คำนำ

เอกสารประกอบการสอนเล่มนี้ ได้จัดทำขึ้นเพื่อใช้ในการสอนรายวิชา ***178 320 ไมโครโพรเซสเซอร์และการต่อประสาน (Microprocessors and Interfacing)*** สำหรับนักศึกษาระดับปริญญาตรี สาขาวิชาวิศวกรรมคอมพิวเตอร์

อย่างไรก็ตาม **นักศึกษาควรจะต้องศึกษาและค้นคว้าเพิ่มเติมด้วยตัวเอง** ทั้งจากหนังสืออ้างอิง (ที่กล่าวไว้ในเอกสารฉบับนี้) และหนังสือเล่มอื่นๆที่เกี่ยวข้อง

ในการผลิตเอกสารประกอบการสอนเล่มนี้ ผู้เขียนใคร่ขอแสดงความขอบคุณบุคลากรภาควิชาวิศวกรรมคอมพิวเตอร์ทุกท่านที่มีส่วนในการสนับสนุนและส่งเสริมจนเอกสารนี้สำเร็จลุล่วงด้วยดี โดยเฉพาะ ดร.วรินทร์ สุวรรณวิสูตร ภาควิชาวิศวกรรมคอมพิวเตอร์ มหาวิทยาลัยขอนแก่น สำหรับเอกสารอ้างอิงต่างๆสำหรับวิชานี้ที่ท่านได้สะสมมา พร้อมทั้ง ดร.สิริวิชญ์ เตชะเจษฎารังษี สำหรับข้อเสนอแนะต่างๆ คุณธเนศ อุไรเรืองพันธ์ สำหรับการจัดทำปก

ท้ายที่สุดนี้ ผู้เขียนหวังเป็นอย่างยิ่งว่า เอกสารเล่มนี้จะช่วยให้ผู้เรียนได้เข้าใจเนื้อหาได้เป็นอย่างดี เป็นพื้นฐานที่ดีและเป็นประโยชน์ในการศึกษาค้นคว้า เพื่อเพิ่มพูนเทคนิคในการออกแบบและพัฒนาระบบไมโครโพรเซสเซอร์ที่ดี

ดร.ดารณี หอมดี

พฤษภาคม 2548

สารบาญ

คำอธิบายรายวิชา

178 320 Microprocessors and Interfacing

ไมโครโพรเซสเซอร์และการต่อประสาน

Review of number systems and basic arithmetic operations used in processors. Processor architecture, instruction sets and addressing mode. Types of electronic memory and their circuit diagram. Input and output device interfacing, parallel and serial interfacing, synchronous and asynchronous interfacing, as well as D/A and A/D device interfacing.

ทบทวนระบบตัวเลขและการดำเนินงานเลขคณิตพื้นฐานในไมโครโพรเซสเซอร์ สถาปัตยกรรมของไมโครโพรเซสเซอร์และแบบจำลองการเขียนโปรแกรม เซตคำสั่งและแบบ วิธีการกำหนดเลขที่อยู่ ชนิดหน่วยความจำอิเล็กทรอนิกส์และวงจร การจัดเรื่องขัดจังหวะ การต่อประสานอุปกรณ์รับเข้าและส่งออก การต่อประสานแบบขนานและอนุกรม การต่อประสานอุปกรณ์แปลงสัญญาณแอนะล็อกกับสัญญาณดิจิตอล

บทที่ 1 บทนำ

1.1 ระบบคอมพิวเตอร์

ความหมายและความเป็นมา

เมื่อพิจารณาศัพท์คำว่า คอมพิวเตอร์ ถ้าแปลกันตรงตัวตามคำภาษาอังกฤษ จะหมายถึงเครื่องคำนวณ ดังนั้นถ้ากล่าวอย่างกว้าง ๆ เครื่องคำนวณที่มีส่วนประกอบเป็นเครื่องกลไกหรือเครื่องไฟฟ้า ต่างก็จัดเป็นคอมพิวเตอร์ได้ทั้งสิ้น ลูกคิดที่เคยใช้กันในร้านค้า ไม้บรรทัดคำนวณ (slide rule) ซึ่งถือเป็นเครื่องมือประจำตัววิศวกรในยุคยี่สิบปีก่อน หรือเครื่องคิดเลข ล้วนเป็นคอมพิวเตอร์ได้ทั้งหมด

ในปัจจุบันความหมายของคอมพิวเตอร์จะระบุเฉพาะเจาะจง หมายถึงเครื่องคำนวณอิเล็กทรอนิกส์ที่สามารถทำงานคำนวณผลและเปรียบเทียบค่าตามชุดคำสั่งด้วยความเร็วสูงอย่างต่อเนื่องและอัตโนมัติ

พจนานุกรมฉบับราชบัณฑิตยสถาน พ.ศ. 2525 ได้ให้คำจำกัดความของคอมพิวเตอร์ไว้ค่อนข้างกะทัดรัดว่า “เครื่องอิเล็กทรอนิกส์แบบอัตโนมัติ ทำหน้าที่เสมือนสมองกล ใช้สำหรับแก้ปัญหาต่าง ๆ ทั้งที่ง่ายและซับซ้อน โดยวิธีทางคณิตศาสตร์”

เครื่องคำนวณในยุคประวัติศาสตร์

เครื่องคำนวณเครื่องแรกของโลก ได้แก่ ลูกคิด มีการใช้ลูกคิดในหมู่ชาวจีน และใช้ในอียิปต์โบราณมากว่า 2500 ปี ลูกคิดของชาวจีนประกอบด้วยลูกปัดร้อยอยู่ในราวเป็นแถวตามแนวตั้ง โดยแต่ละแถวแบ่งเป็นครึ่งบนและล่าง ครึ่งบนมีลูกปัด 2 ลูก ครึ่งล่างมีลูกปัด 5 ลูก แต่ละแถวแทนหลักของตัวเลข

เครื่องคำนวณกลไกที่รู้จักกันดี ได้แก่ เครื่องคำนวณของปาสคาลเป็นเครื่องที่บวกลบด้วยกลไกเฟืองที่ขบต่อกัน Blaise Pascal นักคณิตศาสตร์ชาวฝรั่งเศส ได้ประดิษฐ์ขึ้นในปี พ.ศ. 2185

สรุปเหตุการณ์ที่สำคัญเกี่ยวกับคอมพิวเตอร์

ปี ค.ศ.	เหตุการณ์ที่สำคัญ	หมายเหตุ
2600 BC	ลูกคิด (Abacus) เป็นเครื่องมือประมวลผลข้อมูลที่ใช้งานมานานหลายพันปี	ยุคแอนะล็อก
1500	Leonardo da Vinci คิด แต่ไม่ได้สร้าง	
1642	Blaise Pascal สร้างเครื่องบวกเลข ที่เรียกว่า (Pascalline) ทำงานเหมือนเครื่องนับรอบ	ยุคระบบจักรกล
1694	Gottfried Wilhelm Leibniz สร้าง Leibniz calculator ซึ่งถือว่าเป็นเครื่องคิดเลขระบบจักรกล (mechanical calculator) เครื่องแรกของโลกที่สามารถ +, –, *, / และ $\sqrt[2]{}$ ได้	
1801	Joseph Marie Jacquard ได้ประดิษฐ์ Pasteboard card Jacquard-loom ที่มีการควบคุมลวดลายการทอผ้าจากการอ่านตำแหน่งเส้นด้ายบนบัตรโลหะ	ยุคของการมีหน่วยความจำ

	เจาะรู ถือว่าเป็นต้นแบบของคอมพิวเตอร์แบบอัตโนมัติในยุคต่อมา	
1821	Charles Babbage สร้าง difference engine ที่สามารถคำนวณ polynomial ที่ซับซ้อน และ differential equation ได้, ตีพิมพ์บทความทางวิทยาศาสตร์เกี่ยวกับแนวความคิดที่ใกล้เคียงกับคอมพิวเตอร์ในสมัยปัจจุบัน	
1864	George Boole ได้สร้างระบบพีชคณิตแบบใหม่ เรียกว่า Boolean Algebra ซึ่งต่อมาได้กลายเป็นพื้นฐานทางตรรกศาสตร์ที่ใช้หาข้อเท็จจริงและเหตุผลที่เป็นเงื่อนไข ซึ่งส่งผลถึงแนวคิดเกี่ยวกับเลขฐานสอง (Binary Number) ซึ่งเป็นสัญลักษณ์แทนสถานะทางไฟฟ้าอันเป็นพื้นฐานของระบบดิจิตอลคอมพิวเตอร์ด้วย	
1879	Herman Hollerith ได้สร้างเครื่องนับประชามติ (Electrical Census Counting Machine) โดยใช้เครื่องตรวจจับทางไฟฟ้าเพื่ออ่านข้อมูลจากบัตร ใช้ในการเก็บข้อมูลสำมะโนครัวประชากรของชาวอเมริกาเป็นครั้งแรก และเป็นผู้ก่อตั้งบริษัท IBM	ยุคอิเล็กทรอนิกส์และดิจิตอล
1939	John Atanasoff และ Chifford Berry ได้ทดลองสร้าง electronic digital computer ขึ้นเป็นเครื่องแรก เรียกว่า Atanasoff-Berry Computer: ABC	
1944	John W. Mauchly, J. Presper Eckert Jr. และ Dr. John Von Neumann จากมหาวิทยาลัยฮาร์วาร์ด ได้นำแนวคิดของ Babbage มาประยุกต์สร้างคอมพิวเตอร์ที่ทำงานโดยใช้ระบบไฟฟ้าร่วมกับระบบจักรกล (Electro-mechanical) ที่มีชื่อว่า Mark I สามารถคูณตัวเลขขนาด 10 หลักได้ภายในเวลาไม่กี่วินาที	
1945	John von Neumann ได้เขียนบทความเกี่ยวกับ binary coding, การคำนวณแบบเรียงลำดับ, โปรแกรมที่เก็บไว้ในหน่วยความจำ ซึ่งยังคงเป็นรากฐานของคอมพิวเตอร์ในปัจจุบัน	
1946	Mauchly และ Eckert จากมหาวิทยาลัยเพนซิลวาเนีย ได้สร้างคอมพิวเตอร์ที่ใช้ระบบไฟฟ้าล้วน ที่ชื่อว่า ENIAC (Electronics Numerical Integrator And Calculator) ซึ่งใช้ vacuum tubes	ดิจิตอลคอมพิวเตอร์เครื่องแรกของโลก
1947	Transistor ถูกค้นพบโดยทีมนักวิทยาศาสตร์ประจำ BELL Laboratory ซึ่งประกอบด้วย John Bardeen, Walter Brattain, และ William Schockley	
1950	บริษัท Eckert-Mauchly Computer Corporation ได้ผลิตและจำหน่ายคอมพิวเตอร์ชื่อ UNIVAC (UNIVersal Automatic Computer) เป็นครั้งแรก	
1955	IBM วางตลาดคอมพิวเตอร์ที่ใช้ transistor	ยุค transistors

1957	Robert Noyce กับเพื่อนอีก 7 คน ลาออกจาก Schockley Lab มาอยู่กับ Fairchild	
1959	Jack Kilby ของ TI จดลิขสิทธิ์ IC (Integrated Circuit) ต่อมาอีก 5 เดือน Robert Noyce ก็จดด้วย (ฟ้องร้องกัน วิธีของ Noyce ดีกว่า)	
1968	Noyce ตั้ง Intel	
1969	Marcian Ted Hoff สร้างไมโครโพรเซสเซอร์ (μP)	
1971	μP 4004 เริ่มวางตลาด	
1972	μP 8008 เริ่มวางตลาด	
1974	8080, บริษัท RCA และ Motorola ออกมาร่วมด้วย	

1976	μC (=μP + Mem + IO) Intel 8048 NS COP400	μP Zilog Z80 Intel 8085 Texas TMS9900
1978	Intel 8051 Motorola MC6801 Zilog Z8	Intel 8086 Zilog Z8000
1980		Motorola 68000

ยุคของคอมพิวเตอร์

การพัฒนาทั้งขนาดและประสิทธิภาพในการประมวลผลของดิจิตอลคอมพิวเตอร์ สามารถสรุปได้เป็น 5 ยุค ดังนี้

- คอมพิวเตอร์ยุคที่ 1 (First Generation Computer: พ.ศ. 2497-2501): ยุคของหลอดสุญญากาศ

คอมพิวเตอร์ในยุคนี้ใช้หลอดสุญญากาศ (Vacuum tube) เป็นวงจรอิเล็กทรอนิกส์ เครื่องยังมีขนาดใหญ่มาก ใช้กระแสไฟฟ้าจำนวนมาก ทำให้เครื่องมีความร้อนสูงจึงมักเกิดข้อผิดพลาดง่าย คอมพิวเตอร์ในยุคนี้ได้แก่ UNIVAC I, IBM 600

- คอมพิวเตอร์ยุคที่ 2 (Second Generation Computer: พ.ศ. 2502-2507): ยุคของ Transistors

คอมพิวเตอร์ยุคนี้ใช้ทรานซิสเตอร์ (Transistor) เป็นวงจรอิเล็กทรอนิกส์ และใช้วงแหวนแม่เหล็กเป็นหน่วยความจำ คอมพิวเตอร์มีขนาดเล็กกว่ายุคแรก ต้นทุนต่ำกว่า ใช้กระแสไฟฟ้าและมีความแม่นยำมากกว่า

- คอมพิวเตอร์ยุคที่ 3 (Third Generation Computer: พ.ศ. 2508-2513): ยุคของ IC

คอมพิวเตอร์ยุคนี้ใช้วงจรไอซี (IC: Integrated Circuit) เป็นสารกึ่งตัวนำที่สามารถบรรจุวงจรทางตรรกะไว้แล้วพิมพ์บนแผ่นซิลิกอน(Silicon) เรียกว่า "ชิป"

- คอมพิวเตอร์ยุคที่ 4 (Fourth Generation Computer: พ.ศ. 2514-2523): ยุคของ LSI

คอมพิวเตอร์ยุคนี้ใช้วงจร LSI (Large-Scale Integrated Circuit) เป็นการรวมวงจรไอซีจำนวนมากลงในแผ่นซิลิกอนชิป 1 แผ่น สามารถบรรจุได้มากกว่า 1 ล้านวงจร ด้วยเทคโนโลยีใหม่นี้ทำให้เกิดแนวคิดในการบรรจุวงจรที่สำคัญสำหรับการทำงานพื้นฐานของคอมพิวเตอร์นั่นคือ CPU ลงชิปตัวเดียว เรียกว่า "ไมโครโพรเซสเซอร์"

- <u>คอมพิวเตอร์ยุคที่ 5</u> (Fifth Generation Computer: พ.ศ. 2524-ปัจจุบัน): ยุคของ VLSI และ portable/pocket computers

คอมพิวเตอร์ยุคนี้ใช้วงจร VLSI (Very Large-Scale Integrated Circuit) เป็นการพัฒนาไมโครโพรเซสเซอร์ให้มีประสิทธิภาพมากขึ้น

ประเภทของคอมพิวเตอร์

การจัดแบ่งประเภทของ เครื่องคอมพิวเตอร์ จะอาศัยคุณสมบัติต่างๆ เช่น ความเร็วของการประมวลผล และขนาดความจำ ของหน่วยบันทึกข้อมูล ซึ่งสามารถแบ่งได้ เป็น 4 ประเภท ได้แก่

- Super Computer
- Mainframe Computer
- Mini Computer
- Micro Computer

ที่มา: NECTEC's Web Based Learing

ไมโครโพรเซสเซอร์

Computer = CPU (ALU + CU) + Memory + I/O

Microcomputer = MPU + Memory + I/O

MPU = μP = CPU ที่มีลักษณะเป็น single chip (IC เพียงตัวเดียว) ⟶ วงจรรวม (IC) ที่ทำหน้าที่เป็นหน่วยประมวลผลกลางที่สามารถโปรแกรมได้

โดยที่ μP จะถูกออกแบบขึ้นมาเฉพาะตามลักษณะของงานที่ต้องการให้ μP ทำงาน ซึ่ง μP สามารถที่จะทำงานตามคำสั่งต่างๆ ที่ได้รับการโปรแกรมเข้ามา

ก่อกำเนิดไมโครโพรเซสเซอร์

เมื่อก่อนนั้น Intel เป็นบริษัทผลิตชิปไอซี แห่งหนึ่งที่ไม่ใหญ่โตมากนักเท่าในปัจจุบัน เมื่อปี ค.ศ. 1969 ได้สร้างความสะเทือน ให้กับวงการอิเล็คทรอนิคส์ โดยการออกชิปหน่วยความจำ (Memory) ขนาด 1 Kbyte มาเป็นรายแรก

บริษัทบิสซิคอมพ์ (Busicomp) ซึ่งเป็นผู้ผลิตเครื่องคิดเลขของญี่ปุ่นได้ทำการว่าจ้างให้ Intel ทำการผลิตชิปไอซี ที่บิสซิคอมพ์เป็นคนออกแบบเองที่มีจำนวน 12 ตัว โครงการนี้ถูกมอบหมายให้นาย M.E. Hoff,

Jr. ซึ่งเข้าตัดสินใจที่จะใช้วิธีการออกแบบชิปแบบใหม่ โดยสร้างชิปที่ให้ถูกโปรแกรมได้ หมายถึงว่าสามารถนำเอาชุดคำสั่งของการคำนวณไปเก็บไว้ใน หน่วยความจำก่อนแล้วให้ไอซีตัวนี้อ่านเข้ามาแปลความหมาย และทำงานภายหลัง

ในปี 1971 Intel ได้นำผลิตภัณฑ์ออกสู่ตลาด โดยใช้ชื่อทางการค้าว่า Intel 4004 ในราคา 200 เหรียญสหรัฐ และเรียกชิปนี้ว่าเป็น ไมโครโพรเซสเซอร์(Microprocessor) ก็เพราะว่า 4004 นี้เป็น CPU (Central Processing Unit) ตัวหนึ่ง ซึ่งมีขนาด 4.2 X 3.2 มิลลิเมตร ภายในประกอบด้วยทรานซิสเตอร์จำนวน 2,250 ตัว และเป็นไมโครโพรเซสเซอร์ขนาด 4 บิต

หลังจาก 1 ปีต่อมา Intel ได้ออก ไมโครโพรเซสเซอร์ ขนาด 8 บิตออกมาโดยใช้ชื่อว่า 8008 มีชุดคำสั่ง 48 คำสั่ง และอ้างหน่วยความจำได้ 16 Kbyte ซึ่งทาง Intel หวังว่าจะเป็นตัวกระตุ้นตลาดทางด้านชิปหน่วยความจำได้อีกทางหนึ่ง

เมื่อปี 1973 ทาง Intel ได้ออก ไมโครโพรเซสเซอร์ 8080 ที่มีชุดคำสั่งพื้นฐาน 74 คำสั่งและสามารถอ้างหน่วยความจำได้ 64 Kbyte

ไมโครคอมพิวเตอร์ เครื่องแรกของโลก

เมื่อปี 1975 มีนิตยสารต่างประเทศฉบับหนึ่ง ชื่อว่า Popular Electronics ฉบับเดือนมกราคมได้ลงบทความเกี่ยวกับเครื่องไมโครโพรเซสเซอร์เครื่องแรกของโลกที่มีชื่อว่า อัลแตร์ 8800 (Altair) ซึ่งทำออกมาเป็นชุดคิท โดยบริษัท MITS (Micro Insumentation And Telemetry Systems)

ที่มา: NECTEC's Web BasedLlearing

บริษัท MITS ถูกก่อตั้งเมื่อปี 1969 โดยมีจุดมุ่งหมายเพื่อทำตลาดในด้านเครื่องคิดเลข แต่การค้าชะลอตัวลง ประธานบริษัท ชื่อ H. Edword Roberts เห็นการไกลคิดเปิดตลาดใหม่ซึ่งจะขายชุดคิดคอมพิวเตอร์ ประมาณเอาไว้ว่าอาจขายได้ในจำนวนปีล่ะประมาณ 200-300 ชุด จึงให้ทีมงานออกแบบและพัฒนาแล้วเสร็จก่อนถึงคริสต์มาส ในปี 1974 แต่เพิ่งมา ประกาศตัวในปีถัดไป สำหรับ CPU ที่ใช้คือ 8080 และคำว่า ไมโครคอมพิวเตอร์ จึงถูกเรียกใช้เป็นครั้งแรกเพื่อชุดคิทคอมพิวเตอร์ชุดนี้

ชุดคิทของอัลแตร์นี้ประกอบด้วยไมโครโพรเซสเซอร์ 8080 ของบริษัท Intel มี เพาเวอร์ซัพพลาย มีแผงหน้าปัดที่ติดหลอดไฟ เป็นแถวมาให้เพื่อแสดงผล รวมถึงหน่วยความจำ 256 Bytes นอกนั้นยังมี Slot ให้เสียบอุปกรณ์อื่น ๆ เพิ่มได้ แต่ก็ทำให้

MITS ต้องผิดคาด คือภายในเดือนเดียวมีจดหมายส่งเข้ามาขอสั่งซื้อเป็นจำนวนถึง 4,000 ชุดเลยทีเดียวด้วยชิป 8080 นี่เองได้เป็นแรงดลใจให้บริษัท Digital Research กำเนิดระบบปฏิบัติการ (Operating System) ที่ชื่อว่า CP/M หรือ Control Program for Microcomputer ขึ้นมา ในขณะที่ Microsoft ยังเพิ่งออก Microsoft Basic รุ่นแรก

