INC	R2
0B	1	INC	R3
0C	1	INC	R4
0D	1	INC	R5
0E	1	INC	R6
0F	1	INC	R7
10	3	JBC	bit addr, code addr
11	2	ACALL	code addr
12	3	LCALL	code addr
13	1	RRC	A
14	1	DEC	A
15	2	DEC	data addr
16	1	DEC	@R0
17	1	DEC	@R1
18	1	DEC	R0
19	1	DEC	R1
1A	1	DEC	R2
1B	1	DEC	R3
1C	1	DEC	R4
1D	1	DEC	R5
1E	1	DEC	R6
1F	1	DEC	R7
20	3	JB	bit addr, code addr
21	2	AJMP	code addr
22	1	RET	
23	1	RL	A
24	2	ADD	A, #data
25	2	ADD	A,data addr
26	1	ADD	A,@R0
27	1	ADD	A,@R1
28	1	ADD	A,R0
29	1	ADD	A,R1
2A	1	ADD	A,R2
2B	1	ADD	A,R3
2C	1	ADD	A,R4
2D	1	ADD	A,R5
2E	1	ADD	A,R6
2F	1	ADD	A,R7
30	3	JNB	bit addr, code addr
31	2	ACALL	code addr
32	1	RETI	
33	1	RLC	A
34	2	ADDC	A, #data
35	2	ADDC	A,data addr
36	1	ADDC	A,@R0
37	1	ADDC	A,@R1
38	1	ADDC	A,R0
39	1	ADDC	A,R1
3A	1	ADDC	A,R2
3B	1	ADDC	A,R3
3C	1	ADDC	A,R4
3D	1	ADDC	A,R5
3E	1	ADDC	A,R6
3F	1	ADDC	A,R7
40	2	JC	code addr
41	2	AJMP	code addr
42	2	ORL	data addr,A
43	3	ORL	data addr, #data
44	2	ORL	A, #data
45	2	ORL	A,data addr
46	1	ORL	A,@R0
47	1	ORL	A,@R1
48	1	ORL	A,R0
49	1	ORL	A,R1
4A	1	ORL	A,R2
4B	1	ORL	A,R3
4C	1	ORL	A,R4
4D	1	ORL	A,R5
4E	1	ORL	A,R6
4F	1	ORL	A,R7
50	2	JNC	code addr
51	2	ACALL	code addr
52	2	ANL	data addr,A
53	3	ANL	data addr, #data
54	2	ANL	A, #data
55	2	ANL	A,data addr
56	1	ANL	A,@R0
57	1	ANL	A,@R1
58	1	ANL	A,R0
59	1	ANL	A,R1
5A	1	ANL	A,R2
5B	1	ANL	A,R3
5C	1	ANL	A,R4
5D	1	ANL	A,R5
5E	1	ANL	A,R6
5F	1	ANL	A,R7
60	2	JZ	code addr
61	2	AJMP	code addr
62	2	XRL	data addr,A
63	3	XRL	data addr, #data
64	2	XRL	A, #data
65	2	XRL	A,data addr

Table 11. Instruction Opcodes in Hexadecimal Order (Continued)