ยุค Z80

เมื่อเดือนพฤศจิกายนปี 1974 ได้มี วิศวกรของ Intel บางคนได้ออกมาตั้งบริษัทผลิตชิปเอง โดยมีชื่อว่า Zilog เนื่องจาก วิศวกรเหล่านี้ ได้มีส่วนร่วมในการผลิตชิป 8080 ด้วยจึงได้นำเอาเทคโนโลยีการผลิตนี้มาสร้างตัวใหม่ที่ดีกว่า มีชื่อว่า Z80 ยังคงเป็น ชิปขนาด 8 บิต เมื่อได้ออกสู่ตลาดได้รับความนิยมเป็นอย่างมาก เนื่องจากได้ปรับปรุงข้อบกพร่องต่าง ๆ ที่มีอยู่ใน 8080 จึงทำให้เครื่องคอมพิวเตอร์ หลายต่อหลายยี่ห้อ หันมาใช้ชิป Z80 กัน แม้แต่ CP/M ก็ยังถูกปรับปรุงให้มาใช้กับ Z80 นี้ด้วย

1.2 ภาษาคอมพิวเตอร์

มนุษย์ทุกวันนี้สื่อสารกันด้วยภาษาที่แตกต่างกัน ขึ้นอยู่กับความเป็นมาและสภาพแวดล้อใของแต่ละชาติ ภาษาที่นิยมใช้สื่อสารกันในปัจจุบันนี้ เช่น ภาษาอังกฤษ ภาษาฝรั่งเศส ภาษาญี่ปุ่น ภาษาจีน และอื่น ๆ

การสื่อสารของคอมพิวเตอร์ก็มีลักษณะเช่นเดียวกับการสื่อสารของมนุษย์ กล่าวคือมีภาษาที่แตกต่างกันจำนวนมากที่สามารถสั่งให้คอมพิวเตอร์ ทำงานได้ เช่น ภาษาเบสิก (BASIC) ภาษาปาสคาล (PASCAL) ภาษาโคบอล (COBOL) ถึงแม้ว่าคอมพิวเตอร์จะรับคำสั่งจากภาษาคอมพิวเตอร์ที่มีโครงสร้างแตกต่างกันได้แต่เนื่องจากคอมพิวเตอร์เป็นอุปกรณ์อิเล็กทรอนิกส์ ดังนั้นการรับคำสั่งในการทำงานคอมพิวเตอร์จะรับเป็นสัญญาณไฟฟ้าที่เรียกว่า "ภาษาเครื่อง" หรือ 1 machine language นั่นเอง

ประเภทของภาษาคอมพิวเตอร์

ภาษาที่ใช้ในการโปรแกรมคอมพิวเตอร์มีจำนวนหลายร้อยภาษา แต่ละภาษาก็มีคุณลักษณะโครงสร้างและกฎเกณฑ์ของตนเอง บางภาษาถูกพัฒนาขึ้นให้ทำงานกับเครื่องคอมพิวเตอร์บางประเภท ในขณะที่บางภาษาถูกพัฒนาให้สามารถทำงานเฉพาะอย่าง เช่น ทางวิทยาศาสตร์หรือธุรกิจ

1. <u>ภาษาเครื่อง</u> (Machine Language)

ภาษาเครื่องเป็นภาษาคอมพิวเตอร์เพียงภาษาเดียวที่สามารถสื่อสารได้กับคอมพิวเตอร์โดยตรง ซึ่งในบางครั้งก็เรียกกันว่าภาษาในยุคที่หนึ่ง

คำสั่งในภาษาเครื่องจะเป็นชุดคำสั่งที่ประกอบด้วยตัวเลขของเลขฐานสอง (binary digits หรือ bits) ที่ใช้เลข 0 และ 1 เป็นสัญลักษณ์แทนสัญญาณไฟฟ้าปิดและเปิดตามลำดับ และเนื่องจากคอมพิวเตอร์สามารถสื่อสารเข้าใจกับภาษาเครื่องได้โดยตรง ดังนั้นโปรแกรมภาษาเครื่องจึงไม่จำเป็นต้องมีตัวแปลภาษา ข้อเสียของโปรแกรมที่เขียนด้วยภาษาเครื่องคือ โปรแกรมจะสามารถทำงานเฉพาะกับเครื่องคอมพิวเตอร์ที่ใช้

พัฒนาโปรแกรมนั้นเท่านั้น (machine - dependent) และข้อเสียอีกประการหนึ่งก็คือผู้เขียนโปรแกรมจะรู้สึกยุ่งยากและเบื่อหน่าย ตลอดจนต้องใช้เวลามากในการพัฒนาโปรแกรมด้วยภาษาเครื่อง

2. ภาษาระดับต่ำ (Low-level Language): Assembly

เนื่องมาจากภาษาเครื่องยากแก่การเขียนโปรแกรม ภาษาในยุคที่สอง (second-generation language) ที่เรียกว่า ภาษาแอสแซมบลี จึงได้ถูกพัฒนาขึ้น ภาษาแอสแซมบลีหรือภาษาสัญลักษณ์ (symbolic programming language) จะใช้รหัสและสัญลักษณ์แทน 0 และ 1 ในการเขียนโปรแกรมภาษาแอสแซมบลี มีข้อดีหลายประการเมื่อเทียบกับภาษาที่เขียนด้วยภาษาเครื่อง ตัวอย่างเช่น ภาษาแอสแซมบลีจะใช้ตัวย่อที่มีความหมาย หรือในบางครั้งเรียกว่านีมอนิก (mnemonics) ในคำสั่งเพื่อเขียนโปแกรม เช่น ใช้ A แทนการบวก ใช้ C แทนการเปรียบเทียบ ใช้ M แทนการคูณ เป็นต้น ข้อดีอีกประการหนึ่งก็คือ ภาษาแอสแซมบลีสามารถเรียนรู้ได้ง่ายและเร็วกว่าการเขียนโปรแกรมด้วยภาษาเครื่อง (เลข 0 และ 1) นอกจากนี้ผู้เขียนโปรแกรมด้วยภาษาแอสแซมบลียังสามารถกำหนดชื่อที่เก็บข้อมูลในหน่วยความจำเป็นค่าในภาษาอังกฤษ (แทนที่จะเป็นเลขที่ตำแหน่งในหน่วยความจำ) ตัวอย่างเช่น โปรแกรมเมอร์สามารถใช้ PRICE แทนตำแหน่งที่อยู่ของ unit price ซึ่งเดิมจะเก็บเป็นตัวเลข เช่น 11001011 เป็นต้น

โปรแกรมที่เขียนด้วยภาษาแอสแซมบลี จำเป็นจะต้องทำการแปลด้วยโปรแกรมแปลภาษาที่เรียกว่าแอสแซมบลี (assembler) โดยแอสแซมเบลอจะทำหน้าที่แปลโปรแกรมต้นฉบับ (source program) ที่เขียนด้วยภาษาแอสแซมบลีให้ เป็นภาษาเครื่องซึ่งคอมพิวเตอร์สามารถเข้าใจได้

ข้อดี: เหมาะสำหรับงานที่ต้องการความแน่นอน และความละเอียด
ประหยัดเนื้อที่ของหน่วยความจำ

3. ภาษาระดับสูง (High-level Language)

เนื่องมาจากภาษาเครื่องและภาษาแอสแซมบลี ยังมีข้อจำกัดในการนำไปพัฒนาโปรแกรม ดังนั้นจึงได้มีการพัฒนาภาษาระดับสูง (high-level languages) ขึ้น ภาษาระดับสูงเป็นภาษาที่ง่ายต่อการเรียนรู้และการนำไปประยุกต์ใช้งาน นอกจากนี้โปรแกรมภาษาที่เขียนด้วยภาษาระดับสูงยังสามารถทำงานบนเครื่องคอมพิวเตอร์ต่างชนิดกันได้ด้วย (ไม่ผูกติดกับฮาร์ดแวร์) ภาษาในยุคที่สาม(third- generation languages หรือ 3GL) หรือภาษาโพรซีเยอร์(procedural languages) เป็นชุดคำสั่งที่มีลักษณะเหมือนคำในภาษาอังกฤษ เช่น ใช้คำสั่ง add เพื่อสั่งให้คอมพิวเตอร์บวก และใช้คำสั่ง print เพื่อสั่งให้พิมพ์ นอกจากนี้คำสั่งในภาษายุคที่สามยังใช้สัญลักษณ์ทางคณิตศาสตร์ เช่น * แทนการคูณ + แทนการบวก เป็นต้น

เช่นเดียวกับภาษาแอสแซมบลี โปรแกรมภาษาในยุคที่สามที่ถูกเขียนขึ้นหรือที่เรียกว่า โปรแกรมต้นฉบับ (source code) จึงจำเป็นจะต้องมีตัวแปลภาษาเพื่อให้เป็นภาษาเครื่องซึ่งเป็นภาษาที่คอมพิวเตอร์เข้าใจได้ โดยโปรแกรมแปลภาษาระดับสูงจะจำแนกเป็น 2 ประเภทคือ คอมไพเลอร์ (compiler) และอินเตอร์พรีเตอร์ (interpreter) ตัวแปลภาษาระดับสูงขึ้นอยู่กับภาษาที่ใช้ในการเขียนโปรแกรม

ตัวแปลภาษา (Language translator) แบ่งออกเป็น 3 ประเภท

- Assembler เป็นตัวแปลภาษาแอสแซมบลีให้เป็นภาษาเครื่อง

- Compiler คือ ซอฟต์แวร์ที่ใช้ในการแปลภาษาระดับสูงให้เป็นภาษาเครื่อง (Machine language) คำสั่งที่ใช้เขียนในโปรแกรมภาษาระดับสูงเราเรียกว่า Source code คอมไพเลอร์จะทำการแปล Source code ให้เป็นภาษาเครื่อง เราเรียกว่า Object code เพื่อทำการจัดเก็บไว้เพื่อนำมาใช้งาน (Run) ภายหลัง
- Interpreter คือ ซอฟต์แวร์ที่ทำการเปลี่ยนภาษาระดับสูงให้เป็นภาษาเครื่องในแต่ละครั้งเมื่อต้องการทำงาน แตกต่างจากคอมไพเลอร์ที่ทำการแปลครั้งเดียวสามารถทำงานได้ตลอด

1.3 ระบบเลขฐานที่ใช้ในคอมพิวเตอร์

เป็นที่ทราบกันดีว่าคอมพิวเตอร์ทำงานด้วยกระแสไฟฟ้า ดังนั้นจึงมีการแทนที่สภาวะของกระแสไฟฟ้าได้ 2 สภาวะ คือ สภาวะที่มีกระแสไฟฟ้า และสภาวะที่ไม่มีกระแสไฟฟ้า และเพื่อให้โปรแกรมเมอร์สามารถสั่งการคอมพิวเตอร์ได้ จึงได้มีการสร้างระบบตัวเลขที่นำมาแทนสภาวะของกระแสไฟฟ้า โดยตัวเลข 0 จะแทนสภาวะไม่มีกระแสไฟฟ้า และเลข 1 แทนสภาวะมีกระแสไฟฟ้า

ระบบตัวเลขที่มีจำนวน 2 จำนวน (2 ค่า) เรียกว่าระบบเลขฐานสอง (Binary Number System) ซึ่งเป็นระบบตัวเลขที่สามารถนำมาใช้ในการสั่งงานคอมพิวเตอร์ โดยการแทนที่สภาวะต่างๆ ของกระแสไฟฟ้า แต่ในชีวิตประจำวันของคนเราจะคุ้นเคยกับตัวเลขที่มีจำนวน 10 จำนวน คือ เลข 0 - 9 ซึ่งเรียกว่าระบบเลขฐานสิบ (Decimal Number System) ดังนั้นจึงมีความจำเป็นต้องศึกษาระบบเลขฐาน ประกอบการการศึกษาวิชาด้านคอมพิวเตอร์

ระบบจำนวนที่ใช้ในทางคอมพิวเตอร์ ประกอบด้วย

- ระบบเลขฐานสอง (Binary Number System) ประกอบด้วยตัวเลข 0 และ 1
- ระบบเลขฐานแปด (Octal Number System) ประกอบด้วยตัวเลข 0 - 7
- ระบบเลขฐานสิบ (Decimal Number System) ประกอบด้วยตัวเลข 0 - 9
- ระบบเลขฐานสิบหก (Hexadecimal Number System) ประกอบด้วยตัว เลข 0 - 9 และ A - F

ระบบเลขฐานสิบ (Decimal Number System)

ระบบเลขฐานสิบ เป็นระบบเลขที่ใช้กันในชีวิตประจำวัน ไม่ว่าจะนำไปใช้คำนวณประเภทใด โดยจะมีสัญลักษณ์ที่ใช้แทนตัวเลขต่างๆ ของเลขฐานสิบ (Symbol) จำนวน 10 ตัว ตัวเลขหรือที่เรียกว่า Digit ที่ใช้แทนระบบเลขฐานสิบ ได้แก่ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

ตัวเลขแต่ละตัวจะมีค่าประจำตัว โดยกำหนดให้ค่าที่น้อยที่สุด คือ 0 (ศูนย์) และเพิ่มค่าทีละหนึ่ง จนครบจำนวน 10 ตัว ดังนั้นค่ามากที่สุด คือ 9 การนำตัวเลขเหล่านี้ มารวมกลุ่มกัน ทำให้เกิดความหมายเป็น "ค่า" นั้น อาศัยวิธีการกำหนด "หลัก" ของตัวเลข (Position Notation) กล่าวคือ ค่าของตัวเลขจำนวนหนึ่ง พิจารณาได้จากสองสิ่งคือ

- ค่าประจำตัวของตัวเลขแต่ละตัว
- ค่าหลักในตำแหน่งที่ตัวเลขนั้นปรากฏอยู่

ในระบบที่ว่าด้วยตำแหน่งของตัวเลข ตำแหน่งที่อยู่ทางขวาสุด จะเป็นหลักที่มีค่าน้อยที่สุด เรียกว่า Least Significant Digit (LSD) และตัวเลขที่อยู่ในหลักซ้ายสุดจะมีค่ามากที่สุด เรียกว่า Most Significant Digit (MSD)

<u>นิยาม</u> ค่าหลักของตัวเลขใดๆ คือ ค่าของฐานยกกำลังด้วยค่าประจำตำแหน่ง ของแต่ละหลัก โดยกำหนดให้ค่าประจำตำแหน่งของหลักของ LSD มีค่าเป็น 0

ในระบบเลขฐานสิบ จะมีสัญลักษณ์อยู่ 10 อย่าง คือ 0 - 9 จำนวนขนาดของเลขฐานสิบ สามารถอธิบายได้ โดยใช้ตำแหน่งน้ำหนักของแต่ละหลัก (Positional Weight) โดยพิจารณาจากเลข ดังต่อไปนี้

3472 สามารถขยายได้ดังนี้

3472	$= 3000 + 400 + 70 + 2$
	$= (3 \times 10^3) + (4 \times 10^2) + (7 \times 10^1) + (2 \times 10^0)$

จะเห็นว่าน้ำหนักตามตำแหน่ง ของตัวเลขต่างๆ สามารถขยายตามระบบจำนวนได้ และถูกแทนที่ด้วยสมการ ดังต่อไปนี้

$$N = d_nR^n + ... + d_3R^3 + d_2R^2 + D_1R^1 + D_0R^0$$

เมื่อ

N	คือ ค่าของจำนวนฐานสิบที่ต้องการ
d_n	คือ ตัวเลขที่อยู่ในตำแหน่งต่างๆ
R	คือ ฐานของจำนวนตัวเลขนั้นๆ
n	คือ ค่ายกกำลังของฐานตามตำแหน่งต่างๆ

เลขที่เป็นเศษส่วน หรือจำนวนผสมนั้น ก็สามารถจะเขียนในรูป Positional Notation ได้เช่นกัน โดยตัวเลขแต่ละหลัก จะอยู่ในตำแหน่งหลังจุดทศนิยม กำลังของหลัก จะมีค่าเป็นลบ เริ่มจากลบ 1 เป็นต้นไป นับจากน้อยไปหามาก ดังนั้นในระบบเลขฐานสิบ หลักแรกหลังจุดทศนิยม จะมีค่าเท่ากับ เลขจำนวนนั้นคูณด้วย 10-1 ตัวที่สองจะเป็น 10-2 ไปเรื่อยๆ

$456.395 =$	4×10^2	+	5×10^1	+	6×10^0	+	3×10^{-1}	+	9×10^{-2}	+	5×10^{-3}

กฎการแทนตัวเลขนั้น สามารถนำไปใช้กับระบบตัวเลขทั่วๆ ไปได้ โดยไม่คำนึงว่า เลขนั้นจะเป็นฐานอะไร

ระบบเลขฐานสอง

ระบบเลขฐานสอง มีสัญลักษณ์ที่ใช้เพียงสองตัว คือ 0 และ 1 ถ้าเปรียบเทียบเลขฐานสอง กับเลขฐานสิบแล้ว ค่าของหลักที่ถัดจากหลักที่น้อยที่สุด (LSD) ขึ้นไป จะมีค่าเท่ากับ ฐานสองยกกำลังหมายเลขหลักแทนที่จะเป็น 10 ยกกำลัง ดังนี้

เลขฐานสิบ				เลขฐานสอง		
10^0	= 1	หน่วย		2^0	= 1	หนึ่ง

10^1	= 10	สิบ		2^1	= 2	สอง
10^2	= 100	ร้อย		2^2	= 4	สี่
10^3	= 1000	พัน		2^3	= 8	แปด

ระบบเลขฐานสองเกิดจากการใช้ตัวเลขเพียง 2 ตัว คือ 0 และ 1 ดังนั้น สมการคือ

$$N = ... + (d_3 \times 2^3) + (d_2 \times 2^2) + (d_1 \times 2^1) + (d_{0x} 2^0)$$

เมื่อ d คือค่า 0 หรือ 1 เช่น $1101 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$

เพื่อตัดปัญหายุ่งยาก ในการแทนค่าของเลขระบบต่างๆ เรานิยมเขียน ตัวเลขอยู่ในวงเล็บ และเขียนค่าของฐานนั้น อยู่นอกวงเล็บ เช่น $(101101)_2 = (45)_{10}$

สำหรับเศษส่วน จะเขียนค่าของเศษส่วนอยู่หลังจุด (Binary Point) ยกกำลังเป็นลบ เพิ่มขึ้นตามลำดับ ดังตัวอย่าง $(0.1011)_2 = (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) + (1 \times 2^{-4})$

การแปลงเลขฐานสองเป็นเลขฐานสิบ

การแปลงเลขฐานสองเป็นเลขฐานสิบ มีหลายวิธี แต่ที่จะแนะนำคือ การกระจายค่าประจำหลัก จากนั้นนำมาบวกรวมกันอีกครั้ง ผลลัพธ์ที่ได้จะเท่ากับค่าในเลขฐานสิบ

ตัวอย่าง 10111 มีค่าเท่ากับเท่าไรในระบบเลขฐานสิบ

วิธีทำ

$$(10111)_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

$$= 16 + 0 + 4 + 2 + 1$$

$$= 23$$

การแปลงเลขฐานสิบเป็นฐานสอง

การแปลงเลขฐานสิบเป็นเลขฐานสองก็มีหลายวิธี แต่ที่จะแนะนำคือ การหารด้วย 2 แล้วจดค่าเศษจากการหารไว้ จนกระทั่งหารไม่ได้อีกแล้ว จากนั้นนำเศษ จากการหารแต่ละครั้ง มาไล่ลำดับจากล่างขึ้นไปหาค่าบนสุด ผลลัพธ์ที่ได้จะเท่ากับค่าในเลขฐานสอง

ตัวอย่าง 26_{10} มีค่าเท่ากับเท่าไรในระบบเลขฐานสอง

วิธีทำ หาได้จากวิธีหารสั้นดังนี้

2) 26
2) 13 เศษ 0
2) 6 เศษ 1
2) 3 เศษ 0
2) 1 เศษ 1
0 เศษ 1

เมื่อหารไม่ได้ ให้นำค่าเศษมาเรียงต่อกัน โดยเรียงจากค่าล่างสุด ไปหาค่าบนสุด เพราะฉะนั้นจะได้ค่าเท่ากับ 11010

ดังนั้น 26 (ในฐานสิบ) จึงมีค่าเท่ากับ 11010_2

การแปลงเลขเศษส่วนฐานสอง (Fractional Binary Numbers) ให้เป็นฐานสิบ

เนื้อหาที่กล่าวไปแล้ว ได้กล่าวถึงระบบเลขฐาน และการแปลงเลขฐานสองเป็นเลขฐานสิบ การแปลงเลขฐานสิบเป็นเลขฐานสอง ในส่วนของเลขจำนวนเต็ม ในส่วนนี้จะแนะนำการแปลงเลขฐานที่เป็นเลขเศษส่วน

<u>สมการการแปลงเลขเศษส่วนฐานสองเป็นฐานสิบ คือ</u>

$$N = d_1R^{-1} + d_2R^{-2} + d_3R^{-3} + ... + d_nR^{-n}$$

<u>ตัวอย่าง</u> ต้องการแปลงเลขเศษส่วนฐานสอง 0.1011 เป็นเลขฐานสิบ

พิจารณาทีละจุด

- ตำแหน่งแรกของจำนวนที่ระบุ (d_1) คือ 1 ซึ่งมีค่ายกกำลังฐานสองคือ -1 ดังนั้นค่าประจำตำแหน่งนี้คือ 1×2^{-1}
- ตำแหน่งที่สอง (d_2) คือ 0 มีค่ายกกำลังฐานสองคือ -2 ดังนั้นค่าประจำตำแหน่งคือ 0×2^{-2}
- ตำแหน่งที่สาม (d_3) คือ 1 มีค่ายกกำลังฐานสองคือ -3 ดังนั้นค่าประจำตำแหน่งคือ 1×2^{-3}
- ตำแหน่งที่สี่ (d_4) คือ 1 มีค่ายกกำลังฐานสองคือ -4 ดังนั้นค่าประจำตำแหน่งคือ 1×2^{-4}