Hex Code	Number of Bytes	Mnemonic	Operands
66	1	XRL	A,@R0
67	1	XRL	A,@R1
68	1	XRL	A,R0
69	1	XRL	A,R1
6A	1	XRL	A,R2
6B	1	XRL	A,R3
6C	1	XRL	A,R4
6D	1	XRL	A,R5
6E	1	XRL	A,R6
6F	1	XRL	A,R7
70	2	JNZ	code addr
71	2	ACALL	code addr
72	2	ORL	C,bit addr
73	1	JMP	@A + DPTR
74	2	MOV	A, #data
75	3	MOV	data addr, #data
76	2	MOV	@R0, #data
77	2	MOV	@R1, #data
78	2	MOV	R0, #data
79	2	MOV	R1, #data
7A	2	MOV	R2, #data
7B	2	MOV	R3, #data
7C	2	MOV	R4, #data
7D	2	MOV	R5, #data
7E	2	MOV	R6, #data
7F	2	MOV	R7, #data
80	2	SJMP	code addr
81	2	AJMP	code addr
82	2	ANL	C,bit addr
83	1	MOVC	A,@A + PC
84	1	DIV	AB
85	3	MOV	data addr, data addr
86	2	MOV	data addr,@R0
87	2	MOV	data addr,@R1
88	2	MOV	data addr,R0
89	2	MOV	data addr,R1
8A	2	MOV	data addr,R2
8B	2	MOV	data addr,R3
8C	2	MOV	data addr,R4
8D	2	MOV	data addr,R5
8E	2	MOV	data addr,R6
8F	2	MOV	data addr,R7
90	3	MOV	DPTR, #data
91	2	ACALL	code addr
92	2	MOV	bit addr,C
93	1	MOVC	A,@A + DPTR
94	2	SUBB	A, #data
95	2	SUBB	A,data addr
96	1	SUBB	A,@R0
97	1	SUBB	A,@R1
98	1	SUBB	A,R0
99	1	SUBB	A,R1
9A	1	SUBB	A,R2
9B	1	SUBB	A,R3
9C	1	SUBB	A,R4
9D	1	SUBB	A,R5
9E	1	SUBB	A,R6
9F	1	SUBB	A,R7
A0	2	ORL	C,/bit addr
A1	2	AJMP	code addr
A2	2	MOV	C,bit addr
A3	1	INC	DPTR
A4	1	MUL	AB
A5		reserved	
A6	2	MOV	@R0,data addr
A7	2	MOV	@R1,data addr
A8	2	MOV	R0,data addr
A9	2	MOV	R1,data addr
AA	2	MOV	R2,data addr
AB	2	MOV	R3,data addr
AC	2	MOV	R4,data addr
AD	2	MOV	R5,data addr
AE	2	MOV	R6,data addr
AF	2	MOV	R7,data addr
B0	2	ANL	C,/bit addr
B1	2	ACALL	code addr
B2	2	CPL	bit addr
B3	1	CPL	C
B4	3	CJNE	A, #data,code addr
B5	3	CJNE	A,data addr,code addr
B6	3	CJNE	@R0, #data,code addr
B7	3	CJNE	@R1, #data,code addr
B8	3	CJNE	R0, #data,code addr
B9	3	CJNE	R1, #data,code addr
BA	3	CJNE	R2, #data,code addr
BB	3	CJNE	R3, #data,code addr
BC	3	CJNE	R4, #data,code addr
BD	3	CJNE	R5, #data,code addr
BE	3	CJNE	R6, #data,code addr
BF	3	CJNE	R7, #data,code addr
C0	2	PUSH	data addr
C1	2	AJMP	code addr
C2	2	CLR	bit addr
C3	1	CLR	C
C4	1	SWAP	A
C5	2	XCH	A,data addr
C6	1	XCH	A,@R0
C7	1	XCH	A,@R1
C8	1	XCH	A,R0
C9	1	XCH	A,R1
CA	1	XCH	A,R2
CB	1	XCH	A,R3

Table 11. Instruction Opcodes in Hexadecimal Order (Continued)

Hex Code	Number of Bytes	Mnemonic	Operands
CC	1	XCH	A,R4
CD	1	XCH	A,R5
CE	1	XCH	A,R6
CF	1	XCH	A,R7
D0	2	POP	data addr
D1	2	ACALL	code addr
D2	2	SETB	bit addr
D3	1	SETB	C
D4	1	DA	A
D5	3	DJNZ	data addr,code addr
D6	1	XCHD	A,@R0
D7	1	XCHD	A,@R1
D8	2	DJNZ	R0,code addr
D9	2	DJNZ	R1,code addr
DA	2	DJNZ	R2,code addr
DB	2	DJNZ	R3,code addr
DC	2	DJNZ	R4,code addr
DD	2	DJNZ	R5,code addr
DE	2	DJNZ	R6,code addr
DF	2	DJNZ	R7,code addr
E0	1	MOVX	A,@DPTR
E1	2	AJMP	code addr
E2	1	MOVX	A,@R0
E3	1	MOVX	A,@R1
E4	1	CLR	A
E5	2	MOV	A,data addr
E6	1	MOV	A,@R0
E7	1	MOV	A,@R1
E8	1	MOV	A,R0
E9	1	MOV	A,R1
EA	1	MOV	A,R2
EB	1	MOV	A,R3
EC	1	MOV	A,R4
ED	1	MOV	A,R5
EE	1	MOV	A,R6
EF	1	MOV	A,R7
F0	1	MOVX	@DPTR,A
F1	2	ACALL	code addr
F2	1	MOVX	@R0,A
F3	1	MOVX	@R1,A
F4	1	CPL	A
F5	2	MOV	data addr,A
F6	1	MOV	@R0,A
F7	1	MOV	@R1,A
F8	1	MOV	R0,A
F9	1	MOV	R1,A
FA	1	MOV	R2,A
FB	1	MOV	R3,A
FC	1	MOV	R4,A
FD	1	MOV	R5,A
FE	1	MOV	R6,A
FF	1	MOV	R7,A

INSTRUCTION DEFINITIONS

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, opcode 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 0345 H. 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

$(PC) \leftarrow (PC) + 2$
$(SP) \leftarrow (SP) + 1$
$((SP)) \leftarrow (PC_{7\text{-}0})$
$(SP) \leftarrow (SP) + 1$
$((SP)) \leftarrow (PC_{15\text{-}8})$
$(PC_{10\text{-}0}) \leftarrow$ 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 occured.

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 occured.

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 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) ← (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) ← (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
$(PC) \leftarrow (PC) + 2$
$(PC_{10\text{-}0}) \leftarrow$ 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 55H (01010101B) 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 direct, #data

Bytes: 3

Cycles: 2

Encoding:

0 1 0 1	0 0 1 1	direct address	immediate data

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 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:

1 0 1 1	0 1 0 0	immediate data	rel. address

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:

1 0 1 1	1 r r r	immediate data	rel. address

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:

1 0 1 1	0 1 1 i	immediate data	rel. address

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 on 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 a 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] V [(AC) = 1]]
      THEN(A3-0) ← (A3-0) + 6
           AND

IF    [[(A7-4) > 9] V [(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
$(A)_{15\text{-}8}$
$(B)_{7\text{-}0}$ ← (A)/(B)

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:

1 1 0 1	1 r r r	rel. address

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:

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

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.

Example: Register 0 contains 7EH (01111110B). Internal RAM locations 7EH and 7FH contain 0FFH and 40H, respectively. The instruction sequence.

```
INC  @R0
INC  R0
INC  @R0
```

will leave register 0 set to 7FH and internal RAM locations 7EH and 7FH holding (respectively) 00H and 41H.

INC A

Bytes: 1

Cycles: 1

Encoding:

0 0 0 0	0 1 0 0

Operation:

```
INC
(A) ← (A) + 1
```

INC Rn

Bytes: 1

Cycles: 1

Encoding:

0 0 0 0	1 r r r

Operation: INC
(Rn) ← (Rn) + 1

INC direct

Bytes: 2

Cycles: 1

Encoding:

0 0 0 0	0 1 0 1	direct address

Operation: INC
(direct) ← (direct) + 1

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,

```
INC   DPTR
INC   DPTR
INC   DPTR
```

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. *The bit will not be cleared if it is already a zero.* 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:

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

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 1234H.

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)) ← (PC7-0)
(SP) ← (SP) + 1
((SP)) ← (PC15-8)
(PC) ← addr15-0
```

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) ← addr15-0
```

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:

1 1 1 0	0 1 1 i

Operation: MOV
(A) ← ((Ri))

MOV A,#data

Bytes: 2

Cycles: 1

Encoding:

0 1 1 1	0 1 0 0	immediate data

Operation: MOV
(A) ← #data

MOV Rn,A

Bytes: 1

Cycles: 1

Encoding:

1 1 1 1	1 r r r

Operation: MOV
(Rn) ← (A)

MOV Rn,direct

Bytes: 2

Cycles: 2

Encoding:

1 0 1 0	1 r r r	direct addr.

Operation: MOV
(Rn) ← (direct)

MOV Rn,#data

Bytes: 2

Cycles: 1

Encoding:

0 1 1 1	1 r r r	immediate data

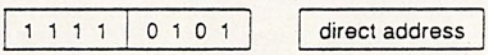
Operation: MOV
(Rn) ← #data

MOV direct,A

Bytes: 2

Cycles: 1

Encoding:



Operation: MOV
(direct) ← (A)

MOV direct,Rn

Bytes: 2

Cycles: 2

Encoding:

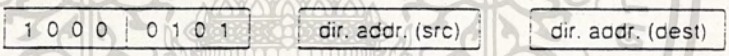
Operation: MOV
(direct) ← (Rn)

MOV direct,direct

Bytes: 3

Cycles: 2

Encoding:



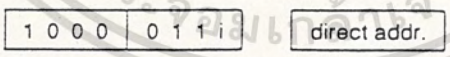
Operation: MOV
(direct) ← (direct)

MOV direct,@RI

Bytes: 2

Cycles: 2

Encoding:



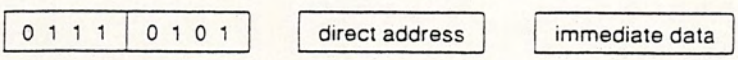
Operation: MOV
(direct) ← ((Ri))

MOV direct, # data

Bytes: 3

Cycles: 2

Encoding:



Operation: MOV
(direct) ← # data

MOV @Ri,A

Bytes: 1

Cycles: 1

Encoding:

1 1 1 1	0 1 1 i

Operation: MOV
((Ri)) ← (A)

MOV @Ri,direct

Bytes: 2

Cycles: 2

Encoding:

1 0 1 0	0 1 1 i	direct addr.