สามารถเขียนสมการได้คือ

$$\begin{aligned} N &= 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3} + 1\times2^{-4} \\ &= \frac{1}{2} + 0 + \frac{1}{2^3} + \frac{1}{2^4} \\ &= 0.5 + 0 + 0.125 + 0.0625 \\ 0.1011_2 &= 0.6875_{10} \end{aligned}$$

การแปลงเลขหลังทศนิยม (เศษส่วน) ฐานสิบ (Fractional Decimal Numbers) ให้เป็นฐานสอง

การเปลี่ยนเลขหลังทศนิยมฐานสิบ ให้เป็นฐานสอง จะใช้วิธีการนำค่าเลขหลังทศนิยมตั้ง แล้วคูณด้วยสอง จากนั้นนำผลลัพธ์ที่ได้เป็นตัวตั้งในการคูณครั้งต่อไป จนกว่าค่าผลลัพธ์ส่วนที่เป็น เลขหลังทศนิยมเท่ากับ .00 กรณีที่คูณแล้วไม่ลงตัวเท่ากับ .00 ก็ให้คูณจนได้ค่าที่ต้องการ สุดท้ายนำค่าตัวเลขก่อนทศนิยมจากผลลัพธ์แต่ละครั้ง มาเขียนเรียงต่อกัน ก็จะได้ค่าฐานสองที่ต้องการ ดังตัวอย่าง

<u>ตัวอย่าง</u> ต้องการแปลงเลข $(0.65625)_{10}$ เป็นเลขฐานสอง

พิจารณาทีละจุด

นำ 0.65625 คูณด้วย 2 ได้ค่าเท่ากับ 1.31250

- ค่า 1 (เลขก่อนทศนิยม) จะเป็นค่าหลักแรกของค่าเลขฐานสอง
- นำ .31250 (เลขหลังทศนิยม) ไปเป็นตัวตั้งในการคูณครั้งถัดไป

นำ 0.31250 คูณด้วย 2 ได้ค่าเท่ากับ 0.62500

- ค่า 0 (เลขก่อนทศนิยม) จะเป็นค่าหลักที่สองของค่าเลขฐานสอง

- นำ .62500 (เลขหลังทศนิยม) ไปเป็นตัวตั้งในการคูณครั้งถัดไป

นำ 0.62500 คูณด้วย 2 ได้ค่าเท่ากับ 1.25000

- ค่า 1 (เลขก่อนทศนิยม) จะเป็นค่าหลักที่สามของค่าเลขฐานสอง
- นำ .25000 (เลขหลังทศนิยม) ไปเป็นตัวตั้งในการคูณครั้งถัดไป

นำ 0.25000 คูณด้วย 2 ได้ค่าเท่ากับ 0.50000

- ค่า 0 (เลขก่อนทศนิยม) จะเป็นค่าหลักที่สี่ของค่าเลขฐานสอง
- นำ .50000 (เลขหลังทศนิยม) ไปเป็นตัวตั้งในการคูณครั้งถัดไป

นำ 0.5000 คูณด้วย 2 ได้ค่าเท่ากับ 1.00000

- ค่า 1 (เลขก่อนทศนิยม) จะเป็นค่าหลักที่ห้าของค่าเลขฐานสอง
- เนื่องจากเลขหลังทศนิยมเท่ากับ .00000 จึงไม่ต้องคูณต่อ

นำเลขก่อนทศนิยมของการคูณแต่ละครั้ง มาเขียนเรียงกัน จะได้ค่าเท่ากับ 10101 ดังนั้นเลขทศนิยมฐานสิบ 0.65625 จะเท่ากับ 0.10101 ในฐานสอง

การแปลงเลขฐาน 10 เป็นฐาน 8 และฐาน 16

การแปลงเลขฐาน 10 เป็นเลขฐาน 8 และเลขฐาน 16 มีลักษณะเดียวกับการแปลงเป็นเลขฐาน 2 โดยใช้ค่าฐานไปหารเพื่อหาเศษ ตัวอย่าง

ต้องการแปลง 169 เป็นเลขฐาน 8 กระทำได้โดย

8 หาร	169	เท่ากับ	21	เศษ	1
8 หาร	21	เท่ากับ	2	เศษ	5
8 หาร	2	ไม่สามารถหารได้		เศษ	2

ดังนั้น 169 เท่ากับ 251_8

ต้องการแปลง 169 เป็นฐาน 16 กระทำได้โดย

16 หาร	169	เท่ากับ	10	เศษ	9
16 หาร	10	ไม่สามารถหารได้		เศษ	10
แต่เนื่องจาก 10 เป็นค่าที่แสดงด้วย A					

ดังนั้น 169 เท่ากับ $A9_{16}$

การแปลงเลขฐาน 8 หรือ 16 เป็นฐาน 10 ก็ใช้วิธีเดียวกับการแปลงเลขฐาน 2 เป็นฐาน 10 ดังตัวอย่างที่แนะนำไปก่อนแล้ว

ระบบตัวเลขกับรหัสข้อมูล

รหัสข้อมูล (Data Representation) หมายถึง รหัสที่ใช้แทนตัวเลข ตัวอักษร สัญลักษณ์ต่างๆ ที่ประกอบอยู่ในคำสั่ง และข้อมูล เพื่อใช้ในการประมวลผล สามารถแบ่งได้ 2 ประเภทคือ

- รหัสภายในระบบคอมพิวเตอร์ (Internal Code) เป็นรหัสที่ใช้แทนข้อมูลในหน่วยความจำของคอมพิวเตอร์ เช่น
    - รหัส BCD - Binary Code Decimal
    - รหัส EBCDIC - Extended Binary Coded Decimal Interchange Code
    - รหัส ASCII - American Standard Code for Information Interchange
    - รหัส Unicode
- รหัสภายนอกระบบคอมพิวเตอร์ (External Code) เป็นรหัสที่พัฒนาสำหรับบันทึกข้อมูลนอกเครื่องคอมพิวเตอร์ เช่นรหัสที่ใช้กับบัตรเจาะรู

บิต (Bit)

สภาวะไฟฟ้า 1 เส้น หรือค่า 0 หรือ 1 แต่ละค่าเรียกว่า บิต (Bit) ซึ่งเป็นคำย่อของ "BInary digiT"

ไบต์ (Byte)

กลุ่มของบิตที่มีความหมายเฉพาะเรียกว่า ไบต์ (Byte) ดังนั้นถ้ามีสายสัญญาณ 8 เส้น แสดงว่ามีสัญญาณที่สามารถผสมผสานกันได้ 8 บิต เมื่อนำค่าสัญญาณต่างๆ มาผสมผสานกัน ก็สามารถสร้างรหัสแทนข้อมูลได้ จำนวน 28 = 256 ค่า เป็นต้น ดังตัวอย่างในตารางที่แสดงอักขระ, การเรียงกันของบิต และค่าเลขฐาน 10 ที่แทนอักขระ

ดังนั้นถ้าต้องการป้อนคำว่า Hello จะมีค่าเท่ากับข้อมูลจำนวน 6 ไบต์ ซึ่งมักจะได้ยินว่า 1 ไบต์ เทียบกับ 1 ตัวอักษรนั่นเอง

Binary Code Decimal (BCD)

BCD เป็นรหัสข้อมูลที่ประกอบด้วยเลขฐานสอง 6 บิต แทนข้อมูล 1 อักขระ (1 Character) จึงสามารถสร้างรหัสข้อมูลได้จำนวน $2^6 = 64$ รหัส

รหัสทั้ง 6 บิต แบ่งได้เป็น 2 กลุ่ม โดย 2 บิตแรกเรียกว่า Zone Bit และ 4 บิตถัดไปเรียกว่า Numeric Bit

Extended Binary Coded Decimal Interchange Code (EBCDIC)

EBCDIC เป็นรหัสแบบ 8 บิต โดยใช้เลขฐานสอง 8 ตัวแทนข้อมูล 1 อักขระ ทำให้สามารถสร้างรหัสได้ 256 รหัส (2^8) และยังสามารถใช้เลขฐาน 16 มาใช้แสดงรหัสข้อมูลได้เช่นกัน เป็นระบบการลงรหัสที่พัฒนาโดย IBM เนื่องจากพัฒนาจาก IBM ทำให้เป็นรหัสที่เด่นกว่า ASCII เมื่อนำไปใช้กับบัตรเจาะรู (Punched cards) ตั้งแต่ปี 1960 อีกทั้งยังมีอักขระ "cent sign" ซึ่งไม่มีใน ASCII

American Standard Code for Information Interchange (ASCII)

ASCII เป็นรหัสที่นิยมใช้กันอย่างแพร่หลายในปัจจุบัน พัฒนาโดยสถาบันมาตรฐานแห่งชาติสหรัฐอเมริกา (American National Standard Institute: ANSI) ประกอบด้วยเลขฐานสอง 7 บิต (ปัจจุบันใช้ 8 บิต) เรียกว่า 1 ไบต์ (Byte) แทนอักขระ 1 ตัว ซึ่งเป็นรหัสที่นิยมใช้กันบนคอมพิวเตอร์ระบบ PC ทั้งนี้ได้แบ่งเป็น 3 ชุดคือ

- 32 ชุดแรก (ตำแหน่งที่ 0 - 31) แทนรหัสควบคุมต่างๆ
- ตำแหน่งที่ 32 - 127 แทนอักขระภาษาอังกฤษ ตัวเลขและสัญลักษณ์ต่างๆ เรียกว่า Lower ASCII
- 128 ชุดหลัง (ตำแหน่งที่ 128 - 255) แทนอักขระในภาษาต่างๆ เช่น อักขระภาษาไทย เป็นต้น ทำให้คอมพิวเตอร์สามารถรับ/ส่งข้อมูลภาษาอื่นๆ ได้ เรียกว่า Higher ASCII

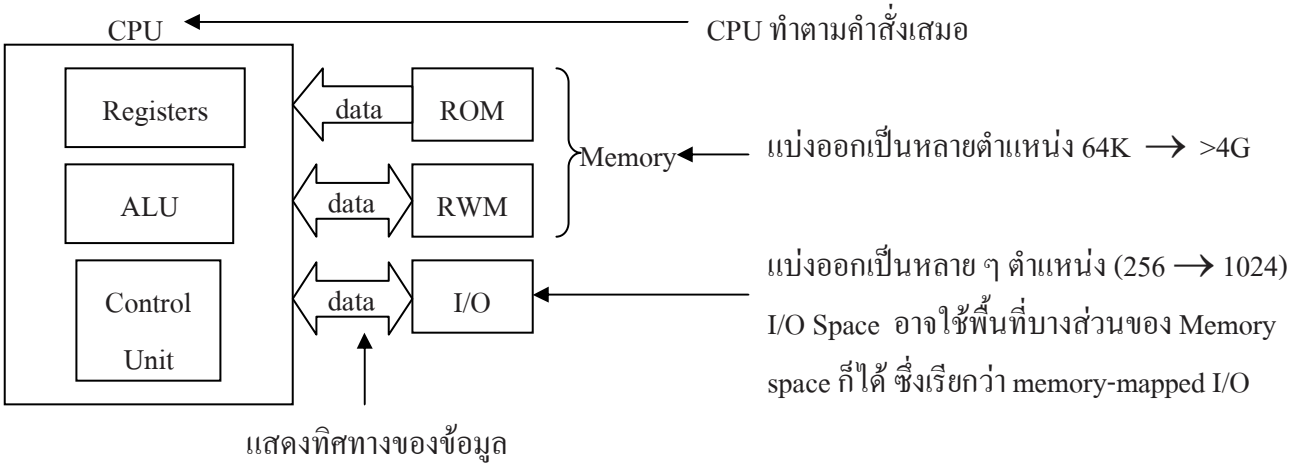
บทที่ 2 ระบบไมโครคอมพิวเตอร์

2.1 ระบบไมโครคอมพิวเตอร์พื้นฐาน

ระบบไมโครคอมพิวเตอร์โดยพื้นฐานประกอบไปด้วยองค์ประกอบ 3 ส่วนด้วยกันคือ

- หน่วยประมวลผลกลาง หรือ CPU (Central Processing Unit)
- หน่วยความจำ
- ส่วนที่ติดต่อกับอุปกรณ์ input/output

นอกจากองค์ประกอบหลักทั้ง 3 ส่วนนี้แล้ว ยังต้องมีส่วนสำคัญที่เป็นตัวเชื่อมโยงทุกส่วนให้ทำงานร่วมกันนั่นก็คือ ระบบเส้นทางในการติดต่อสื่อสารข้อมูล ดังแสดงในรูปที่ 2.1



รูปที่ 2.1 ระบบไมโครคอมพิวเตอร์พื้นฐาน

หน่วยความจำ (Memory) ทำหน้าที่เก็บคำสั่งและข้อมูลในรูปของเลขฐานสอง หน่วยความจำนี้สามารถแบ่งออกได้เป็น 2 ประเภทใหญ่ คือ

1. *หน่วยความจำหลัก (Main Memory)* เช่น ROM (Read-Only Memory) และ RWM (Read/Write Memory)
2. *หน่วยความจำเก็บข้อมูล (Storage Memory)* เช่น แผ่นดิสก์ และ ฮาร์ดดิสก์

โดยที่

ROM เป็นหน่วยความจำที่สามารถอ่านได้อย่างเดียว ไม่สามารถเปลี่ยนแปลงข้อมูลภายในได้ จึงใช้สำหรับเก็บข้อมูลที่ไม่ต้องการการเปลี่ยนแปลง โดยทั่วไปข้อมูลที่อยู่ภายใน ROM จะได้รับการเขียนบันทึกจากโรงงาน ซึ่งมักจะเป็นโปรแกรมที่สำคัญเพื่อให้ระบบเริ่มต้นทำงานเฉพาะอย่าง

RWM หรือที่นิยมเรียกกันว่า **RAM** (Random Access Memory) เป็นหน่วยความจำที่สามารถทั้งอ่านและเขียนได้ ใช้สำหรับเก็บโปรแกรมและข้อมูลทั่วๆไปของผู้ใช้ โดยที่หน่วยความจำประเภทนี้มีลักษณะเป็นแบบ *volatile* คือข้อมูลจะสูญหายไปจากหน่วยความจำเมื่อไม่มีไฟเลี้ยงระบบ

อุปกรณ์ input: ทำหน้าที่ถ่ายโอนข้อมูลที่อยู่ในรูปของเลขฐานสอง จากภายนอก เข้ามายัง CPU

อุปกรณ์ output: ทำหน้าที่ถ่ายโอนข้อมูลที่อยู่ในรูปของเลขฐานสอง จาก CPU ออกสู่ภายนอก

อุปกรณ์อินพุตอย่างเดียว - แป้นพิมพ์, เมาส์, สวิทช์, …

อุปกรณ์เอาท์พุตอย่างเดียว - เครื่องพิมพ์, จอภาพ, หลอดไฟ LED, …

อุปกรณ์ที่เป็นทั้งอินพุตและเอาท์พุต - secondary storage, communication devices (harddisk drives, floppy disk drives, tapes)

ระบบสายสัญญาณ (Bus System)

สายสัญญาณในการติดต่อสื่อสารข้อมูลระหว่างวงจรและอุปกรณ์ต่างๆ ที่ทำงานร่วมในระบบไมโครโพรเซสเซอร์มี 3 ประเภท คือ

- **Address Bus**: ทำงานได้ทิศทางเดียว ใช้ในการกำหนดตำแหน่งที่ใช้ในการติดต่อระหว่างวงจรและอุปกรณ์ต่างๆ ที่ต่อและทำงานร่วมกับ CPU ตามปกติ address bus นี้จะถูกกำหนดเป็น A_0 ถึง A_n โดย n เป็น MSB (Most Significant Bit) ของ address bus นั้น
- **Control Bus**: ใช้ควบคุมการติดต่อกับวงจรและอุปกรณ์ต่างๆ ที่ต่อร่วมกับ CPU โดย CPU จะส่งสัญญาณควบคุมออกไปเพื่อกำหนดการติดต่อให้อุปกรณ์ที่ต่อร่วมทราบ
- **Data Bus**: ใช้ในการรับและส่งข้อมูล ทำงานได้สองทิศทาง โดยจะทำงานหลังจาก address bus และ control bus ได้กำหนดตำแหน่งและการติดต่อสื่อสารแล้ว เพื่อเป็นการส่งข้อมูลถึงกันเช่นกันตามปกติ data bus นี้จะถูกกำหนดเป็น D_0 ถึง D_n โดย n เป็น MSB (Most Significant Bit) ของ data bus นั้น โดยที่เราจะเรียกประเภทของไมโครโพรเซสเซอร์ตามขนาดของ data bus ดังนี้ ไมโครโพรเซสเซอร์ 8 บิต เช่น ไมโครโพรเซสเซอร์ Z80 เนื่องจากมี data bus ขนาด 8 บิต เป็นต้น

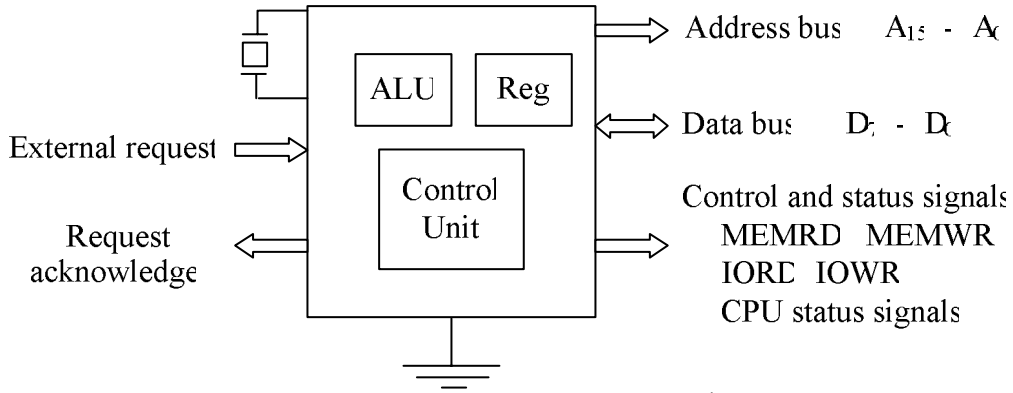
2.2 หน่วยประมวลผลกลาง

หน่วยประมวลผลกลาง (Central Process Unit: CPU) เป็นส่วนสำคัญที่สุดของระบบไมโครโพรเซสเซอร์ โดยมีวงจรควบคุมให้การทำงานเป็นไปตามคำสั่ง ส่วนประกอบหลักภายในวงจร CPU คือ

- หน่วยคำนวณทางตรรกะและคณิตศาสตร์ (Arithmetic and Logic Unit: ALU) ที่สามารถคำนวณได้ทั้งตัวเลขและตัวอักษร โดยการประมวลผลในส่วนของ ALU ที่สำคัญจะประกอบด้วย การบวก (add), การลบ (subtract), ลอจิก AND, ลอจิก OR, ลอจิก EX-OR, เปรียบเทียบ (compare),

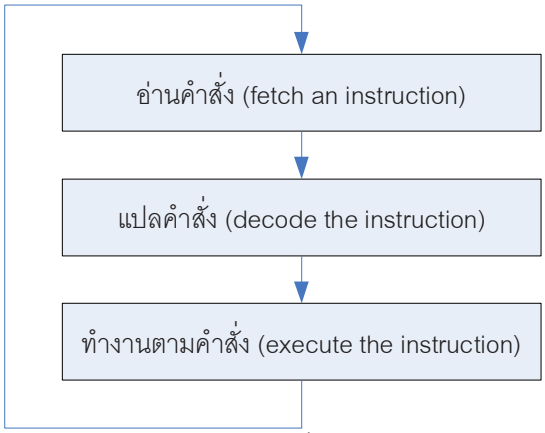
การเลื่อนบิตทางซ้ายหรือขวา, การเพิ่มค่า (increment), การลดค่า (decrement), การเซ็ทบิต (set bit), การรีเซ็ทบิต (reset bit), การทดสอบบิต (test bit)

- หน่วยควบคุม (Control Unit) ที่ทำหน้าที่ควบคุมการทำงานภายในให้ปฏิบัติตามคำสั่งที่กำหนด
- วงจรหน่วยความจำภายในหรือ register ที่ทำหน้าที่เก็บค่าตัวแปรต่างๆ ที่ใช้ในการคำนวณและผลลัพธ์ที่ได้จากการคำนวณ ดังแสดงในรูปที่ 2.2



รูปที่ 2.2 ระบบไมโครโพรเซสเซอร์พื้นฐาน

การทำงานของ CPU มีอยู่ด้วยกัน 3 ลักษณะหลักๆ คือ



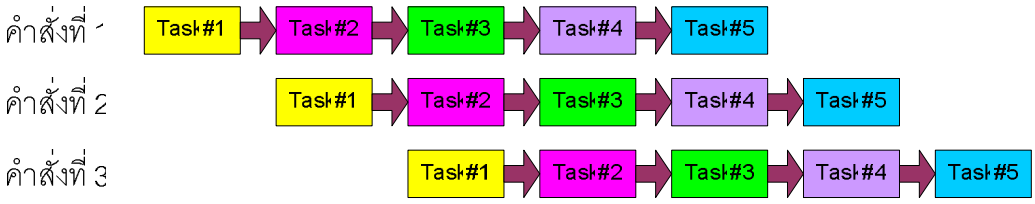
รูปที่ 2.3การทำงานพื้นฐานของ CPU

- **Fetch**: CPU ทำการอ่านคำสั่งที่เป็นเลขฐานสองจากหน่วยความจำ
- **Decode:** CPU ทำการแปลหรือถอดรหัสของคำสั่งที่อ่านมา เพื่อหาว่าเป็นคำสั่งประเภทไหน มีการทำงานอย่างไร รวมทั้งต้องมีการติดต่อกับส่วนอื่นๆ เช่น หน่วยความจำหรือ I/O บ้างหรือไม่อย่างไร
- **Execute**: CPU นำคำสั่งนั้นไปปฏิบัติ โดยการทำงานต่างๆ ในขั้นตอนนี้ จะขึ้นอยู่กับว่าคำสั่งที่ได้จากการถอดรหัสเป็นอะไร และมีการติดต่อกับส่วนใดบ้าง ซึ่งแต่ละคำสั่งจะมีจำนวนรหัสเท่าไรนั้นก็ขึ้นอยู่กับชนิดของ CPU เช่น Z80 จะมีขนาดตั้งแต่ 1-4 bytes ส่วน MCS-51 จะมีขนาดตั้งแต่ 1-3 bytes เป็นต้น

การทำงานของ CPU อาจเลื่อมกันได้แบบ *pipelining*

Pipeline คือการทำงานแบบคาบเกี่ยวกัน (overlap) โดยการแบ่ง CPU ออกเป็นส่วนย่อย ๆ แล้วแบ่งงานกันรับผิดชอบ โดยเดิมที pipeline เป็นเทคนิคของสถาปัตยกรรมแบบ RISC (Reduced Instruction Set Computer) ต่อมานำมาใช้กับสถาปัตยกรรมแบบ CISC (Complex Instruction Set Computer) ซึ่งการทำงานของระบบที่มี pipeline นั้นก็คือ จะสามารถรับงานใหม่ได้ในขณะที่ยังคงทำงานเก่าต่อไปได้ด้วย

จากรูปจะเห็นว่า ขั้นตอนการทำงานของแต่ละคำสั่งแบ่งการทำงานออกเป็น 5 ขั้นตอน ซึ่งสามารถแยกการทำงานออกจากกันเป็นขั้นตอนได้ เมื่อคำสั่งแรกทำ task#1 เสร็จจะเข้าสู่ขั้นตอน task#2 ซึ่งใน cycle นี้ ในส่วนที่ทำ task#1 จะว่าง คำสั่งต่อไปสามารถทำ task#1 ได้เลย โดยไม่ต้องรอให้คำสั่งแรกทำงานจนเสร็จถึงขั้นตอน task#5 ซึ่งจะทำให้การทำงานเร็วขึ้นถึงประมาณ 5 เท่าจากปกติ ดังรูปที่ 2.4



รูปที่ 2.4 Pipelining

Z80 ทำงานแบบธรรมดา ไม่มี pipeline

2.3 สถาปัตยกรรมของไมโครโพรเซสเซอร์

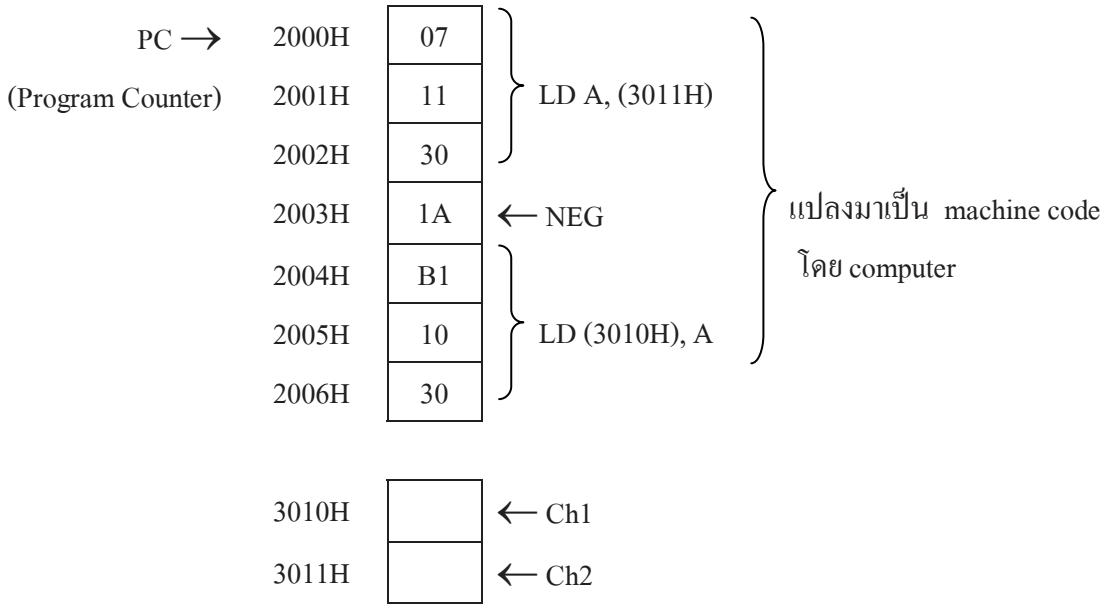
หมายถึง โครงสร้างและการทำงานของไมโครโพรเซสเซอร์อย่างกว้างๆ ความหมายไม่ได้จำกัดอยู่เฉพาะภายในไมโครโพรเซสเซอร์แค่นั้น แต่อาจรวมถึงฮาร์ดแวร์และซอฟต์แวร์ภายนอกที่ใช้งานกับไมโครโพรเซสเซอร์ก็ได้

ไมโครโพรเซสเซอร์แต่ละเบอร์จะมีสถาปัตยกรรมที่ต่างกัน โดยทั่วไปเราจะเปรียบเทียบไมโครโพรเซสเซอร์ใน องค์ประกอบหลักต่อไปนี้

1. ขนาด data word → ยิ่งใหญ่ ยิ่งเร็ว
2. ขนาด address bus → ยิ่งใหญ่ ยิ่งมีหน่วยความจำมาก
3. ความเร็ว

องค์ประกอบรองที่ใช้ในการเปรียบเทียบ เช่น

1. มี register กี่ตัว มีชนิดอะไรบ้าง
2. จำนวนคำสั่ง และชนิดของคำสั่ง
3. วิธีการที่คำสั่งอ้างถึงข้อมูล (addressing modes) มีกี่วิธี อะไรบ้าง
4. วงจรหรืออุปกรณ์ประกอบมีอะไรบ้าง

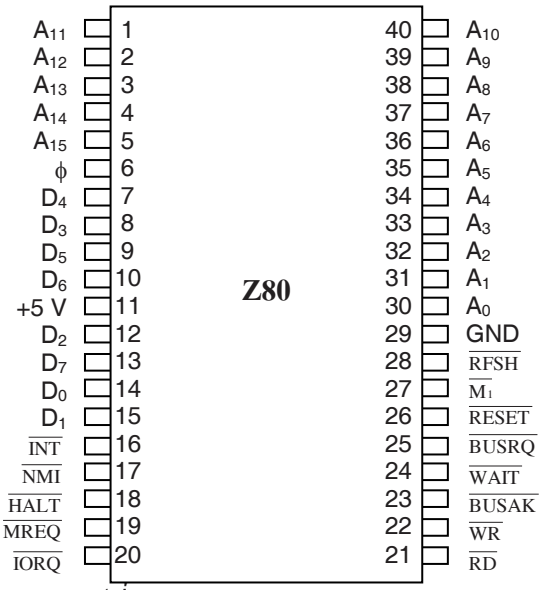

PC →
(Program Counter)
2000H
2001H
2002H
2003H
2004H
2005H
2006H
07
11
30
1A
B1
10
30
LD A, (3011H)
← NEG
LD (3010H), A
แปลงมาเป็น machine code
โดย computer
3010H
3011H
← Ch1
← Ch2

ทำไมต้องเป็น Z80: Z80 เป็นไมโครโพรเซสเซอร์ที่มีโครงสร้างง่ายต่อการเรียนรู้และการจัดวงจรแต่ละขาของตัวไมโครโพรเซสเซอร์มีหน้าที่เดียว ไม่ซับซ้อน ผู้ที่เริ่มต้นศึกษา Z80 จะเกิดความรู้สึกว่า การเรียนรู้ไมโครโพรเซสเซอร์ไม่ใช่เรื่องยาก

โครงสร้างของไมโครโพรเซสเซอร์ Z80 มีโครงสร้างที่พัฒนามาจาก 8080 ดังนั้นในแง่โครงสร้างพื้นฐานจะ เหมือนกับ CPU 8080 แต่เนื่องจาก Z80 มีการพัฒนามากขึ้นทางซอฟต์แวร์และฮาร์ดแวร์จึงทำให้มีรายละเอียดแตกต่างเพิ่มเติมอีกหลายประการด้วยกัน

3.1 ขาสัญญาณบน Z80

Z80 เป็น IC ขนาด 40 ขา แบบ DIP (Dual Inline package) ดังแสดงในรูปที่ 3.1 โดยมีโครงสร้างภายในและตำแหน่งของขาต่างๆ ดังต่อไปนี้

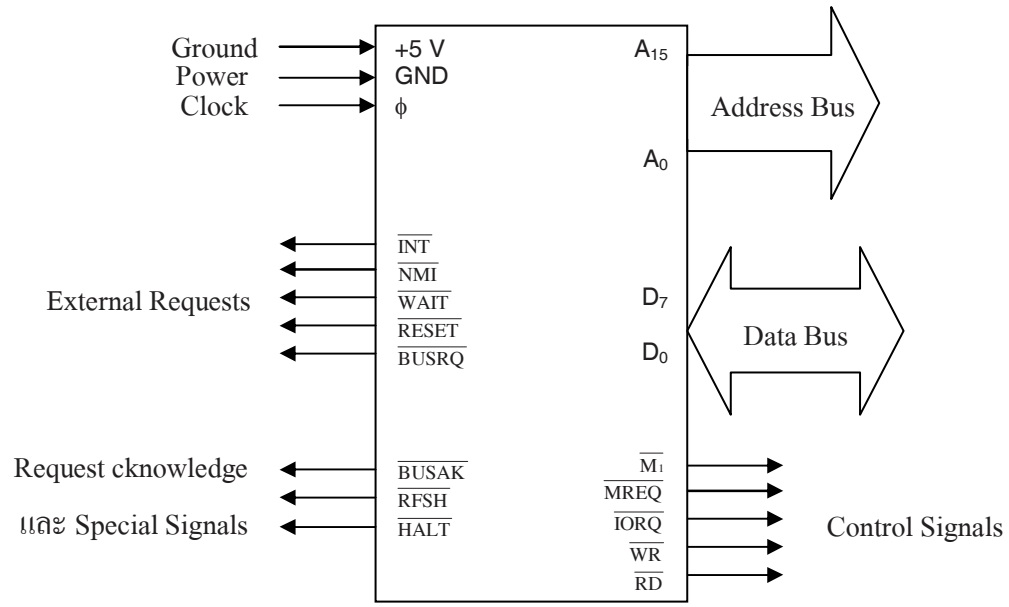


รูปที่ 3.1 Z80 Microprocessor Pin layout

หน้าที่ของขาสัญญาณบน Z80

จากรูปที่ 3.1 ขาและสัญญาณต่างๆของ Z80 สามารถแบ่งออกได้เป็น 6 กลุ่มดังในรูปที่ 3.2 คือ

1. Address bus
2. Data bus
3. Control signals
4. External requests
5. Request acknowledge และ Special signals
6. Power และ Frequency signals



รูปที่ 3.2 Z80 Microprocessor Logic Signals

โดยรายละเอียดของขาและสัญญาณต่างๆ มีดังนี้

Address Bus:

สายสัญญาณ tri-state ซึ่งเป็นสายสัญญาณที่มีทิศทางเดียว (unidirectional) 16 สัญญาณ (A_{15}- A_0) ใช้ในการส่งตำแหน่งของ memory registers หรือ อุปกรณ์ I/O ต่างๆ ซึ่งจะสามารถอ้างได้ถึง 64k (2^{16}) ตำแหน่ง

Data Bus:

สายสัญญาณ tri-state ซึ่งเป็นสายสัญญาณที่มี 2 ทิศทาง (bidirectional) 8 สัญญาณ (D_7- D_0) ใช้ในการโอนย้ายข้อมูล โดยในข้อมูลสามารถเคลื่อนที่ในทิศทางใดก็ได้ในแต่ละสายสัญญาณคืออาจจะเป็นจากไมโครโพรเซสเซอร์ไปยัง memory หรืออุปกรณ์ I/O ต่างๆ หรือในทางกลับกันคือจาก memory หรืออุปกรณ์ I/O ต่างๆ ไปยังไมโครโพรเซสเซอร์

Control และ status signals:

- $\overline{M1}$ - *Machine cycle one*: ขาสัญญาณที่ active low โดยจะ active เมื่อ opcode ถูก fetch จาก memory
- $\overline{MREQ}$ - *Memory request*: ขาสัญญาณ tri-state ที่ active low โดยจะ active เมื่อมีการ read/write กับ memory โดยเป็นการบอกว่า ค่าตำแหน่งของข้อมูลใน memory ที่ต้องการ read/write ได้อยู่ที่ address bus แล้ว
- $\overline{IORQ}$ - *I/O request*: ขาสัญญาณ tri-state ที่ active low โดยจะ active เมื่อมีการ read/write กับ I/O โดยเป็นการบอกว่า ค่าตำแหน่งของข้อมูลของ I/O ที่ต้องการ read/write ได้อยู่ที่ low-order address bus (A_7 - A_0) แล้ว
- $\overline{RD}$ - *Read*: ขาสัญญาณ tri-state ที่ active low โดยจะ active เมื่อมีการ read จาก memory หรือ I/O
- $\overline{WR}$ - *Write*: ขาสัญญาณ tri-state ที่ active low โดยจะ active เมื่อมีการ write จาก memory หรือ I/O

External Requests :

- $\overline{RESET}$ - *Reset*: ขาสัญญาณที่ active low โดยเป็นการ reset CPU ซึ่งจะต้อง active อย่างน้อย 3 clock periods เพื่อ clear ค่า PC, R, I โดยที่ระหว่าง reset สายสัญญาณต่างๆ จะไม่ active

- $\overline{\text{INT}}$ - *Interrupt request*: (level-sensitive) ขาสัญญาณที่ active low เพื่อขอขัดจังหวะการทำงานของ CPU รับการขัดจังหวะ (interrupt acknowledge) จะมีสัญญาณ $\overline{\text{IORQ}}$ ในช่วง M_1 ด้วย
- $\overline{\text{NMI}}$ - *Nonmaskable interrupt*: (edge-sensitive) ขาสัญญาณที่ active low ถ้าสัญญาณนี้ active จะขัดจังหวะการทำงานของ CPU ไม่สามารถจะ disable ได้ ใช้สำหรับการทำงานฉุกเฉิน
- $\overline{\text{BUSAK}}$ - *Bus acknowledge*: ขาสัญญาณที่ active low โดยจะ active เพื่อแสดงว่า CPU ตอบรับ $\overline{\text{BUSRQ}}$
- $\overline{\text{WAIT}}$ - *Wait*: ขาสัญญาณที่ active low โดยเป็นสัญญาณที่ขอให้ CPU คอยก่อน

Request Acknowledge และ Special Signals:

- $\overline{\text{HALT}}$ - *Halt*: ขาสัญญาณที่ active low โดยเป็นสัญญาณแสดงว่า CPU อยู่ในสถานะ halt
- $\overline{\text{RFSH}}$ - *Refresh*: ขาสัญญาณที่ active low โดยจะ active เมื่อ CPU จะทำการ refresh DRAM
- $\overline{\text{BUSRQ}}$ - *Bus request*: ขาสัญญาณที่ active low โดยอุปกรณ์ภายนอกจะส่งสัญญาณนี้เพื่อขอควบคุม bus เอง

Clock และ Power Supply:

- ϕ - *Clock*: ต่อกับแหล่งความถี่ภายนอก เนื่องจาก Z80 ไม่มีวงจรนาฬิกาภายในตัว chip
- +5V - *Power supply*: แหล่งจ่ายไฟเลี้ยงระบบ (ควรมี Capacitor ชนิด tantalum ต่อคร่อม power supply)
- GND - *Ground*: ขาสายดิน

3.2 รีจิสเตอร์บนไมโครโพรเซสเซอร์ Z80

ส่วนของไมโครโพรเซสเซอร์ที่เกี่ยวข้องกับการเขียนโปรแกรม ซึ่งได้แก่ รีจิสเตอร์ (register) ที่ผู้เขียนโปรแกรมสามารถใช้งานได้ โครงสร้างภายในของไมโครโพรเซสเซอร์ Z80 ประกอบด้วย รีจิสเตอร์ภายในที่สามารถเขียนและอ่านได้ถึง 208 บิต โดยแบ่งได้เป็นกลุ่มของรีจิสเตอร์ขนาด 8 บิต 18 รีจิสเตอร์และรีจิสเตอร์ขนาด 16 บิต อีก 4 รีจิสเตอร์

รีจิสเตอร์ใน Z80 เมื่อแบ่งตามหน้าที่สามารถแบ่งได้เป็นดังนี้

กลุ่มรีจิสเตอร์หลัก (Main Registers)

รีจิสเตอร์ในกลุ่มแรก คือ A, B, C, D, E, H, L และ F เป็นรีจิสเตอร์ขนาด 8 บิต ที่ใช้งานทั่วไปโดยรีจิสเตอร์ เหล่านี้สามารถประกอบรวมกันเป็นคู่รีจิสเตอร์ได้ คือ AF, BC, DE และ HL โดยคู่รีจิสเตอร์เหล่านี้ได้รับการใช้งานในลักษณะของรีจิสเตอร์ ขนาด 16 บิต การกระทำภายใน CPU อาจจะอาศัยเพียงรีจิสเตอร์เดียวหรือกระทำเป็นคู่รีจิสเตอร์ได้ รูปที่ 3.3 แสดงให้เห็นถึงทั้งรีจิสเตอร์หลักและรีจิสเตอร์รอง (ซึ่งจะกล่าวต่อไป) ที่มีอยู่ใน Z80 โดยที่ A คือ Accumulator และ F คือ flag ซึ่งมีลักษณะพิเศษเฉพาะ จึงขออธิบายรายละเอียดของรีจิสเตอร์ทั้งสองตัวนี้เพิ่มเติมดังนี้

Accumulator และ Flag register

CPU จะมีรีจิสเตอร์ที่ใช้เป็นหลักในการ เป็นตัว operand สำหรับกระทำทางคณิตศาสตร์และลอจิก โดยรีจิสเตอร์หลักนี้จะมีเพียง 8 บิต เรียกว่า accumulator การกระทำในส่วนของหน่วยคณิตศาสตร์ และทางตรรกศาสตร์ย่อมเกิดเงื่อนไขได้หลายอย่างที่จะต้องแสดงสถานะภาพของเงื่อนไขเหล่านั้น เช่น เงื่อนไขผลลัพธ์

เป็นศูนย์ผลลัพธ์เป็นบวกหรือลบมีตัวทดหรือตัวขอยืมในการกระทำ ทางคณิตศาสตร์ แสดงเงื่อนไข parity คู่หรือคี่ สิ่งเหล่านี้จะให้ผลลัพธ์แสดงสถานะ ได้ด้วย flag

Main Register		Alternate Register		การเรียกใช้ register pairs
A	F	A′	F′	AF, BC, DE, HL
B	C	B′	C′	
D	E	D′	E′	
H	L	H′	L′	

8 บิต 8 บิต

16 บิต

A - Accumulator เป็น register ที่ใช้บ่อย โดยเฉพาะคำสั่งที่เกี่ยวข้องกับการคำนวณ

F - Flag register บอกสถานะของผลลัพธ์จากการทำคำสั่งที่เกี่ยวข้องกับการคำนวณ

ส่วนที่เหลือ - General purposed register

รูปที่ 3.3 กลุ่มรีจิสเตอร์หลักและสำรองขนาด 8-bit ของไมโครโพรเซสเซอร์ Z80

Flag ของ Z80 จะมีด้วยกันทั้งหมด 6 ตัว จึงใช้เพียง 6 บิต แต่ Z80 อาศัยการเพิ่มบิตขึ้นอีก 2 บิต และกลายเป็น register F ซึ่งมีขนาด 8 บิต รูปที่ 3.4 แสดงถึงโครงสร้างภายในของแต่ละบิตใน flag register โดยที่รายละเอียดของแต่ละบิตจะมีแสดงอยู่ในตารางที่ 3.1

รีจิสเตอร์ F นี้สามารถได้รับการ set, reset, การกระทำตามคำสั่งทางคณิตศาสตร์ หรือ ทางตรรกศาสตร์ได้ และเราสามารถใช้ F เหมือนรีจิสเตอร์หนึ่ง ซึ่งเมื่อรวมกับ accumulator (A) แล้ว จะกลายเป็นรีจิสเตอร์ขนาด 16บิต ได้ โดยที่คนเขียนโปรแกรมยังสามารถใช้คำสั่งในการเคลื่อนย้ายข้อมูลจาก accumulator (A) และ flag register (F) ไปเก็บไว้ใน A' และ F' ได้ เพื่อทำให้การใช้งาน ของ A และ F มีประสิทธิภาพดียิ่งขึ้น

	7	6	5	4	3	2	1	0
Flag register	S	Z	X	H	X	P/V	N	C

รูปที่ 3.4 Flag Register ของไมโครโพรเซสเซอร์ Z80

S	sign flag	0 - ถ้า bit ที่ 7 ของผลลัพธ์เป็น 0	ใช้กับ signed number
		1 - ถ้า bit ที่ 7 ของผลลัพธ์เป็น 1	
Z	zero flag	0 – ถ้าผลลัพธ์ ≠ 0	
		1 – ถ้าทุก bit ของผลลัพธ์ เป็น 0	