Operation: MOV
((Ri)) ← (direct)

MOV @Ri,#data

Bytes: 2

Cycles: 1

Encoding:

0 1 1 1	0 1 1 i	immediate data

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:

1 0 1 0	0 0 1 0	bit address

Operation: MOV
(C) ← (bit)

MOV bit,C

Bytes: 2

Cycles: 2

Encoding:

1 0 0 1	0 0 1 0	bit address

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:

1 0 0 1	0 0 0 0	immed. data15-8	immed. data7-0

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

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 for 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)

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

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:

1 0 0 1	0 1 0 1	direct address

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

SUBB A,@Ri

Bytes: 1

Cycles: 1

Encoding:

1 0 0 1	0 1 1 i

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

SUBB A,#data

Bytes: 2

Cycles: 1

Encoding:

1 0 0 1	0 1 0 0	immediate data

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:

1 1 0 0	0 1 0 0

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

ภาคผนวก ค

การใช้งานโปรแกรม XTALK

โปรแกรม XTALK คือโปรแกรมสำหรับการสื่อสารทั่วไป โดยในที่นี้จะใช้สื่อสารในแบบ LOCAL ซึ่งเป็นการสื่อสารโดยผ่านสาย RS232 โปรแกรมนี้มีรายละเอียดค่อนข้างมาก แต่ในที่นี้จะสรุปการใช้งานพอสังเขป และจะเน้นที่การใช้งานกับ ANT-32 เป็นแนวทางหลัก โดยพอจะสรุปเป็นข้อ ๆ ได้ดังนี้

1. เป็นการใช้งานตามการกำหนดคือ BAUD RATE = 9600, STOP BIT = 1, PARITY = NONE DATA = 8, PORT = COM1, CWAIT = DELAY 2, LOCAL COMMUNICATION

2. การใช้งานมีโหมดต่าง ๆ คือ
   - โหมดการสื่อสาร โดยจะอยู่ในขบวนการรับและส่งข้อมูลทางสาย และที่บรรทัดล่างสุดจะแสดงสถานะบางอย่าง พร้อมทั้งแสดงคำสั่ง (^A) ในการเปลี่ยนโหมดด้วยโหมด COMMAND จะเป็นการรับคำสั่งต่าง ๆ ของ XTALK ในจุดนี้ผู้ใช้จะขอดูตัวแปรต่าง ๆ ของการสื่อสารได้ด้วยคำสั่ง ^F คำสั่งทั้งหมดของ XTALK จะดูได้ด้วยคำสั่ง HE (HELP) ในโหมดนี้สิ่งต่าง ๆ ที่ปรากฏบนจอจะเกิดขึ้นจาก XTALK เอง ไม่ใช่สิ่งที่มาจากการสื่อสาร ผู้ใช้จะกลับไปโหมดการสื่อสารได้ด้วยการกด ENTER หรือคำสั่ง GO LOCAL

3. การใช้คำสั่ง CA (CAPTURE) เพื่อการเก็บสิ่งต่าง ๆ ที่ปรากฏบนจอลงใน FILE จะทำได้โดยใช้ CA และตามด้วยชื่อ FILE หลังจากนั้นสิ่งที่เกิดขึ้นบนจอทุกอย่าง (ที่มาจากการสื่อสาร) จะถูกเก็บลงใน FLIE และผู้ใช้จะสิ้นสุดขบวนการได้ด้วยคำสั่ง CA OFF กรณีนี้จะใช้สำหรับการเก็บโปรแกรมภาษา BASIC-52 ในแบบ TEXT FILE ได้

4. คำสั่ง SE (SEND) ใช้สำหรับการส่งข้อมูลจาก FILE ออกทางสาย ซึ่งเป็นขบวนการ DOWN LOAD จากเครื่อง PC ไปยัง ANT-32 การใช้คำสั่ง SE และ CA จะช่วยให้การใช้งานสะดวกมากยิ่งขึ้น ซึ่งสามารถจะทำการ UP และ DOWN LOAD ข้อมูลไปมา ทำให้การเก็บโปรแกรมหรือข้อมูลต่าง ๆ กระทำอยู่บนเครื่อง PC ได้

ภาคผนวก ง

```
                ORG     8000H
                MOV     DPH,#0FCH       ; SET PORT
                MOV     DPL,#03H
                MOV     A,#98H
                MOVX    @DPTR,A


                MOV     77H,#8AH
                MOV     76H,#00H
                MOV     75H,#8CH
                MOV     74H,#11H

AGAIN:          MOV     DPH,77H         ;CHECK COMMAND CODE
                MOV     DPL,76H
                MOVX    A,@DPTR
                CJNE    A,#00H,NOT_EQ0
                LCALL   LOOP1
                MOV     DPTR,#LDAT
                LCALL   LOOPA
                LCALL   LOOP2
                LJMP    AGAIN

NOT_EQ0:        CJNE    A,#02H,NOT_EQ1
                LCALL   LOOP1
                MOV     DPTR,#ANDE
                LCALL   LOOPA
                LCALL   LOOP2
                LJMP    AGAIN

NOT_EQ1:        CJNE    A,#04H,NOT_EQ2
                LCALL   LOOP1
                MOV     DPTR,#ORAT
                LCALL   LOOPA
                LCALL   LOOP2
                LJMP    AGAIN


NOT_EQ2:        CJNE    A,#08H,NOT_EQ3
                LCALL   LOOP1
                MOV     DPTR,#OUTS
                LCALL   LOOPA
                LCALL   LOOP2
                LJMP    AGAIN
```

```
NOT_EQ3:    CJNE    A,#10H,NOT_EQ4
            LCALL   LOOP3
            MOV     DPTR,#ENDT
            MOV     70H,DPL         ;DPH
            MOV     6FH,DPH         ;DPL
            MOV     DPH,75H
            MOV     DPL,74H
            MOV     A,6FH
            MOVX    @DPTR,A
            INC     DPTR
            MOV     A,70H
            MOVX    @DPTR,A
            INC     DPTR
            MOV     A,#02H          ;THIS  IS  CODE  OF  LJMP
            MOVX    @DPTR,A
            INC     DPTR
            MOV     A,#8CH
            MOVX    @DPTR,A
            INC     DPTR
            MOV     A,#00H
            MOVX    @DPTR,A
            INC     DPTR
            MOV     75H,DPH
            MOV     74H,DPL
            LJMP    8C00H

;_____________________________________________________________
NOT_EQ4:    CJNE    A,#01H,NOT_EQ5
            LCALL   LOOP1
            MOV     DPTR,#LDOT
            LCALL   LOOPA
            LCALL   LOOP2
            LJMP    AGAIN

NOT_EQ5:    CJNE    A,#03H,NOT_EQ6
            LCALL   LOOP1
            MOV     DPTR,#ANOT
            LCALL   LOOPA
            LCALL   LOOP2
            LJMP    AGAIN

NOT_EQ6:    CJNE    A,#05H,NOT_EQ7
            LCALL   LOOP1
            MOV     DPTR,#OROT
            LCALL   LOOPA
            LCALL   LOOP2
            LJMP    AGAIN
```

```
NOT_EQ7:    CJNE    A,#09H,NOT_EQ8
            LCALL   LOOP1
            MOV     DPTR,#OUTN
            LCALL   LOOPA
            LCALL   LOOP2
            LJMP    AGAIN

NOT_EQ8:    CJNE    A,#06H,NOT_EQ9
            LCALL   TEM
            LCALL   LOOP3
            MOV     DPTR,#ANLD
            LCALL   LEV
            LJMP    AGAIN

NOT_EQ9:    CJNE    A,#07H,NOT_EQ10
            LCALL   TEM
            LCALL   LOOP3
            MOV     DPTR,#ORLD
            LCALL   LEV
            LJMP    AGAIN
NOT_EQ10:   LJMP    NOT_EQ10


TEM:        MOV     DPH,77H
            MOV     DPL,76H
            INC     DPTR
            MOV     77H,DPH
            MOV     76H,DPL
            RET


LEV:        MOV     6EH,DPH
            MOV     6DH,DPL
            MOV     DPH,75H
            MOV     DPL,74H
            MOV     A,6EH
            MOVX    @DPTR,A
            INC     DPTR
            MOV     A,6DH
            MOVX    @DPTR,A
            INC     DPTR
            MOV     75H,DPH
            MOV     74H,DPL
            RET
```

```
LOOP3:      MOV     A,#12H          ;12 IS  CODE  OF  LCALL
            MOV     DPH,75H
            MOV     DPL,74H
            MOVX    @DPTR,A
            INC     DPTR
            MOV     75H,DPH
            MOV     74H,DPL
            RET

LOOPA:      MOV     78H,DPL         ;DPH
            MOV     73H,DPH         ;DPL