H	half carry flag	0 – ถ้าไม่มีการทด/ยืมระหว่าง bit บน-ล่าง	
		1 - ถ้ามีการทด/ยืมระหว่าง bit บน-ล่าง	
P/V	parity/overflow	0 – ถ้าไม่เกิด overflow	กรณี overflow (สำหรับ signed number)
		1 - ถ้าเกิด overflow	
		0 – ถ้าจำนวน bit ที่เป็น 1 เป็นเลขคี่	กรณี parity
		1 - ถ้าจำนวน bit ที่เป็น 1 เป็นเลขคู่	
N	add/subtract	0 – ถ้าเป็นคำสั่งประเภท add	
		1 - ถ้าเป็นคำสั่งประเภท subtract	
C	Carry flag	0 - ถ้าไม่มีการทด/ยืม ที่ bitที่ 7 หรือ bit ที่ 15	เสมือนเป็น bit ที่ 8 หรือ 16
		1 – ถ้ามีการทด/ยืม ที่ bit ที่ 7 หรือ bit ที่ 15	

ตารางที่ 3.1 รายละเอียดของแต่ละบิตใน Flag Register

บางคำสั่งมีผลต่อ carry flag โดยตรง เช่น

SCF - set carry flag

CCF - complement carry flag Note: *ไม่มีคำสั่งที่ reset carry flag*

คำสั่งประเภท rotate และ shift ก็มีผลต่อ carry flag ได้ด้วย

กลุ่มรีจิสเตอร์สำรอง (Alternate Register)

รีจิสเตอร์ชุดนี้เป็นรีจิสเตอร์ที่ใช้ในการเก็บข้อมูลชั่วคราว โดยเป็นตัวเก็บข้อมูลที่มาจากรีจิสเตอร์หลัก ในการที่ต้องการใช้รีจิสเตอร์หลักทำงานอย่างอื่นก่อน ดังนั้นรีจิสเตอร์กลุ่มนี้จึงไม่สามารถกระทำทางคณิตศาสตร์และทางตรรกศาสตร์ได้ รีจิสเตอร์เหล่านี้มีด้วยกัน 8 ตัว คือ A', B', C', D', E', H', L' และ F'

กลุ่มรีจิสเตอร์ที่ใช้งานเฉพาะอย่างขนาด 16 บิต

นอกเหนือจาก Accumulator และ Flag register ที่มีหน้าที่เฉพาะอย่างแล้ว Z80 ยังปบระกอบไปด้วย register ที่มีหน้าที่เฉพาะอย่างอื่นๆ อีก ซึ่งมีขนาด 16 บิต (ในขณะที่ A และ F ต่างก็มีขนาดเพียง 8 บิต) คือ

Program counter (PC) เป็นรีจิสเตอร์ขนาด 16 บิต ที่เป็นตัวกำหนดตำแหน่งของโปรแกรมในขณะสภาวะการกระทำการ fetch โดยขณะทำ การ fetch ค่าที่อยู่ใน PC จะไปปรากฏอยู่ที่ address bus เพื่อชี้ไปยังตำแหน่งในหน่วยความจำ ให้ CPU อ่านคำสั่งมาตีความหมายค่าที่อยู่ใน PC จะเพิ่มค่าขึ้นได้อย่างอัตโนมัติ หลังการกระทำการ fetch แต่ถ้าหาก CPU กระทำคำสั่งให้ข้ามไปยังตำแหน่งอื่น (Jump) ค่า address ที่จะกระโดดข้ามนั้นจะโหลดเข้ามายัง PC ได้อย่างอัตโนมัติ

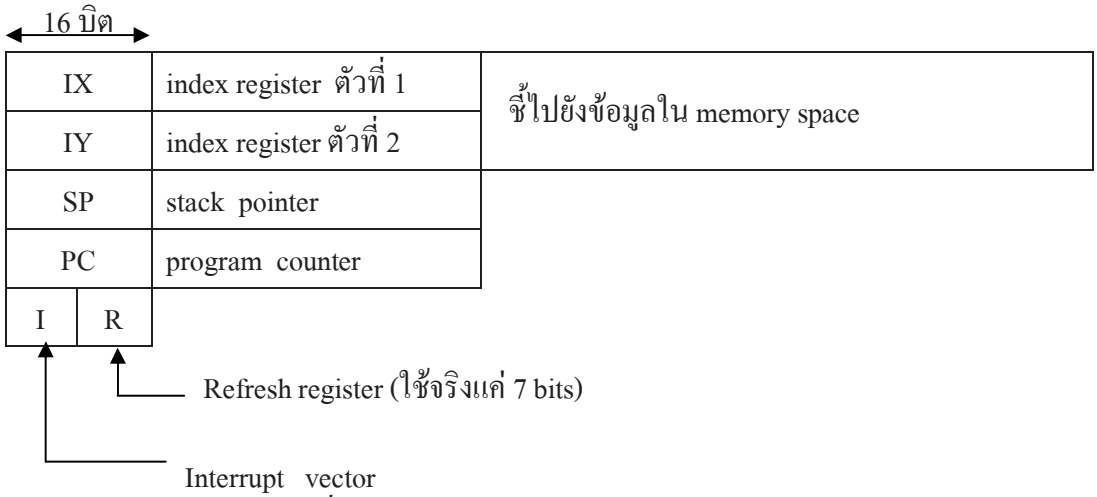
Stack pointer (SP) เป็นรีจิสเตอร์ที่มีขนาด 16 บิตที่ใช้สำหรับชี้ไปยัง address ชั้นบนสุดของ stack ที่อยู่ใน RAM โดยส่วนของ stack มีลักษณะโครงสร้าง เป็นหน่วยความจำ เป็นแบบเก็บทีหลังเรียกออกก่อน (last-in-first-out) ข้อมูลใน stack อาจได้รับการ push หรือ pop มาจาก รีจิสเตอร์ภายใน CPU ลักษณะของ stack ในที่นี้ยังเป็นส่วนช่วยในการกระทำ interrupt และการเรียก โปรแกรมย่อย กล่าวคือ ในการ interrupt ค่าของโปรแกรม

เคาน์เตอร์จะได้รับการรักษาไว้ในชั้น stack ครั้นเมื่อโปรแกรมกลับจาก interrupt ไปกระทำยังโปรแกรมหลักก็จะนำค่าจาก stack กลับเข้ามายัง PC ใหม่ ในทำนองเดียวกัน การกระโดดไปกระทำยังโปรแกรมย่อยก็เช่นเดียวกัน ดังนั้น การกระทำในรูปของ interrupt ของ โปรแกรมย่อยสามารถ ซ้อนกันได้ไม่มีสิ้นสุด

Index Register (IX, IY) Z80 มี index register ขนาด 16 บิตอยู่ 2 ตัว แต่ละตัวใช้ประโยชน์หลักในการทำหน้าที่เป็นตัวเก็บ base address เพื่อ ทำหน้าที่อ้าง address แบบ index addressing ซึ่งใน mode นี้ซึ่งมีข้อมูลที่อยู่ใน index register นี้จะใช้รวมกับข้อมูลที่ติดมากับคำสั่ง (operand) อีก 8 บิต เพื่อเป็นตัวกำหนด address ให้กับคำสั่ง ข้อมูลที่ติดมากับคำสั่งนี้เราเรียกว่า displacement ซึ่งจะเก็บในรูปของตัวเลข 2's complement

Interrupt page address register (I) การ interrupt ของ Z80 มีด้วยกันอยู่หลาย modes และ mode หนึ่งที่ทำให้การ interrupt ของ Z80 มีประสิทธิภาพสูง กล่าวคือ เมื่อเกิดการ interrupt ใน mode นี้ขึ้น มันสามารถอ้างถึง address โดยทางอ้อม ไปทำงานในที่ใดก็ได้ในหน่วยความจำ โดยอาศัยค่าในรีจิสเตอร์ I นี้ซึ่งมีขนาดเพียง 8 บิต รวมกับค่าที่ส่งมาจากอุปกรณ์ peripheral อีก 8 บิต ชี้ไปยังค่าในหน่วยความจำเพื่อนำค่านั้นมา load เข้าใน PC เพื่อทำงานต่อไป ด้วยวิธีการนี้เราจึงสามารถกระโดดเข้าไปทำที่ส่วนใดก็ได้ในหน่วยความจำ

Memory refresh register (R) การต่อไมโครโพรเซสเซอร์กับหน่วยความจำนั้น โดย ปกติจะต่อกับหน่วยความจำชนิด static ได้โดยง่าย แต่ชนิด dynamic ที่ต้องการการ refresh มีราคาถูกกว่ามีความหนาแน่นสูงกว่า Z80 ให้ข้อดีกว่าประการหนึ่งคือมันสามารถให้การ refresh หน่วยความจำได้ อย่างอัตโนมัติ โดยค่าใน R รีจิสเตอร์จะเพิ่มค่าขึ้นอีก 1 ทุกครั้งที่มีการกระทำการ fetch คำสั่ง และ ข้อมูลในรีจิสเตอร์ R นี้ซึ่งมีขนาด 8 บิต จะถูกส่งออกไปยัง address bus ในส่วนบิตที่มีนัยสำคัญต่ำกว่าจังหวะของการส่งนี้จะเป็นจังหวะเดียวกันกับที่ CPU ส่งสัญญาณ refresh ออกมา ผู้โปรแกรมสามารถกำหนดค่าให้กับ รีจิสเตอร์ R นี้ได้แต่ค่าในรีจิสเตอร์ นี้จะเรียกใช้โดยผู้โปรแกรมทางคำสั่งโดยตรงไม่ได้



รูปที่ 3.5 16-bit Registers ของไมโครโพรเซสเซอร์ Z80

3.3 รูปแบบของคำสั่งในไมโครโพรเซสเซอร์ Z80

ในทุกคำสั่งจะมีรูปแบบที่เป็นมาตรฐาน โดยจะประกอบไปด้วย ส่วนรหัสคำสั่ง (mnemonic – อ่านว่า นีโมนิก หรือ operation code– opcode) และส่วนตัวแปรดำเนินการ (operand) โดยในส่วนของรหัสคำสั่งนั้นมักเป็น

คำย่อของการกระทำ เช่น LD ซึ่งมาจากคำว่า load หมายถึง การโอนข้อมูล หรือ INC ซึ่งมาจากคำว่า increment หมายถึง การเพิ่มขึ้น สำหรับในส่วนของ operand จะแสดงด้วย register และตัวเลข โดยจำนวนของ operand นั้นขึ้นอยู่กับชนิดของคำสั่ง

คำสั่งแต่ละคำสั่งของ Z80 จะมีขนาดไม่เท่ากัน โดยที่บางคำสั่งมีความยาวเพียง 1 byte และบางคำสั่งอาจมีมากกว่านั้น ในการทำงานตามคำสั่งหนึ่งๆ Z80 อาจจะต้องมีการติดต่อกับส่วนอื่นเช่น memory หรือ I/O เพื่อ read/write ข้อมูลที่จำเป็นในการทำงานของคำสั่งนั้นๆ

อย่างไรก็ตามความยาวของคำสั่งไม่ได้บ่งบอกถึงจำนวนชิ้นงานที่ CPU ต้องทำเพื่อให้ทำงานคำสั่งหนึ่งๆ นั้น เสมอไป ตัวอย่างเช่น คำสั่ง OUT(10H),A ซึ่งมีความยาว 2 bytes โดยจะเป็นการส่งข้อมูลใน accumulator ไปยัง output port หมายเลข 10_H

Byte ที่ 1 เก็บ OUT → opcode บอกว่าเป็นการส่งข้อมูลออกพอร์ต

Byte ที่ 2 เก็บ (10H), A → operand บอกว่าข้อมูลขนาด 1 byte จาก accumulator จะถูกส่งไปยังพอร์ตหมายเลข 10_H

โดยคำสั่งนี้ Z80 จะมีการทำงาน 3 อย่างดังนี้

1. อ่าน byte แรก จากหน่วยความจำ
2. อ่าน byte ที่สอง จากหน่วยความจำ
3. ส่งข้อมูลไปยังพอร์ตหมายเลข 10_H

รูปแบบของคำสั่งของ Z80 มีด้วยกัน 4 ประเภทดังต่อไปนี้

1-byte instructions

เช่น	LD	A, B	01 111 000 (78_H)	

2-byte instructions

เช่น	LD	B, 32H	0000 0110 (06_H)	Byte ที่ 1
			0011 0010 (32_H)	Byte ที่ 2

3-byte instructions

เช่น	LD	BC, 2080H	0000 0001 (01_H)	Byte ที่ 1
			1000 0000 (80_H)	Byte ที่ 2
			0010 0000 (20_H)	byte ที่ 3

4-byte instructions

เช่น	LD	IX, 2000H	1101 1101 (DD_H)	Byte ที่ 1
			0010 0001 (21_H)	Byte ที่ 2
			0000 0000 (00_H)	Byte ที่ 3
			0010 0000 (20_H)	Byte ที่ 4

3.4 Z80 Addressing Modes

Addressing mode คือวิธีการได้มาซึ่งข้อมูลที่ได้อ้างถึงใน operand ของแต่ละคำสั่ง โดยที่ Z80 มี addressing modes ทั้งหมด 10 ประเภท ดังแสดงในตารางที่ 3.2

Addressing Modes	คำอธิบาย	ตัวอย่าง
Immediate	operand ตัวที่สองเป็นข้อมูลขนาด 8 บิตเพื่อ load ใส่ register เดียว	LD B,97H
Immediate Extended	operand ตัวที่สองเป็นข้อมูลหรือตำแหน่งขนาด 16 บิตเพื่อ load ใส่ register คู่	LD BC,8045H
Register	operand ตัวที่สองเป็น register	LD B,A
Implied	เป็นคำสั่งที่มีการอ้างถึง register ตัวอื่นที่ไม่ได้ปรากฏอยู่ใน operand โดยส่วนใหญ่จะเป็นการอ้างถึง register A	AND B
Register Indirect	ค่าที่อยู่ใน register คู่ จะเป็นตำแหน่งในหน่วยความจำ (memory pointer) ของข้อมูลที่ต้องการอ้างถึง	LD B,(HL)
Extended	ข้อมูลขนาด 16 บิต คือ ตำแหน่งของคำสั่งต่อมาที่ต้องการข้ามกระโดดไป	JP 2080H
Relative	operand จะเป็นเพียงค่า displacement ในรูปของ 2's complement	JR 14H
Indexed	ตำแหน่งของข้อมูลที่ต้องการหาได้จากการหาผลบวกของ index register และ ค่า displacement ขนาด 1 byte	INC (IX+10H)
Bit	mode นี้ใช้สำหรับการทำงานแบบบิต (bit operation) โดยมีการกำหนดบิตที่ต้องการจะกระทำการจาก register หรือตำแหน่งในหน่วยความจำหนึ่งๆ โดยวิธีใดวิธีหนึ่งจาก 3 โหมดนี้ (register, register indirect หรือ indexed)	SET 7,B
Page Zero	ประกอบไปด้วย 8 คำสั่ง restart ซึ่งเป็น one-byte call โดยตำแหน่งที่จะเรียกใช้จะมี byte บนเป็น 00_H จึงสามารถกำหนดเฉพาะ byte ล่างในคำสั่งได้	RST 28H

ตารางที่ 3.2 รายละเอียดของ Addressing Mode ของไมโครโพรเซสเซอร์ Z80

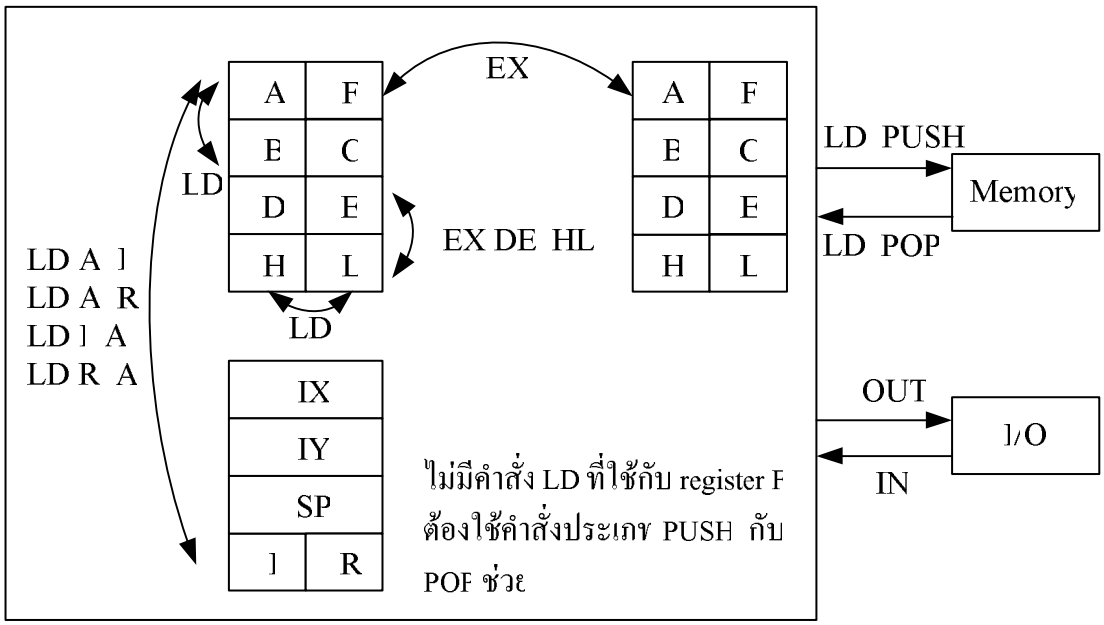
บทที่ 4 ชุดคำสั่งของ Z80

ชุดคำสั่ง (Instruction Set) ของ Z80 แบ่งออกเป็น 6 ประเภท ดังนี้

1. data transfer ⇒ การโอนย้ายข้อมูล
2. arithmetic operations ⇒ การคำนวณผลทางคณิตศาสตร์
3. logic operations ⇒ การคำนวณผลทางตรรกศาสตร์
4. bit manipulation ⇒ การจัดการข้อมูลระดับบิต
5. branch operations ⇒ การประมวลผลแบบข้ามกระโดด
6. machine control operations ⇒ การจัดการควบคุมระบบ

4.1 Data Transfer Instructions

- กลุ่ม 8-bit load
- กลุ่ม 16-bit load
- กลุ่ม exchange
- กลุ่ม block transfer
- กลุ่ม IO



LD = คำสั่งประเภท load มีแบบ 8 bits และ 16 bits

EX = คำสั่งประเภท exchange

รูปที่ 4.1ชุดคำสั่งกลุ่ม load และ exchange

หมายเหตุ: คำสั่ง data transfer ไม่มีผลต่อ flags ต่างๆ ยกเว้นบางคำสั่งในกลุ่ม I/O

กลุ่ม 8-bit load

LD r_d, { r_s / n / (HL) / (IX + d) / (IY + d) }

การทำงาน: $r_d \leftarrow r_s$; $r \leftarrow n$; $r \leftarrow (HL)$; $r \leftarrow (IX + d)$; $r \leftarrow (IY + d)$

หรือ

LD { (HL) / (IX + d) / (IY + d) }, r

การทำงาน: (HL) $\leftarrow r$; (IX + d) $\leftarrow r$; (IX + d) $\leftarrow r$

โดยที่ r, r_d, r_s = A, B, C, D, E, H,หรือ L (7 ตัวเท่านั้น ไม่มี F)

n = ข้อมูลขนาด 8 บิต

d = ค่า displacement ขนาด 1 byte

ตัวอย่าง

ถ้าขณะนี้ registers A B C D HL IX และ IY เก็บค่า 10_H 20_H 30_H 40_H $4F4F_H$ 8080_H และ 8120_H ตามลำดับ

LD A, B เป็นคำสั่งที่ทำให้ accumulator มีค่าเท่ากับ 20_H (ค่าของ register B)

LD D, 90H เป็นคำสั่งที่ทำให้ register D มีค่าเท่ากับ 90_H

LD E, (IX + 20H) เป็นคำสั่งที่ทำให้ register E มีค่าเท่ากับค่าที่เก็บในตำแหน่ง $80A0_H$ (8080_H+20_H)

LD (HL), B เป็นคำสั่งที่ทำให้ตำแหน่ง $4F4F_H$ ที่ HL ชี้อยู่มีค่าเท่ากับ 20_H (ค่าของ register B)

LD (IY+40H), C เป็นคำสั่งที่ทำให้ตำแหน่ง 8160_H (8120_H+40_H) มีค่าเท่ากับ 30_H (ค่าของ register C)

LD { r / (HL) / (IX + d) / (IY + d) }, n

การทำงาน: r $\leftarrow$ n ; (HL) $\leftarrow$ n ; (IX+d) $\leftarrow$ n ; (IY+d) $\leftarrow$ n

โดยที่ r = A, B, C, D, E, H,หรือ L (7 ตัวเท่านั้น ไม่มี F)

n = ข้อมูลขนาด 8 บิต และ d = ค่า displacement ขนาด 1 byte

ตัวอย่าง

ถ้าขณะนี้ registers HL IX และ IY เก็บค่า $4F4F_H$ 8080_H และ 8120_H ตามลำดับ

LD (HL), 50H เป็นคำสั่งที่ทำให้ตำแหน่ง $4F4F_H$ ที่ HL ชี้อยู่มีค่าเท่ากับค่าคงที่ 50_H

LD (IX + 20H), 60H เป็นคำสั่งที่ทำให้ตำแหน่ง $80A0_H$ (8080_H + 20_H) มีค่าเท่ากับค่าคงที่ 60_H

LD (IY+40H), 70H เป็นคำสั่งที่ทำให้ตำแหน่ง 8160_H (8120_H + 40_H) มีค่าเท่ากับค่าคงที่ 70_H

LD A, { (BC) / (DE) / (HL) / (nn) }

การทำงาน: A ← (BC); A ← (DE); A ←(HL); A ← (nn)

หรือกลับกัน

โดยที่ nn = ข้อมูลขนาด 16 บิต

ตัวอย่าง

ถ้าขณะนี้ registers BC DE และ HL เก็บค่า 8000_H 8001_H และ 8002_H ตามลำดับ โดยที่ค่าข้อมูล ณ ตำแหน่งเหล่านั้น คือ 10_H 20_H และ 30_H ตามลำดับ

LD A, (BC) เป็นคำสั่งที่ทำให้ accumulator มีค่าเท่ากับ 10_H ซึ่งอยู่ ณ ตำแหน่ง 8000_H ที่ BC ชี้อยู่

LD A, (DE) เป็นคำสั่งที่ทำให้ accumulator มีค่าเท่ากับ 20_H ซึ่งอยู่ ณ ตำแหน่ง 8001_H ที่ DE ชี้อยู่

LD A, (HL) เป็นคำสั่งที่ทำให้ accumulator มีค่าเท่ากับ 30_H ซึ่งอยู่ ณ ตำแหน่ง 8002_H ที่ HL ชี้อยู่

LD A, { I / R }

การทำงาน: A ← I; A ← R

หรือกลับกัน

ตัวอย่าง

ถ้าขณะนี้ registers A, I และ R เก็บค่า 10_H, 20_H และ 30_H ตามลำดับ

LD R, A เป็นคำสั่งที่ทำให้ memory refresh register มีค่าเท่ากับ 30_H (ค่าของ accumulator)

LD A, I เป็นคำสั่งที่ทำให้ accumulator มีค่าเท่ากับ 20_H (ค่าของ interrupt vector register)

กลุ่ม 16-bit load

LD { rr / IX / IY }, nn

การทำงาน: rr ← nn; IX ← nn; IY ← nn

โดยที่ rr = BC, DE, HL, SP

nn = ข้อมูลขนาด 16 บิต

ตัวอย่าง

ไม่ว่าขณะนี้ registers BC IX และ IY จะเก็บค่าอะไร

LD BC, 8050H เป็นคำสั่งที่ทำให้ register คู่ BC มีค่าเท่ากับค่าคงที่ 8050_H

LD IX, 8060H เป็นคำสั่งที่ทำให้ indexed register คู่ IX มีค่าเท่ากับค่าคงที่ 8060_H

LD IY, 8070H เป็นคำสั่งที่ทำให้ indexed register คู่ IY มีค่าเท่ากับค่าคงที่ 8070_H

LD { rr, IX, IY } , (nn) *การทำงาน*: rr ← (nn)

IX ← (nn)

IY ← (nn)

หรือกลับกัน *โดยที่ rr = BC, DE, HL, SP (ถ้าใช้ HL จะใช้ 3 bytes, ตัวอื่นๆ ใช้ 4 bytes)*

nn = ข้อมูลขนาด 16 บิต

ตัวอย่าง

ถ้าขณะนี้ที่ตำแหน่ง 8010_H 8020_H และ 8030_H เก็บค่า 10_H 20_H และ 30_H และไม่ว่า registers DE, IX และ IY จะเก็บค่าอะไร

LD DE, (8010H) เป็นคำสั่งที่ทำให้ register คู่ DE มีค่าเท่ากับ 10_H ซึ่งอยู่ ณ ตำแหน่ง 8010_H

LD IX, 8020H เป็นคำสั่งที่ทำให้ indexed register คู่ IX มีค่าเท่ากับ 20_H ซึ่งอยู่ ณ ตำแหน่ง 8020_H

LD IY, 8030H เป็นคำสั่งที่ทำให้ indexed register คู่ IY มีค่าเท่ากับ 30_H ซึ่งอยู่ ณ ตำแหน่ง 8030_H

LD SP, { HL, IX, IY }

ตัวอย่าง

ถ้าขณะนี้ registers HL IX และ IY เก็บค่า $4F4F_H$ 8080_H และ 8120_H ตามลำดับ และไม่ว่า stack pointer (SP) จะเก็บค่าอะไร

LD SP, HL เป็นคำสั่งที่ทำให้ SP มีค่าเท่ากับ $4F4F_H$ (ค่าของ register คู่ HL)

LD SP, IX เป็นคำสั่งที่ทำให้ SP มีค่าเท่ากับ 8080_H (ค่าของ register คู่ IX)

LD SP, IY เป็นคำสั่งที่ทำให้ SP มีค่าเท่ากับ $812o_H$ (ค่าของ register คู่ IY)

PUSH { rr, IX, IY } *การทำงาน*: SP ← SP - 1

(SP) ← (opr_H)

SP ← SP - 1

(SP) ← (opr_L)

POP { rr, IX, IY } *การทำงาน*: (opr_L) ← (SP)

SP ← SP + 1

(opr_H) ← (SP)

SP ← SP + 1

โดยที่ rr = AF, BC, DE หรือ HL

opr_H / opr_L =byte บน/ล่างของ operand ของคำสั่ง PUSH และ POP

ตัวอย่าง

ถ้าขณะนี้ registers AF และ IY เก็บค่า $4F00_H$ และ 8160_H ตามลำดับ และ stack pointer (SP) มีค่า 2040_H หลังจากทำคำสั่ง

PUSH AF เป็นคำสั่งที่ทำให้ตำแหน่ง $203F_H$ เก็บค่า $4F_H$ และตำแหน่ง $203E_H$ เก็บค่า 00_H

PUSH IY เป็นคำสั่งที่ทำให้ตำแหน่ง $203D_H$ เก็บค่า 81_H และตำแหน่ง $203C_H$ เก็บค่า 60_H หลังจากนั้น SP จะมีค่าเท่ากับ $203C_H$ หลังจากนั้นเมื่อพบคำสั่ง

POP IX เป็นคำสั่งที่ทำให้ indexed register IX มีค่าเท่ากับ 8160_H

POP BC เป็นคำสั่งที่ทำให้ register BC มีค่าเท่ากับ $4F00_H$ หลังจากนั้น SP จะกลับมามีค่าเท่ากับ 2040_H

กลุ่ม exchange

EX DE, HL	*การทำงาน*:	DE ↔ HL

เป็นคำสั่งที่ใช้ในการสลับค่า register คู่ ทั้ง 2 ตัว โดยที่ register คู่ DE จะมีค่าเท่ากับค่าของ register คู่ HL โดย HL จะมีค่าเท่ากับค่าที่ HL เคยเก็บไว้

ตัวอย่าง

ถ้าขณะนี้ registers คู่ DE และ HL เก็บค่า $4F00_H$ และ 8160_H ตามลำดับ

EX DE, HL จะทำให้ DE มีค่าเท่ากับ 8160_H (ค่าของ HL) และ HLมีค่าเท่ากับ $4F00_H$ (ค่าของ DE เดิม)

EX AF, AF′	*การทำงาน*:	AF ↔ AF′

คำสั่ง EX นี้ยังเป็นคำสั่งเดียวที่ใช้ในการเข้าถึง alternative registers A′, B′, C′, D′, E′, F′, H′ และ L′ อีกด้วย

EXX	*การทำงาน*:	BC ↔ BC′
		DE ↔ DE′
		HL ↔ HL′

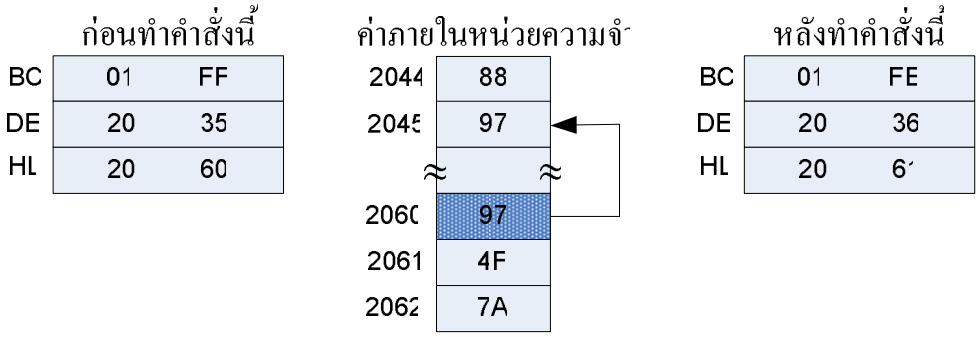
คำสั่ง EXX เป็นคำสั่งที่ใช้ในการสลับข้อมูลระหว่าง main registers และ alternative registers ทีเดียว 3 คู่

กลุ่ม block transfer

LDI	*การทำงาน*:	(DE) ← (HL)	
		DE ← DE + 1	
		HL ← HL + 1	
		BC ← BC - 1	*increment*

คำสั่ง LDI เป็นคำสั่งที่ใช้ในการโอนย้ายข้อมูลจากหน่วยความจำต้นฉบับที่ชี้โดย register HL ไปยังหน่วยความจำปลายทางที่ชี้โดย register DE แล้วเพิ่มค่าตัวชี้หน่วยความจำทั้งสอง และลดค่าตัวนับ BC

ตัวอย่าง

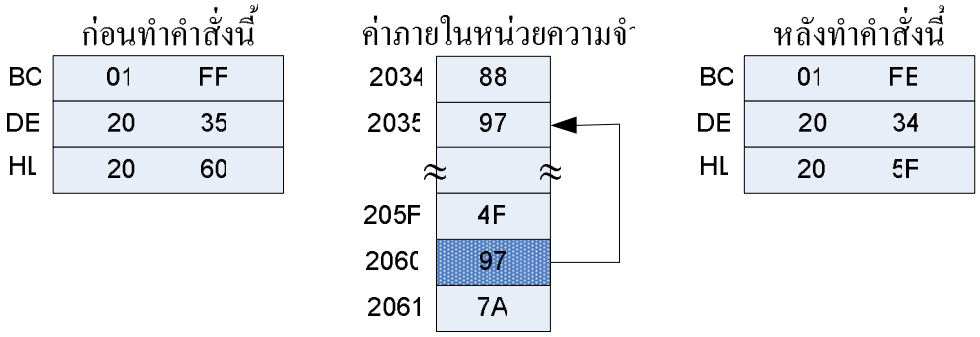


LDIR	*การทำงาน*:	ทำ LDI จนกระทั่ง BC = 0	*repeat*

LDD	*การทำงาน*:	(DE) ← (HL)	
		DE ← DE - 1	
		HL ← HL - 1	
		BC ← BC – 1	*decrement*

คำสั่ง LDD เป็นคำสั่งที่ใช้ในการโอนย้ายข้อมูลจากหน่วยความจำต้นฉบับที่ชี้โดย register HL ไปยังหน่วยความจำปลายทางที่ชี้โดย register DE แล้วลดค่าตัวชี้หน่วยความจำทั้งสอง และลดค่าตัวนับ BC

ตัวอย่าง



LDDR	*การทำงาน*:	ทำ LDD จนกระทั่ง BC = 0	*repeat*

กลุ่ม I/O

IN	A, (n)	
OUT	(n), A	*โดยที่ n เป็น 8-bit port number*

คำสั่ง IN เป็นคำสั่งที่ใช้ในการรับค่าข้อมูลจากพอร์ตหมายเลขที่ n ไปเก็บไว้ที่ accumulator

คำสั่ง OUT เป็นคำสั่งที่ใช้ในการส่งค่าข้อมูลใน accumulator ออกไปยังพอร์ตหมายเลขที่ n

IN	r, (C)	
OUT	(C), r	*โดยที่ r = A, B, C, D, E, H, L*

คำสั่ง IN เป็นคำสั่งที่ใช้ในการรับค่าข้อมูลจากพอร์ตหมายเลขที่อยู่ใน register C ไปเก็บไว้ที่ register r

คำสั่ง OUT เป็นคำสั่งที่ใช้ในการส่งค่าข้อมูลใน register r ออกไปยังพอร์ตหมายเลขที่อยู่ใน register C

INI	*การทำงาน*:	(HL) ← (C) HL ← HL + 1 B ← B - 1	*increment*

คำสั่ง INI เป็นคำสั่งที่ใช้ในการรับค่าข้อมูลจากพอร์ตหมายเลขที่อยู่ใน register C ไปเก็บไว้ที่หน่วยความจำตำแหน่งที่โดย register HL แล้วเพิ่มค่า HL และลดค่าตัวนับ B

INIR	*การทำงาน*:	ทำ INI จนกระทั่ง B = 0	*repeat*

OUTI	*การทำงาน*:	(C) ← (HL) HL ← HL + 1 B ← B - 1	*increment*

คำสั่ง OUT เป็นคำสั่งที่ใช้ในการส่งค่าข้อมูลในหน่วยความจำตำแหน่งที่โดย register HL ออกไปยังพอร์ตหมายเลขที่อยู่ใน register C แล้วเพิ่มค่า HL และลดค่าตัวนับ B

OTIR	*การทำงาน*:	ทำ OUTI จนกระทั่ง B = 0	*repeat*

IND	*การทำงาน*:	(HL) ← (C) HL ← HL - 1 B ← B - 1	*decrement*

คำสั่ง INI เป็นคำสั่งที่ใช้ในการรับค่าข้อมูลจากพอร์ตหมายเลขที่อยู่ใน register C ไปเก็บไว้ที่หน่วยความจำตำแหน่งที่โดย register HL แล้วลดค่า HL และลดค่าตัวนับ B

INDR	*การทำงาน*:	ทำ IND จนกระทั่ง B = 0	*repeat*

OUTD	*การทำงาน*:	(C) ← (HL) HL ← HL - 1 B ← B - 1	*decrement*

คำสั่ง OUT เป็นคำสั่งที่ใช้ในการส่งค่าข้อมูลในหน่วยความจำตำแหน่งที่โดย register HL ออกไปยังพอร์ตหมายเลขที่อยู่ใน register C แล้วลดค่า HL และลดค่าตัวนับ B

OTDR	*การทำงาน*:	ทำ OUTD จนกระทั่ง B = 0	*repeat*

4.2 Arithmetic Instructions

- กลุ่ม 8-bit addition
- กลุ่ม 8-bit subtraction
- กลุ่ม compare

- กลุ่ม increment และ decrement แบบ 8-bit
- กลุ่มที่มี operand เพียงตัวเดียว (คือ accumulator)
- กลุ่ม 16-bit addition
- กลุ่ม 16-bit subtraction
- กลุ่ม increment และ decrement แบบ 16-bit

<u>กลุ่ม 8-bit addition</u>

ADD A, { r, n, (HL), (IX + d), (IY + d) } *<u>การทำงาน</u>*: A ← A + { r, n, (HL), (IX + d), (IY + d) }

คำสั่ง ADD เป็นคำสั่งที่ใช้ในการบวกข้อมูลขนาด 8 บิตจากแหล่งต่างๆ กับ accumulator แล้วเก็บผลลัพธ์ที่ accumulator โดยแหล่งสำหรับข้อมูล 8 บิต นั้นได้แก่

r	หมายถึง	Register ต่างๆ
n	หมายถึง	ข้อมูลขนาด 8 บิตจริงๆ
(HL)	หมายถึง	ค่าในหน่วยความจำที่ชี้โดย register HL
(IX+d) หรือ (IY+d)	หมายถึง	ค่าในหน่วยความจำที่ชี้โดย index register IX หรือ IY บวกกับค่าระยะห่าง *d* bytes

ADC A, { r, n, (HL), (IX + d), (IY + d) } *<u>การทำงาน</u>*: A ← A + { r, n, (HL), (IX + d), (IY + d) } + CY

คำสั่ง ADC เป็นคำสั่งที่ใช้ในการบวกข้อมูลขนาด 8 บิตจากแหล่งต่างๆ กับค่าตัวทดใน carry flag และ accumulator แล้วเก็บผลลัพธ์ที่ accumulator

<u>กลุ่ม 8-bit subtraction</u>

SUB A, { r, n, (HL), (IX + d), (IY + d) } *<u>การทำงาน</u>*: A ← A - { r, n, (HL), (IX + d), (IY + d) }

คำสั่ง SUB เป็นคำสั่งที่ใช้ในการลบค่า accumulator ด้วยข้อมูลขนาด 8 บิตจากแหล่งต่างๆ แล้วเก็บผลลัพธ์ที่ accumulator โดยแหล่งสำหรับข้อมูล 8 บิตนั้นได้แก่ r n (HL) (IX+d) หรือ (IY+d)

SBC	A, { r, n, (HL), (IX + d), (IY + d) }	*การทำงาน*:	A ← A - { r, n, (HL), (IX + d), (IY + d) } - CY

คำสั่ง SBC เป็นคำสั่งที่ใช้ในการลบค่า accumulator ด้วยข้อมูลขนาด 8 บิตจากแหล่งต่างๆ แล้วเก็บผลลัพธ์ที่ accumulator

กลุ่ม compare

CP	{ r, n, (HL), (IX + d), (IY + d) }	*การทำงาน*:	A - { r, n, (HL), (IX + d), (IY + d) }

คำสั่ง CP เป็นคำสั่งที่ใช้ในการเปรียบเทียบค่าใน accumulator กับค่าต่างๆ โดยไม่มีการเปลี่ยนแปลงค่าเหล่านี้ แต่จะมีการเปลี่ยนแปลง flags ต่างๆ ดังนี้

	Carry flag	Zero flag
ถ้า A มีค่าน้อยกว่า	Set (1)	Reset (0)
ถ้ามีค่าเท่ากัน	Reset (0)	Set (1)
ถ้า A มีค่ามากกว่า	Reset (0)	Reset (0)

CPI	*การทำงาน*:	A - (HL)	
		HL ← HL + 1	
		BC ← BC - 1	*increment*

คำสั่ง CPI เป็นคำสั่งที่ใช้ในการเปรียบเทียบค่าใน accumulator กับค่าในหน่วยความจำที่ชี้โดย register HL แล้วเพิ่มค่า HL และลดค่าตัวนับ BC

CPIR	*การทำงาน*:	ทำ CPI จนกระทั่ง A = (HL) หรือ BC = 0	*repeat*

CPD	*การทำงาน*:	A - (HL)	
		HL ← HL - 1	
		BC ← BC - 1	*decrement*

คำสั่ง CPI เป็นคำสั่งที่ใช้ในการเปรียบเทียบค่าใน accumulator กับค่าในหน่วยความจำที่ชี้โดย register HL แล้วลดค่า HL และลดค่าตัวนับ BC

CPDR	*การทำงาน*:	ทำ CPD จนกระทั่ง A = (HL) หรือ BC = 0	*repeat*

กลุ่ม increment และ decrement แบบ 8-bit

INC { r, (HL), (IX + d), (IY + d) } หรือ DEC { r, (HL), (IX + d), (IY + d) } *มีผลต่อทุก flags ยกเว้น CY*

คำสั่ง INC/DEC เป็นคำสั่งที่ใช้ในการเพิ่ม/ลดค่าต่างๆ 1 ค่า

กลุ่มที่มี operand เพียงตัวเดียว (คือ accumulator)

DAA - decimal-adjust accumulator

CPL - 1's complement

NEG - 2's complement

SCF - set carry flag

CCF - Complement carry flag

กลุ่ม 16-bit addition

ADD HL, { BC, DE, HL, SP } หรือ ADD IX, { BC, DE, IX, SP } หรือ ADD IX, { BC, DE, IX, SP }

มีผลเฉพาะ N และ CY flags นั้น, Flag H จะ undefined

ADC HL, { BC, DE, HL, SP }

มีผลต่อทุก flags, Flag H จะ undefined

คำสั่งในกลุ่มนี้จะต่างจากคำสั่งสำหรับการบวกก่อนหน้านี้ คือเป็นการบวกค่าขนาด 16 บิต

กลุ่ม 16-bit subtraction

SBC HL, { BC, DE, HL, SP } *การทำงาน*: HL ← HL - { BC, DE, HL } - CY

SP*มีผลต่อทุก flags, Flag H จะ undefined*

คำสั่งในกลุ่มนี้จะต่างจากคำสั่งสำหรับการลบก่อนหน้านี้ คือเป็นการลบค่าขนาด 16 บิตซึ่งมีเฉพาะ SBC ซึ่งเป็นการลบเพิ่มเติมด้วยค่า carry flag ด้วย

<u>กลุ่ม increment และ decrement แบบ 16-bit</u>

INC { BC, DE, HL, SP, IX, IY } หรือ DEC { BC, DE, HL, SP, IX, IY }

ไม่มีผลต่อ flags ใดๆเลย

คำสั่งในกลุ่มนี้จะต่างจากคำสั่งสำหรับการเพิ่มและลดค่าก่อนหน้านี้ คือเป็นการเพิ่มและลดค่าขนาด 16 บิต

4.3 Logical Instructions

AND A, { r, n, (HL), (IX + d), (IY + d) } <u>การทำงาน</u>: A ← A & { r, n, (HL), (IX + d), (IY + d) }

OR A, { r, n, (HL), (IX + d), (IY + d) } <u>การทำงาน</u>: A ← A | { r, n, (HL), (IX + d), (IY + d) }

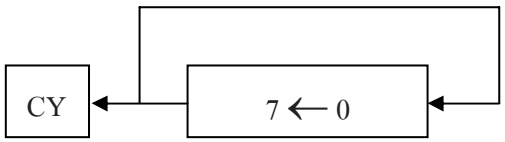
XOR A, { r, n, (HL), (IX + d), (IY + d) } <u>การทำงาน</u>: A ← A XOR { r, n, (HL), (IX + d), (IY + d) }

4.4 Bit Manipulation Instructions

- กลุ่ม rotate
- กลุ่ม shift
- กลุ่ม bit, set, reset และ test

<u>กลุ่ม rotate</u>

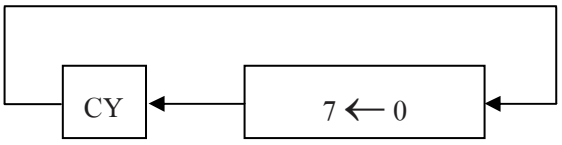
RLC { r, (HL), (IX + d), (IY + d) }



rotate left circular

RLC A, RLCA } คนละคำสั่ง แต่ทำงานเหมือนกัน โดยจะมีผลกระทบกับ flags ไม่เหมือนกัน และจำนวน byte ของคำสั่งไม่กัน