            MOV     DPH,75H
            MOV     DPL,74H
            MOV     A,73H
            MOVX    @DPTR,A
            INC     DPTR
            MOV     A,78H
            RET

LOOP2:      MOVX    @DPTR,A
            INC     DPTR
            MOV     75H,DPH
            MOV     74H.DPL
            RET

LOOP1:      MOV     DPH,75H
            MOV     DPL,74H
            MOV     A,#78H    ;THIS IS CODE OF   MOV  R0,# ADD_
            MOVX    @DPTR,A
            INC     DPTR
            MOV     75H,DPH
            MOV     74H,DPL
            MOV     DPH,77H
            MOV     DPL,76H
            INC     DPTR
            MOVX    A,@DPTR
            INC     DPTR
            MOV     77H,DPH
            MOV     76H,DPL
            MOV     DPH,75H
            MOV     DPL,74H
            MOVX    @DPTR,A
            INC     DPTR
            MOV     A,#12H
            MOVX    @DPTR,A
            INC     DPTR
            MOV     75H,DPH
            MOV     74H,DPL
            RET
```

```
LDOT:   POP     7FH         ;SAVE HIGH BYTE IN TO 7FH    LDAT
        POP     7EH         ;SAVE LOW BYTE IN TO 7EH
        MOV     A,R0
        SWAP    A
        ANL     A,#0FH
        CJNE    A,#00H,INRC1            ;NEW WAVE
        ADD     A,#20H
        MOV     R1,A        ;SAVE A
        MOV     A,R0
        ANL     A,#0FH
        MOV     DPTR,#MASK  ;BRING ADDRESS OF MASK IN TO DPT
        MOVC    A,@A+DPTR
        ANL     A,@R1
        JNZ     SEX         ;JUMP IF ACC IS NOT ZERO
        MOV     A,#0FFH     ;#00H
        PUSH    ACC
        PUSH    7EH
        PUSH    7FH
        RET

SEX:    MOV     A,#00H      ; #0FFH
        PUSH    ACC
        PUSH    7EH
        PUSH    7FH
        RET

MASK:   DB      01H
        DB      02H
        DB      04H
        DB      08H
        DB      10H
        DB      20H
        DB      40H
        DB      80H

ANDE:   POP     7BH         ;SAVE HIGH BYTE IN TO 7BH
        POP     7AH
        POP     79H
        LCALL   LDAT
        POP     ACC         ;NO  DATA IN SP NOW
        ANL     A,79H
        PUSH    ACC
        PUSH    7AH
        PUSH    7BH
        RET
```

```
ANOT:       POP     7BH
            POP     7AH
            POP     79H
            LCALL   LDAT
            POP     ACC
            CPL     A
            ANL     A,79H
            PUSH    ACC
            PUSH    7AH
            PUSH    7BH
            RET

SEX1:       LJMP    SEX

ORAT:       POP     7BH
            POP     7AH
            POP     79H
            LCALL   LDAT
            POP     ACC
            ORL     A,79H
            PUSH    ACC
            PUSH    7AH
            PUSH    7BH
            RET
 INRC1:     LJMP    INRC
;-------------------------------------------------------------------

OUTS:       POP     71H
            POP     72H
            POP     ACC

            ANL     A,0FFH
            JZ      RST_OUT

SET_OUT:    MOV     A,R0
            ANL     A,#0F0H
            CJNE    A,#20H,OUT1

            MOV     A,R0
            ANL     A,#0FH
            MOV     DPTR,#MASK
            MOVC    A,@A+DPTR
            MOV     R1,#22H
            ORL     A,@R1
            MOV     @R1,A
            PUSH    72H
            PUSH    71H
            RET
```

```
RST_OUT:        MOV     A,R0
                ANL     A,#0F0H
                CJNE    A,#20H,OUT2

                MOV     A,R0
                ANL     A,#0FH
                MOV     DPTR,#MASK
                MOVC    A,@A+DPTR
                CPL     A
                MOV     R1,#22H
                ANL     A,@R1
                MOV     @R1,A
                PUSH    72H
                PUSH    71H
                RET
;_____________________________________________________________

ENDT:           MOV     DPH,#0FCH       ;OUTPUT
                MOV     DPL,#01H
                MOV     A,22H
                MOVX    @DPTR,A
                RET

;_____________________________________________________________