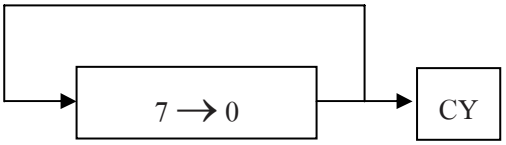
RL { r, (HL), (IX + d), (IY + d) }



rotate left

RL A, RLA } คนละคำสั่ง แต่ทำงานเหมือนกัน โดยจะมีผลกระทบกับ flags ไม่เหมือนกัน และจำนวน byte ของคำสั่งไม่กัน

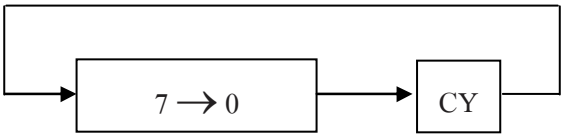
RRC { r, (HL), (IX + d), (IY + d) }



rotate right circular

RRC A, RRCA } คนละคำสั่ง แต่ทำงานเหมือนกัน โดยจะมีผลกระทบกับ flags ไม่เหมือนกัน และจำนวน byte ของคำสั่งไม่กัน

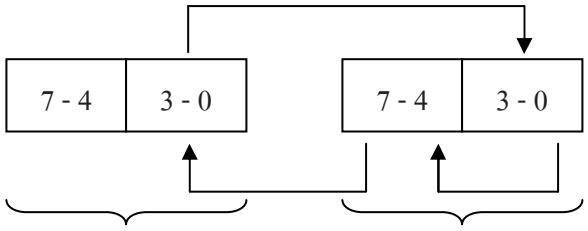
RR { r, (HL), (IX + d), (IY + d) }



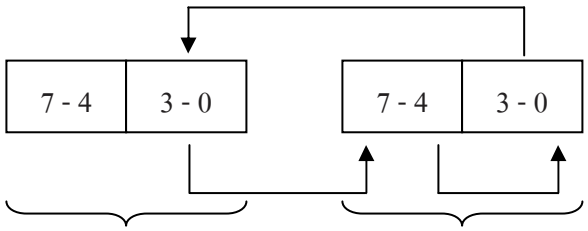
rotate right

RR A, RRA } คนละคำสั่ง แต่ทำงานเหมือนกัน โดยจะมีผลกระทบกับ flags ไม่เหมือนกัน และจำนวน byte ของคำสั่งไม่กัน

RLD ใช้กับตัวเลข BCD

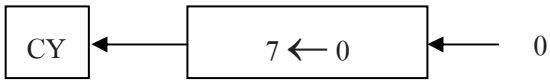


RRD ใช้กับตัวเลข BCD



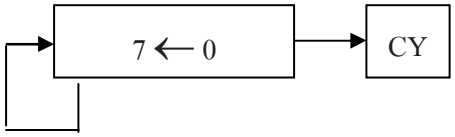
<u>กลุ่ม shift</u>

SLA { r, (HL), (IX + d), (IY + d) }



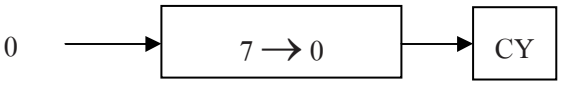
shift left arithmetic

SRA { r, (HL), (IX + d), (IY + d) }



shift right arithmetic

SRL { r, (HL), (IX + d), (IY + d) }



shift right logical

<u>กลุ่ม bit, set, reset และ test</u>

BIT b, { r, (HL), (IX + d), (IY + d) } หรือ SET b, { r, (HL), (IX + d), (IY + d) } หรือ RES b, { r, (HL), (IX + d), (IY + d) }

r = A, B, C, D, E, H และ L โดยที่ b คือตำแหน่ง bit 0 → 7

4.5 Branch Instructions

- กลุ่ม unconditional jump
- กลุ่ม conditional jump
- กลุ่ม call และ return

<u>กลุ่ม unconditional jump</u>

JP	nn	<u>การทำงาน</u>:	PC ← nn	*absolute jump* *nn = ตำแหน่ง 16 บิต*
JR	e	<u>การทำงาน</u>:	PC ← PC + e	*relative jump* *e = เลข 8 บิต (-128 → +127)*

JP	(HL) (IX) (IY)	<u>การทำงาน</u>:	PC ← HL IX IY

<u>กลุ่ม conditional jump</u>

JP	cc, nn			*conditional absolute jump* *cc = NZ, Z, NC, C, PO, PE, P, M*
JR	cc, nn			*conditional relative jump* *cc = NZ, Z, C*
DJNZ	e	<u>การทำงาน</u>:	B ← B -1 If B = 0 continue Else (B ≠ 0) PC ← PC + e	*decrease and jump if not zero*

<u>กลุ่ม call และ return</u>

CALL	nn	<u>การทำงาน</u>:	(SP - 1) ← PC_H (SP - 2) ← PC_L SP ← SP - 2 PC ← nn	
CALL	cc, nn			*conditional call*

RET	<u>การทำงาน</u>:	$PC_L \leftarrow (SP)$ $PC_H \leftarrow (SP+1)$ $SP \leftarrow SP + 2$	
RET cc			*conditional return*
RETI			*return from interrupt* เหมือนกับ RET แต่ peripheral chip บางตัวจะถูกสร้างให้รู้จัก RETI
RETN			*return from non-maskable interrupt*
RST p	<u>การทำงาน</u>:	$(SP - 1) \leftarrow PC_H$ $(SP - 2) \leftarrow PC_L$ $PC_H \leftarrow 0$ $PC_L \leftarrow p$	*restart* เหมือนกับ CALL เพียงแต่ high address เป็น 00_H และ low address มี 8 ตำแหน่ง

4.6 Machine Control Operations Instructions

HALT	<u>การทำงาน</u>: พักการทำงานชั่วคราว
NOP	<u>การทำงาน</u>: ไม่มีการปฏิบัติงาน (No Operation) โดยไม่มีผลต่อ flags ใดๆ

4.7 Assembler Directives

นอกเหนือจากคำสั่งทั้ง 6 ประเภทที่ได้อธิบายผ่านมานั้น ในการเขียนโปรแกรมภาษาแอสแซมบลีเพื่อกำหนดการทำงานไมโครโพรเซสเซอร์ Z80 ยังมี Assembler Directives ที่เกี่ยวข้องดังนี้

1. ค่าต่อท้ายคงที่ (Constant Suffix)

เป็นตัวอักษร (D B Q หรือ H) ที่ใช้กำหนดเลขฐานต่าง ๆ ของข้อมูล โดยที่

D แสดงว่าข้อมูลเป็นเลขฐานสิบ (โดยอาจละไว้ในฐานที่เข้าใจได้)

B แสดงว่าข้อมูลเป็นเลขฐานสอง

Q แสดงว่าข้อมูลเป็นเลขฐานแปด

H แสดงว่าข้อมูลเป็นเลขฐานสิบหก โดยที่ตัวเลขที่เริ่มต้นด้วย A ถึง F ต้องมี 0 นำหน้าเสมอ

<u>*ตัวอย่าง*</u>

97, 0A1H, 1011B หมายถึง $97\ A1_H\ 1011_2$ ตามลำดับ

2. คำสั่ง ORG (Origin)

เป็นคำสั่งที่ใช้กำหนดตำแหน่งเริ่มต้นในหน่วยความจำของโปรแกรม

ตัวอย่าง

ORG 2000H เป็นคำสั่งที่แสดงว่ากลุ่มคำวั่งต่อจากนี้จะเริ่มเก็บที่ตำแหน่ง 2000_H ในหน่วยความจำ

3. คำสั่ง END

เป็นคำสั่งของ assembler ที่แสดงจุดสิ้นสุดของโปรแกรม โดยที่คำสั่ง HALT ของไมโครโพรเซสเซอร์ แสดงการสิ้นสุดการทำงานของโปรแกรม แต่ไม่ได้หมายถึงการสิ้นสุดการทำงานของแอสแซมบลี

4. คำสั่ง EQU (Equate)

เป็นคำสั่งที่ใช้กำหนดค่าคงที่ให้แก่ตัวแปรต่างๆ

ตัวอย่าง

PORTA EQU 0B3H ตัวแปรที่ชื่อ PORTA มีค่าเท่ากับ $B3_H$

5. คำสั่ง DB (Define Byte)

เป็นคำสั่งที่ใช้กำหนดค่าต่าง ๆ ลงในหน่วยความจำครั้งละ 1 byte

ตัวอย่าง

DATA: DB 00H,01H,10H ที่ตำแหน่ง DATA เก็บข้อมูล 00_H ตามด้วย 01_H และ 10_H

6. คำสั่ง DW (Define Word)

เป็นคำสั่งที่ใช้กำหนดค่าขนาด 1 word หรือ 2 bytes ลงในหน่วยความจำ

ตัวอย่าง

ADDR: DW 2050H ที่ตำแหน่ง ADDR เก็บข้อมูล 2050_H

7. คำสั่ง DFS (Define Storage)

เป็นคำสั่งที่ใช้จองจำนวนตำแหน่งหน่วยความจำที่กำหนด โดยตัวเลขที่ตามมานั้นมีหน่วยเป็น byte

ตัวอย่าง

DAT: DFS 4 ที่ตำแหน่ง DAT ได้จองหน่วยความจำขนาด 4 bytes

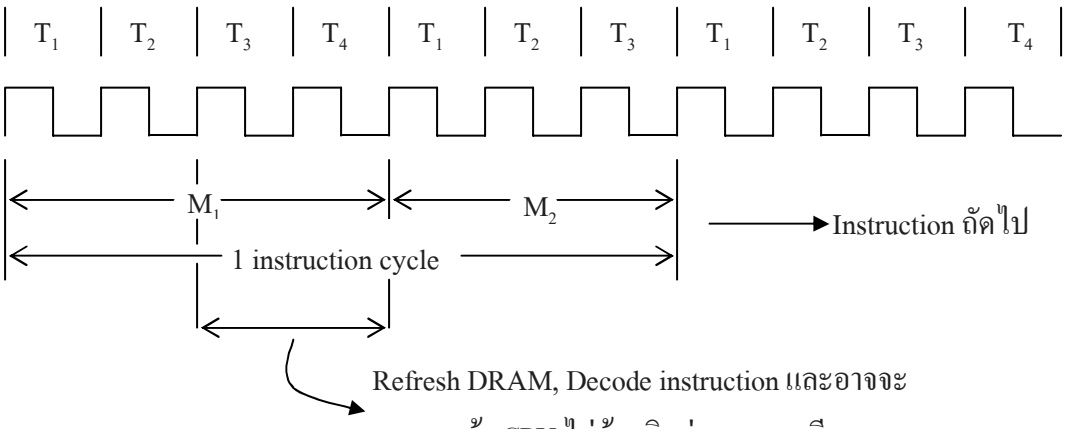
บทที่ 5 Z80 Machine Cycles and Bus Timing

การทำงานของ CPU ในแต่ละคำสั่ง (instruction) จะประกอบไปด้วยกิจกรรม 3 อย่าง คือ

1. **Fetch** อ่านคำสั่ง
2. **Decode** แปลคำสั่ง
3. **Execute** ทำงานตามคำสั่ง ซึ่งอาจจะเป็น
   - CPU ไม่มีการติดต่อกับภายนอก
   - CPU ↔ Memory (memory read/write)
   - CPU ↔ I/O (I/O read/write)
   - CPU ตอบรับการขอจากภายนอก (external request acknowledge)

เมื่อทำงานครบทั้ง 3 อย่างนี้ จะเรียกว่า 1 **instruction cycle** ในแต่ละคำสั่งจะแบ่งการทำงานย่อย ๆ ลงไปอีก เรียกว่า **machine cycle** ได้แก่ การติดต่อกับ memory หรือ I/O หรือ การตอบรับ (acknowledge) หรือ การขอร้องจากภายนอก (external request)

$\overline{M1}$ หมายถึง machine cycle แรก ซึ่งต้องเป็นการ fetch opcode เสมอ ในช่วงปลายของ $\overline{M1}$ จะมีการ refresh DRAM, decode สัญญาณ และอาจมีการ execute ถ้า CPU ไม่ต้องติดต่อกับภายนอกอีก



รูปที่ 5.1 Z80 Machine Cycles และ Timings

T-states หมายถึง คาบเวลาของนาฬิกา หรือ clock periods นั่นเอง

"1 instruction cycle" ประกอบด้วย 1 – 6 machine cycles

"1 machine cycle" ประกอบด้วย 3 – 6 T–states ขึ้นอยู่กับชนิดของ machine cycle

ดังนั้น instruction ที่สั้นที่สุดจะใช้ 1 machine cycle คือ M_1 ซึ่งจะมี 4 T-states

ในบทนี้เราจะมุ่งความสนใจไปที่การทำงาน 3 อย่างแรกจากตารางที่ 5.1 คือ opcode fetch, memory read และ memory write

Machine Cycle	$\overline{M1}$	$\overline{MREQ}$	$\overline{IORQ}$	$\overline{RD}$	$\overline{WR}$
Opcode Fetch ($\overline{M1}$)	0	0	1	0	1
Memory Read	1	0	1	0	1
Memory Write	1	0	1	1	0
I/O Read	1	1	0	0	1
I/O Write	1	1	0	1	0
Interrupt Acknowledge	0	1	0	1	1
Nonmaskable Interrupt	0	0	1	0	1
Bus Acknowledge ($\overline{BUSAK} = 0$)	1	Z	Z	Z	Z

หมายเหตุ: Logic 0 = Active, Logic 1 = Inactive, Z = High Impedance

ตารางที่ 5.1 ชนิดของ Machine Cycles และ Control Signals ในไมโครโพรเซสเซอร์ Z80

5.1 Opcode Fetch Machine Cycle

การทำงานลำดับแรกในคำสั่งใดๆคือ การที่ไมโครโพรเซสเซอร์อ่าน (fetch) คำสั่งเครื่อง (machine code) ซึ่งในที่นี้ก็คือค่า opcode จาก memory ตัวอย่างที่ 5.1 อธิบายให้เห็นถึงการทำงานของ opcode fetch รวมไปถึง timing signals ที่ใช้ใน machine cycle นี้ด้วย

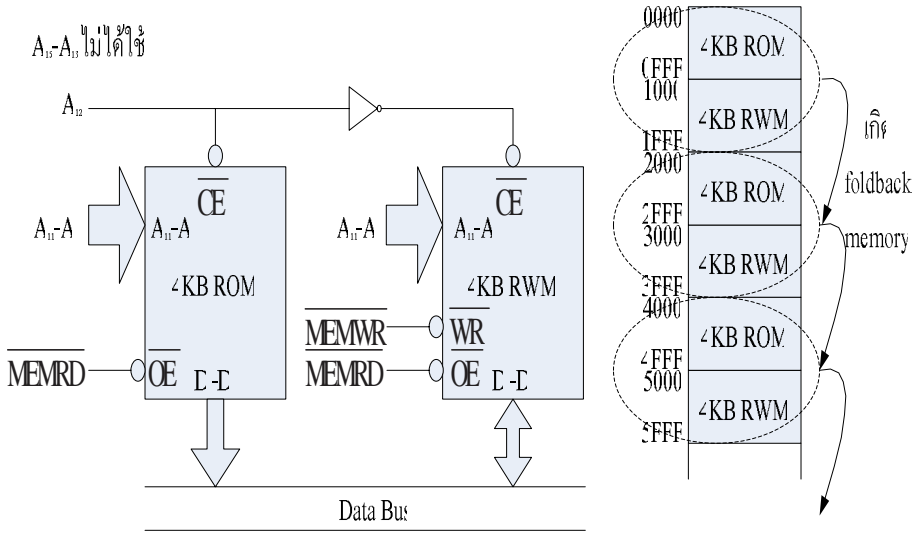
ตัวอย่างที่ 5.1 Accumulator ของไมโครโพรเซสเซอร์ Z80 มีข้อมูล $9F_H$ อยู่ และ opcode 0100 0111 (47_H) ของคำสั่ง LD B, A ถูกเก็บไว้ใน memory ตำแหน่ง 2002_H คำสั่งนี้เป็นคำสั่งขนาด 1 byte เมื่อถูก execute ข้อมูลที่อยู่ใน accumulator จะถูก copy ไปไว้ที่ register B ด้วย จงหาลำดับของเหตุการณ์ต่างๆ ที่จะเกิดขึ้นเพื่อที่จะ execute คำสั่งเครื่องคำสั่งนี้ พร้อมทั้งแสดงรายละเอียดของสัญญาณบน bus ต่าง ๆ ที่ขึ้นอยู่กับ system clock

คำตอบ

สมมติว่า Z80 ได้เสร็จสิ้นการ execute คำสั่งใน memory ตำแหน่งที่ 2001_H และขณะนี้ PC มีค่าเท่ากับ 2002_H หลังจากนั้นขั้นตอนของกิจกรรมต่าง ๆ ที่ Z80 ต้องทำ คือ

1. Z80 นำข้อมูลจาก PC (2002_H) ไปไว้ที่ address bus แล้วทำการเพิ่มค่าของ PC ไปเป็นตำแหน่งถัดไป ซึ่งในที่นี้คือ 2003_H เนื่องจาก PC จะชี้ไปยังคำสั่งถัดไปที่จะถูก execute เสมอ
2. Z80 ส่งสัญญาณควบคุม $\overline{MREQ}$ และ $\overline{RD}$ ไปเพื่อ enable memory output buffer (เพื่อบอกว่านี่เป็นการขอร้องใช้ memory เพื่อการอ่าน)
3. ข้อมูล (opcode 47_H) จาก memory ตำแหน่ง 2002_H จะไปอยู่ที่ data bus และจะถูกนำเข้ามาที่ instruction decoder ของไมโครโพรเซสเซอร์

4. Z80 จะ decode ค่า opcode และทำการ execute คำสั่งนั้น ซึ่งจะหมายถึงการ copy ข้อมูลจาก accumulator ไปยัง register B

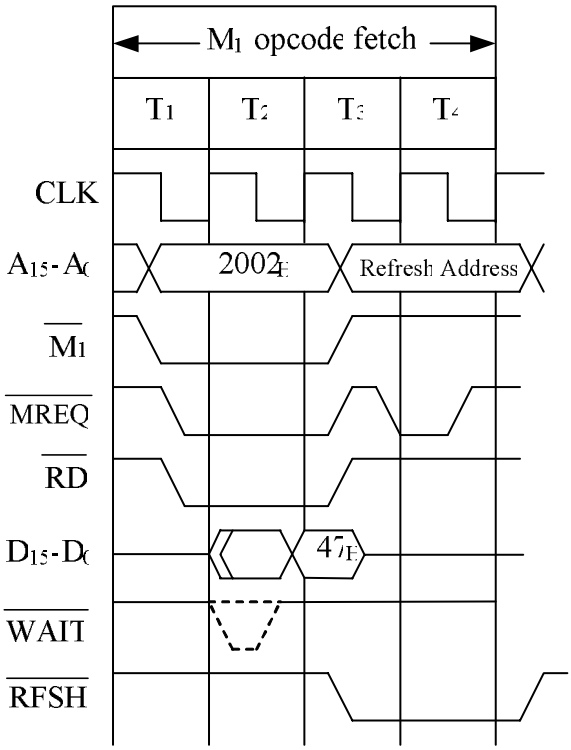


รูปที่ 5.2 Z80 Machine Cycles และ Timings

รูปที่ 5.2 แสดงให้เห็นถึงการทำงานของ Z80 เมื่อทำการ fetch ค่า opcode โดยอาศัย address bus และ data bus ร่วมด้วยสัญญาณควบคุม

รูปที่ 5.3 แสดง timings ของ Opcode fetch machine cycle โดยเป็นสัดส่วนกับ system clock สังเกตว่า address bus จะถูกแสดงโดยใช้เส้นคู่ขนานซึ่งเป็นวิธีที่ใช้บ่อยในการแสดงระดับของ logic ของกลุ่มของข้อมูล (บ้างก็ '0' หรือ low บ้างก็ '1' หรือ high) และการตัดกันของเส้นทั้ง 2 หมายถึงการเปลี่ยนแปลงของข้อมูล ซึ่งก็คือเมื่อ address ใหม่เข้ามาที่ address bus ส่วนสถานะที่เป็น high impedance จะแสดงในลักษณะของเส้นตรงที่กึ่งกลาง (ไม่เป็น '0' หรือ '1') ดังเช่นในบางส่วนของ data bus รายละเอียดต่าง ๆ เกี่ยวกับ timings ใน Opcode fetch machine cycle มีดังนี้

1. จากรูปที่ 5.3 แสดงให้เห็นว่า Opcode fetch cycle เสร็จสมบูรณ์ใน 4 clock periods หรือ T-states โดยที่ Machine cycle นี้ ถูกนิยามว่า M_1 cycle
2. ในช่วงเริ่มต้นของ clock period แรก (T_1) สัญญาณควบคุม $\overline{M_1}$ จะเป็น '0' (active low) และ ค่าของ PC (2002_H) จะไปปรากฏที่ address bus
3. หลังจากขาลง (falling-edge) ของ T_1 Z80 จะสั่งให้สัญญาณควบคุมที่เหมาะสม ($\overline{MREQ}$ และ $\overline{RD}$) ทำจน (ทั้งคู่ active low)