   INT4:        MOV     A,R0
                ANL     A,#0FH
                MOV     DPTR,#MASK
                MOVC    A,@A+DPTR
                MOV     R1,#24H
                ANL     A,@R1
                JNZ     SET
                MOV     A,#00H
                PUSH    ACC
                PUSH    7EH
                PUSH    7FH
                RET


   INT2:        MOV     A,R0
                ANL     A,#0FH
                MOV     DPTR,#MASK
                MOVC    A,@A+DPTR
                MOV     R1,#24H
                ANL     A,@R1
                JNZ     SEX1
                MOV     A,#0FFH
                PUSH    ACC
                PUSH    7EH
                PUSH    7FH
                RET
```

```
OUT1:   MOV     A,R0
        ANL     A,#0FH
        MOV     DPTR,#MASK
        MOVC    A,@A+DPTR
        MOV     R1,#24H
        ORL     A,@R1
        MOV     @R1,A
        PUSH    72H
        PUSH    71H
        RET

OUT2:   MOV     A,R0
        ANL     A,#0FH
        MOV     DPTR,#MASK
        MOVC    A,@A+DPTR
        CPL     A
        MOV     R1,#24H
        ANL     A,@R1
        MOV     @R1,A
        PUSH    72H
        PUSH    71H
        RET

INT3:   LJMP    INT4

LDAT:   POP     7FH
        POP     7EH
        MOV     A,R0
        SWAP    A

        ANL     A,#0FH
        CJNE    A,#00H,INR0             ;NEW WAVE
        ADD     A,#20H
        MOV     R1,A
        MOV     A,R0
        ANL     A,#0FH
        MOV     DPTR,#MASK
        MOVC    A,@A+DPTR
        ANL     A,@R1
        JNZ     SET
        MOV     A,#00H
        PUSH    ACC
        PUSH    7EH
        PUSH    7FH
        RET
```

```
SET:    MOV     A,#0FFH
        PUSH    ACC
        PUSH    7EH
        PUSH    7FH
        RET

INRC:   MOV     A,R0
        ANL     A,#0F0H
        CJNE    A,#20H,INT2

        MOV     A,R0
        ANL     A,#0FH
        MOV     DPTR,#MASK
        MOVC    A,@A+DPTR
        MOV     R1,#22H
        ANL     A,@R1
        CJNE    A,#00H,NC
        MOV     A,#0FFH
        PUSH    ACC
        PUSH    7EH
        PUSH    7FH
        RET

NC:     MOV     A,#00H
        PUSH    ACC
        PUSH    7EH
        PUSH    7FH
        RET

INRO:   MOV     A,R0
        ANL     A,#0F0H
        CJNE    A,#20H,INT3

        MOV     A,R0
        ANL     A,#0FH
        MOV     DPTR,#MASK
        MOVC    A,@A+DPTR
        MOV     R1,#22H
        ANL     A,@R1
        CJNE    A,#00H,NO
        PUSH    ACC
        PUSH    7EH
        PUSH    7FH
        RET
```

```
NO:     MOV     A,#0FFH
        PUSH    ACC
        PUSH    7EH
        PUSH    7FH
        RET

OROT:   POP     7BH
        POP     7AH
        POP     79H
        LCALL   LDAT
        POP     ACC
        CPL     A
        ORL     A,79H
        PUSH    ACC
        PUSH    7AH
        PUSH    7BH
        RET

ORLD:   POP     37H
        POP     36H
        POP     69H
        POP     ACC
        ORL     A,69H
        PUSH    ACC
        PUSH    36H
        PUSH    37H
        RET

ANLD:   POP     39H
        POP     38H
        POP     6AH
        POP     ACC
        ANL     A,6AH
        PUSH    ACC
        PUSH    38H
        PUSH    39H
        RET

OUTN:   LJMP    OUTN


        ORG     8C00H
        MOV     DPTR,#0FC00H
        MOVX    A,@DPTR
        MOV     20H,A
        LJMP    8C11H
```

```
ORG     8A00H

DB      00H             ;LD 00
DB      00H

DB      08H             ;OUT 20
DB      20H

DB      00H             ;LD 00
DB      00H

DB      02H             ;AND 20
DB      20H

DB      00H             ;LD 01
DB      01H

DB      02H             ;AND 02
DB      02H

DB      07H             ;OR LD

DB      08H             ;OUT 21
DB      21H

DB      00H             ;LD 02
DB      02H

DB      04H             ;OR 03
DB      03H

DB      00H             ;LD 20
DB      20H

DB      04H             ;OR 21
DB      21H

DB      06H             ;AND LD

DB      08H             ;OUT 22
DB      22H

DB      00H             ;LD 04
DB      04H

DB      03H             ;AND_NOT
DB      22H

DB      08H             ;OUT 23
DB      23H

DB      01H             ;LD_NOT 23
DB      23H
```

```
DB      02H             ;AND 20
DB      20H

DB      08H             ;OUT 24
DB      24H

DB      00H             ;LD 20
DB      20H

DB      04H             ;OR 23
DB      23H

DB      00H             ;LD 22
DB      22H

DB      05H             ;OR_NOT 24
DB      24H

DB      06H             ;AND LD

DB      08H             ;OUT 25
DB      25H

DB      00H             ;LD 05
DB      05H

DB      03H             ;AND_NOT 25
DB      25H

DB      08H             ;OUT 26
DB      26H

DB      00H             ;LD 20
DB      20H

DB      02H             ;AND 24
DB      24H

DB      05H             ;OR_NOT 26
DB      26H

DB      08H             ;OUT 27
DB      27H

DB      10H             ;END

END
```

```
:1080000007583FC7582037498F075778A75760075B0
:10801000758C75741857783857682E0B4000F12B4
:10802000815B90829112813F128152028015B402CD
:10803000F12815B9081BF12813F12815202801525
:10804000B4040F12815B9081EB12813F12815202C6
:10805000815B4080F12815B90820212813F128159
:1080600052028015B4102F12812E908239858270B1
:1080700085836F857583857482E56FF0A3E570F065
:10808000A37402F0A3748CF0A37400F0A38583752D
:1080900085827402 8C00B4010F12815B90818D1275
:1080A000813F128152028015B4030F12815B9081CF
:1080B000D312813F128152028015B4050F12815BE9
:1080C00090830312813F128152028015B4090F126E
:1080D000815B90833A12813F128152028015B4066F
:1080E0000F12810512812E908329128113028015AF
:1080F000B4070F12810512812E908318128113028A
:10810000801502810285778385768 A385837785B2
:10811000827622858 6E85826D857583857482E57E
:1081200 6EF0A3E56DF0A385837585827422741 2C9
:1081300085758385748 F0A385837585827422851 
:10814000827885837385758385748 E573F0A3E5F2
:1081500078 2F0A385837585827422857583857462
:10816000827478F0A385837585827485778385769C
:1081700082A3E0A385837785827685758385748263
:10818000F0A37412F0A385837585827422D07FD00A
:1081900 7EE8C4540FB400672420F9E8540F90819E
:1081A000B79357700974FFC0E0C07EC07F2274008F
:1081B000C0E0C07EC07F220102040810204080D0B1
:1081C0007BD07AD079128291D0E05579C0E0C07A24
:1081D000C07B22D07BD07AD079128291D0E0F45546
```

```
:1081E00079C0E0C07AC07B220281AED07BD07AD049
:1081F00079128291D0E04579C0E0C07AC07B22023A
:1082000082BBD071D072D0E055FF6016E854F0B454
:10821000205BE8540F9081B793792247F7C072C072
:108220007122E854F0B42055E8540F9081B793F4CC
:10823000792257F7C072C071227583FC758201E5FF
:1082400022F022E8540F9081B79379245770637419
:1082500000C0E0C07EC07F22E8540F9081B79379C0
:10826000245770847 4FFC0E0C07EC07F22E8540FA2
:10827000 9081B793792447F7C072C07122E8540FF8
:10828000 9081B793F4792457F7C072C071220282AB
:1082900043D07FD07EE8C4540FB400442420F9E8D2
:1082A000540F9081B7935770097400C0E0C07EC02E
:1082B0007F2274FFC0E0C07EC07F22E854F0B4206B
:1082C00097E8540F9081B793792257B4000974FF4F
:1082D000C0E0C07EC07F227400C0E0C07EC07F22AC
:1082E000E854F0B420A8E8540F9081B7937922574E
:1082F000B40007C0E0C07EC07F2274FFC0E0C07E33
:10830000C07F22D07BD07AD079128291D0E0F44520
:1083100079C0E0C07AC07B22D037D036D069D0E0B7
:10832004569C0E0C036C03722D039D038D06AD0D5
:0D833000E0556AC0E0C038C03922028333A2F
:098C000090FC00E0F520028C114B
:108A000000000406082000030401020205050204 8
:108A1000082100070421082200200823012008243F
:018A2001045
:00000001FF
```

ภาคผนวก 4

CODE	คำสั่ง	ความหมาย
00	LD	นำค่าสภาวะจากตารางข้อมูลมาเป็นผลลัพธ์
01	LD NOT	นำค่าสภาวะที่ตรงข้ามจากตารางข้อมูลมาเป็นผลลัพธ์
02	AND	กระทำลอจิก AND กับสภาวะที่นำจากตารางข้อมูล
03	AND NOT	กระทำลอจิก AND กับสภาวะตรงข้ามที่นำมาจากตารางข้อมูล
04	OR	กระทำลอจิก OR กับสภาวะที่นำจากตารางข้อมูล
05	OR NOT	กระทำลอจิก OR กับสภาวะตรงข้ามที่นำมาจากตารางข้อมูล
06	AND LD	กระทำลอจิก AND กับสภาวะที่นำจาก STACK
07	OR LD	กระทำลอจิก OR กับสภาวะที่นำจาก STACK
08	OUT	นำผลลัพธ์ทางลอจิกไว้ในตารางข้อมูล
09	OUT NOT	นำผลลัพธ์ทางลอจิกแล้วทำตรงข้ามไว้ในตารางข้อมูล
10	END	สิ้นสุดคำสั่งให้เริ่มทำรอบใหม่