รูปที่ 5.3 Z80 Opcode Fetch Machine Cycle (M_1) และ Bus Timings

4. ในส่วนของ memory ซึ่งไม่ได้ถูกแสดงในรูปที่ 5.3 นี้ จะทำการ decode ตำแหน่งจาก address bus เพื่อให้ได้ค่า 2002_H ซึ่งจะเป็นตำแหน่งใน memory ส่วนสัญญาณควบคุมทั้งสอง ($\overline{MREQ}$ และ $\overline{RD}$) จะไป enable memory output buffer เพื่อไปส่งให้ data bus ซึ่งเคยมีสถานะเป็น high impedance ทำงานเป็น input bus ในช่วงต้นของ T_2 เพื่อส่งค่าเข้าไปยังไมโครโพรเซสเซอร์หลังจากขาลงของ T_2 memory จะนำค่า 47_H ที่อ่านได้จากตำแหน่ง 2002_H ไปไว้ที่ data bus
5. หลังจากขาขึ้นของ T_3 แล้วไมโครโพรเซสเซอร์จะอ่านข้อมูลจาก data bus และทำการ inactive สัญญาณควบคุมต่างๆ ($\overline{M_1}$, $\overline{MREQ}$ และ $\overline{RD}$)
6. ในช่วงของ T_3 และ T_4 instruction decoder ในไมโครโพรเซสเซอร์จะ decode และ execute ตามค่า opcode ที่อ่านมาได้ สิ่งเหล่านี้คือการทำงานภายใน ไมโครโพรเซสเซอร์

ส่วนอีก 2 ขั้นตอนข้างล่างนี้ ถึงแม้จะไม่มีส่วนเกี่ยวข้องกับปัญหาในตัวอย่างนี้ อย่างไรก็ตามขั้นตอนเหล่านี้ก็ถูกรวมอยู่ที่นี่ด้วยในฐานะที่เป็นส่วนหนึ่งของ M_1 cycle

1. ในช่วง T_3 และ T_4 ในขณะที่ Z80 ปฏิบัติงานภายใน low-order ของ address bus จะถูกใช้ในการจ่าย address ขนาด 7 บิต เพื่อที่จะ refresh ตัว dynamic memory ซึ่งถ้าระบบมี memory ประเภทนี้อยู่ ขั้นตอนนี้จะทำให้การการต่อประสานกับฮาร์ดแวร์เป็นไปได้ง่ายขึ้น
2. ในรูปที่ 5.3 นี้มีการแสดงสัญญาณ $\overline{WAIT}$ ซึ่งจะถูก sample โดย Z80 ในช่วง T_2 ถ้าสัญญาณ $\overline{WAIT}$ นี้ active คือถูกทำให้เป็น low โดยอุปกรณ์ภายนอกหนึ่ง ๆ (เช่น memory หรือ I/O) Z80 จะเพิ่ม wait states

(clock cycles) เพื่อที่จะขยาย machine cycle ออกไป จนกระทั่ง สัญญาณ $\overline{WAIT}$ กลับมาเป็น high อีกครั้ง เทคนิคนี้จะใช้เมื่ออุปกรณ์ภายนอกมีการตอบสนองที่ช้ามาก ๆ

5.2 Memory Read Machine Cycle

Machine cycle ต่อมาที่จะกล่าวถึง คือ Memory read machine cycle ซึ่งจะมีความคล้ายกับ Opcode fetch machine cycle ที่ได้กล่าวมาข้างต้นแล้ว โดยที่ Opcode fetch cycle นั้นถือว่าเป็น Memory read machine cycle ที่มี $\overline{M_1}$ active ด้วย

ตัวอย่างที่ 5.2 คำสั่งเครื่อง (machine codes) 2 คำสั่ง 0011 1110 ($3E_H$) และ 1001 1110 ($9E_H$) ถูกเก็บไว้ที่ memory ตำแหน่ง 2000_H และ 2001_H ตามลำดับ โดยคำสั่งเครื่องคำสั่งแรก ($3E_H$) เป็นคำสั่งที่จะ load ข้อมูลขนาด 1 byte ไปไว้ที่ accumulator โดยคำสั่งเครื่องคำสั่งที่สอง ($9F_H$) จะเป็นข้อมูลที่จะถูก load ไปที่ accumulator จงหา bus timings เมื่อคำสั่งเครื่อง 2 คำสั่งนี้ถูก execute และจงหาระยะเวลาที่ใช้ในการ execute ในส่วนของ Opcode fetch machine cycle. Memory read cycle และ ในการ execute ทั้ง instruction เมื่อ clock frequency มีค่าเท่ากับ 4 MHz.

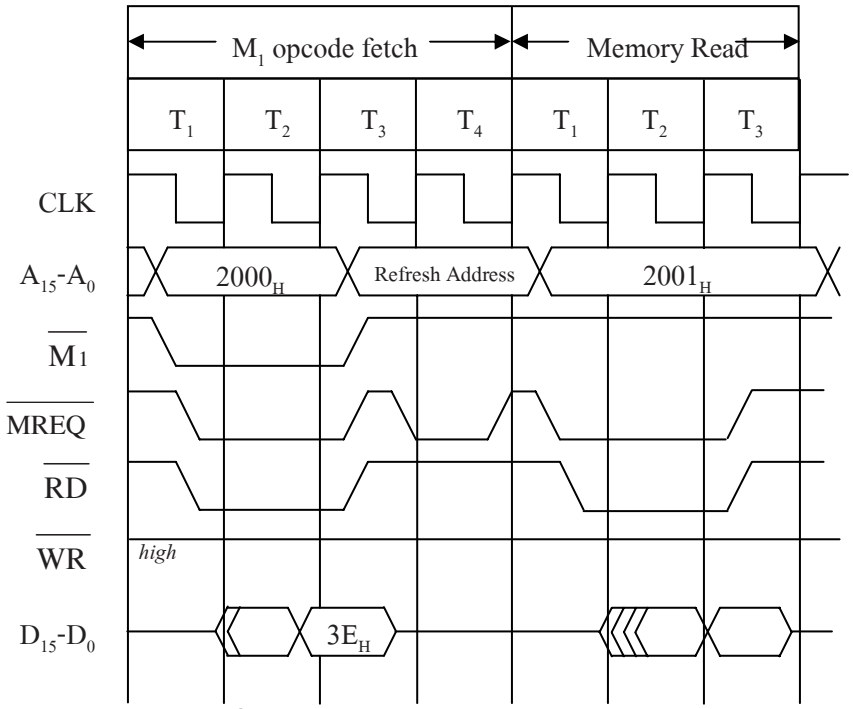
Address	**Machine code**	**Instruction**	**Comment**
2000_H	0 0 1 1 1 1 1 0 → 3E	LD A, 9FH	;Load 9FH in the accumulator
2001_H	1 0 0 1 1 1 1 1 → 9F		

คำตอบ

คำสั่งนี้ประกอบด้วย 2 bytes - byte แรก คือ opcode และ byte ที่สอง คือ ข้อมูลขนาด 1 byte เริ่มต้นด้วยการที่ Z80 จะต้องทำการอ่านข้อมูลทั้ง 2 bytes นี้จาก memory ซึ่งจะต้องใช้อย่างน้อย 2 machine cycles: opcode fetch และ memory read

รูปที่ 5.4 แสดง timings ของ Memory read machine cycle โดยมีรายละเอียดต่างๆ ดังนี้

1. Machine cycle แรก (opcode fetch) จะเหมือนกันในเชิง bus timings กับ machine cycle ใน ตัวอย่างที่ 5.1 ยกเว้นค่าที่อยู่ใน bus ต่าง ๆ address bus เก็บ 2000_H data bus เก็บค่า opcode $3E_H$ หลังจากที่ Z80 ได้ decode ค่า opcode ในช่วง T_3 ไมโครโพรเซสเซอร์จะรู้ว่าต้องมีการอ่าน byte ที่ 2 ด้วย
2. หลังจากเสร็จสิ้น Opcode fetch cycle แล้ว Z80 จะวางค่าตำแหน่ง 2001_H ไว้ที่ address bus แล้ว ทำการเพิ่มค่าของ PC ไปเป็นตำแหน่งถัดไป (2002_H) สิ่งที่ทำให้ Machine cycle นี้ (Memory read cycle) แตกต่างจาก Opcode fetch cycle คือ $\overline{M_1}$ ยังคงเป็น high อยู่ คือไม่ active



รูปที่ 5.4 Z80 Memory Read Machine Cycle และ Bus Timings

3. หลังจากขาลงของ T_1 ใน Memory read cycle, สัญญาณควบคุม $\overline{MREQ}$ และ $\overline{RD}$ จะ active โดยสัญญาณเหล่านี้พร้อมกับตำแหน่งของ memory จะเป็นการไป enable ตัว memory chip

4. หลังจากขาขึ้นของ T_3 ใน Memory read cycle, Z80 จะ active ส่วนของ data bus ให้ทำหน้าที่เป็น input bus เพื่อรองรับค่า $9F_H$ ที่ memory ส่งมาให้ หลังจากนั้น Z80 จะอ่านข้อมูลดังกล่าวแล้วเก็บไว้ใน accumulator ในช่วง T_3 นี้

5. หลังจากขาลงของ T_3 ใน Memory read cycle, สัญญาณควบคุมทั้งสองจะกลับไปเป็น high คือไม่ active หลังจากนั้น Machine cycle ถัดมา (ของ instruction ต่อไป) ก็สามารถเริ่มทำงานได้

และเมื่อความถี่ของ system clock (f) = 4MHz เวลาต่าง ๆ สามารถคำนวณได้ดังนี้

T-state = clock period (1/f) = 0.25 µs

Execution time ของ Opcode fetch: (4T) x 0.25 = 1.0 µs

Execution time ของ Memory Read: (3T) x 0.25 = 0.75 µs

∴ Execution time ของ Instruction ทั้งหมด: (7T) x 0.25 = 1.75 µs

5.3 Memory Write Machine Cycle

Machine cycle ลำดับที่สามที่จะกล่าวต่อไป คือ Memory write machine cycle โดยที่ Machine cycle นี้จะทำการ store ข้อมูลใน memory register ที่กำหนดในคำสั่ง ดังจะแสดงให้เห็นจากตัวอย่างต่อไปนี้

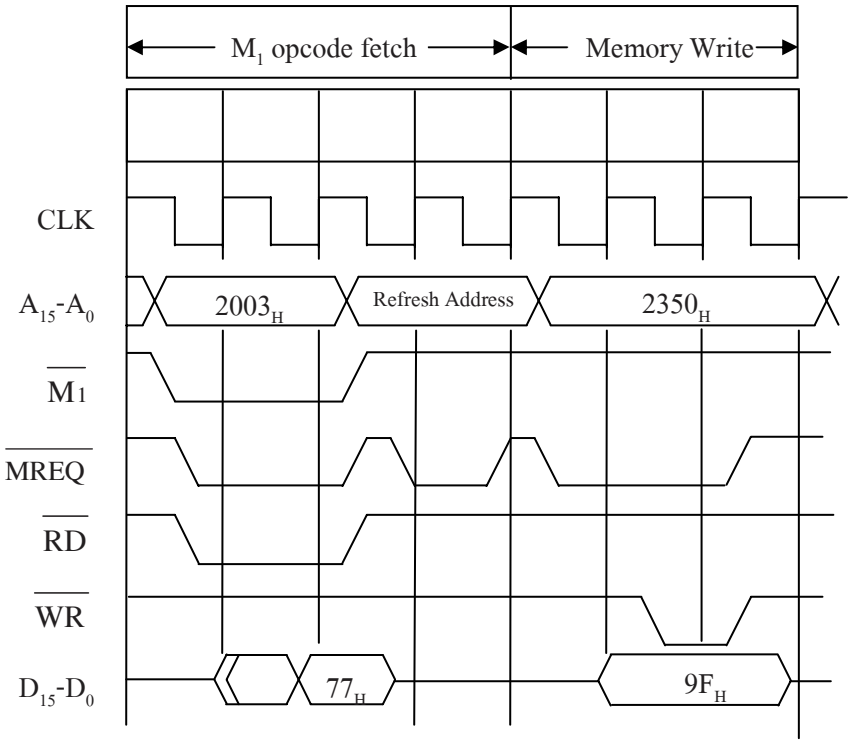
ตัวอย่างที่ 5.3 Register HL เก็บตำแหน่ง 2350_H และ accumulator เก็บค่า $9F_H$ โดยที่ instruction code 0111 0111 (77_H) ถูกเก็บไว้ที่ memory ตำแหน่ง 2003_H เมื่อคำสั่งนี้ถูก execute ข้อมูลใน accumulator จะถูกนำไปเก็บที่ตำแหน่งใน memory ที่ชี้โดย register HL จงหาค่าที่จะถูกเก็บไว้ใน bus และ timings เมื่อคำสั่งนี้กำลังถูก execute

คำสั่ง:

```
LD (HL),A      ;Copy contents of the accumulator into memory
               ;location, the address of which is stored in HL
               ;register.
```

คำตอบ

คำสั่งในตัวอย่างนี้เป็นคำสั่งขนาด 1 byte ที่มี 2 Machine cycles คือ Opcode fetch และ Memory write โดยที่ใน Machine cycle แรก Z80 จะ fetch ค่า code 77_H และใน Machine cycle ที่สอง Z80 จะคัดลอก byte $9F_H$ จาก accumulator ไปไว้ที่ memory ตำแหน่ง 77_H



รูปที่ 5.4 Z80 Memory Write Machine Cycle และ Bus Timings

รูปที่ 5.4 แสดงถึง timings ของ Machine cycles ทั้งสอง โดยที่มีรายละเอียดดังนี้

1. ในส่วนของ Opcode fetch cycle นั้น Z80 จะวางตำแหน่ง 2003_H ไว้ที่ address bus และอ่านค่าข้อมูล 77_H จาก data bus โดยใช้สัญญาณควบคุม $\overline{MREQ}$ และ $\overline{RD}$ เหมือนในตัวอย่างก่อน ๆ โดยที่ในตัวอย่างนี้ค่า PC จะถูกเพิ่มขึ้นเป็น 2004_H

2. ในช่วง T_3 และ T_4 Z80 จะ decode คำสั่งเครื่อง 77_H และเตรียมตัวเพื่อทำการเขียนข้อมูลลง memory

3. ในช่วงแรกของ Machine cycle ต่อมาซึ่งก็คือ Memory write cycle นั้น Z80 จะวางข้อมูล (2350_H) จาก register HL ไว้ที่ address bus ส่วนในตอนขาลงของ T_1 ใน Memory write cycle นั้น $\overline{MREQ}$ จะเป็น low และข้อมูล $9F_H$ จาก accumulator จะถูกวางไว้ที่ data bus

4. หลังจากนั้น รอให้ 1 T-state ผ่านไป (หลัง $\overline{MREQ}$ active) เพื่อให้ตำแหน่งที่จะเขียนข้อมูล มีความเสถียรแล้ว Z80 จะทำให้สัญญาณควบคุมการเขียน $\overline{WR}$ active เพื่อแสดงว่าจะเขียนข้อมูลไปยังตำแหน่งที่เก็บอยู่ใน address bus

5. หลังจากขาลงของ T_3 ใน Memory write cycle สัญญาณควบคุมทั้งสองจะไม่ active และ data bus จะถูกเปลี่ยนเป็น high impedance

5.4 ทบทวนสาระสำคัญ

1. คำสั่งทุกคำสั่งต้องมีอย่างน้อย 1 Machine cycle นั้นคือ M_1 หรือ Opcode fetch machine cycle เสมอ ซึ่งจะมี 4 T-states
2. Memory read cycle คล้ายกับ Opcode fetch cycle จะต่างกันตรงที่ ใน Opcode fetch cycle จะมี $\overline{M_1}$ active ด้วย
3. Memory write cycle มี $\overline{MREQ}$ และ $\overline{WR}$ active ในขณะที่ Memory read cycle มี $\overline{MREQ}$ และ $\overline{RD}$ active
4. $\overline{WR}$ และ $\overline{RD}$ จะไม่ active พร้อมกัน เนื่องจากไมโครโพรเซสเซอร์ Z80 ไม่สามารถจะอ่านและ เขียนพร้อมกันได้

ใน Memory read cycle $\Rightarrow$ $\overline{MREQ}$ และ $\overline{RD}$ จะ active เพื่อ enable memory output buffer โดยที่ตำแหน่งของ memory ที่ต้องการอ่านข้อมูลควรที่จะอยู่ที่ address bus แล้ว เพื่อที่จะอ่านค่าข้อมูลใน memory ที่ ตำแหน่งนั้นผ่านทาง data bus

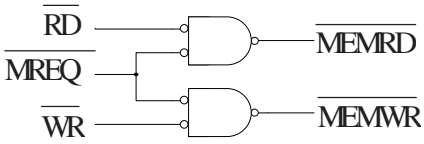
ในขณะที่ใน Memory write cycle $\Rightarrow$ $\overline{MREQ}$ จะ active และข้อมูลที่จะเขียนจะต้องอยู่บน data bus แล้ว $\overline{WR}$ ค่อย active เพื่อที่จะเขียนข้อมูลจาก data bus ลงไปยังตำแหน่งที่อ่านได้จาก address bus

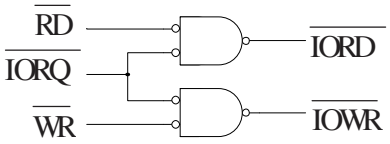
5.5 การสร้างสัญญาณควบคุม

Z80 มีสัญญาณควบคุมอยู่ 4 สัญญาณ คือ $\overline{MREQ}$ $\overline{IORQ}$ $\overline{RD}$ และ $\overline{WR}$

การทำงาน	$\overline{MREQ}$	$\overline{IORQ}$	$\overline{RD}$	$\overline{WR}$
การอ่าน memory	0	1	0	1
การเขียน memory	0	1	1	0
การอ่านอุปกรณ์ I/O	1	0	0	1
การเขียนบนอุปกรณ์ I/O	1	0	1	0

ในกรณีที่ต้องการสัญญาณ $\overline{\text{MEMRD}}$ $\overline{\text{MEMWR}}$ $\overline{\text{IORD}}$ และ $\overline{\text{IOWR}}$ สามารถสร้างสัญญาณได้ดังนี้





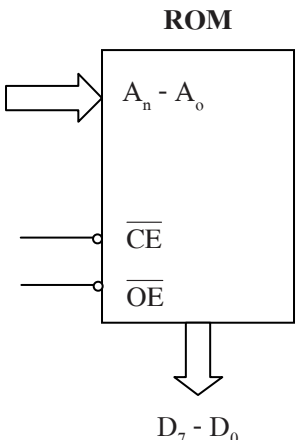
บทที่ 6 การต่อประสานกับหน่วยความจำ

ในการทำงานของ Z80 ไมโครโพรเซสเซอร์เพื่อ execute โปรแกรมหนึ่งๆ มีความจำเป็นที่จะต้อง access หน่วยความจำบ่อยครั้ง วังเดตได้จากในช่วงแรกของการทำงานของทุกคำสั่ง จะเป็น opcode fetch machine cycle ซึ่งจำเป็นต้อง fetch คำสั่งจากหน่วยความจำ

บทนี้จะว่าด้วยการต่อประสาน Z80 ไมโครโพรเซสเซอร์กับชิบหน่วยความจำ โดยเริ่มจากศึกษาโครงสร้างของชิบหน่วยความจำ เพื่อให้ได้มาซึ่งรายละเอียดที่จำเป็นในการอ่านและเขียนข้อมูลในหน่วยความจำ รวมไปถึงขั้นตอนที่สำคัญต่างๆ สำหรับการต่อประสานกับหน่วยความจำ

6.1 แนวคิดพื้นฐานเกี่ยวกับหน่วยความจำ

A_n-A_0 = address signals มี n+1 เส้น



ขนาดของ memory	จำนวนสายสัญญาณ
1K (0000-03FF)	10 (A_9-A_0)
2K (0000-07FF)	11 (A_{10}-A_0)
4K (0000-0FFF)	12 (A_{11}-A_0)
8K (0000-1FFF)	13 (A_{12}-A_0)
16K (0000-3FFF)	14 (A_{13}-A_0)
32K (0000-7FFF)	15 (A_{14}-A_0)
64K (0000-FFFF)	16 (A_{15}-A_0)

D_7-D_0 = Data Bus

$\overline{CE}$ (Chip Enable) หรือ $\overline{CS}$ (Chip Select) เป็นสัญญาณเลือก chip

$\overline{OE}$ (Output Enable) เป็นสัญญาณสำหรับ enable output buffer ของ memory

ถ้าเป็น static RWM จะมีขา $\overline{WR}$ ด้วย

Memory บางเบอร์อาจมีขา Chip Enable มากกว่า 1 ขา เช่น $\overline{CS_1}$ (active low) และ CS_2 (active high)

เบอร์ที่ใช้อยู่

EPROM	RWM
2716	6264
2732	62256
2764	…
27120	
27256	
…	

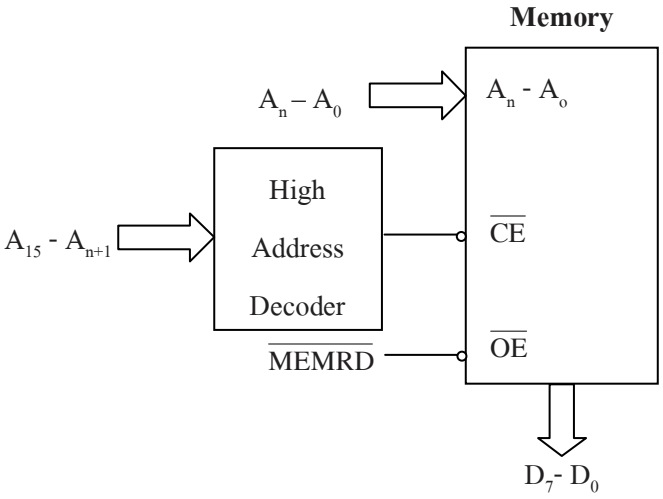
ในจำนวนนี้มีรุ่นที่ flash memory ด้วย

การจัดวางเรียงขาของ dynamic RAM จะต่างจาก static RAM การต่อ dynamic RAMโดยทั่วไปต้องใช้ dynamic RAM controller ด้วย

ขาของ memory ต่างขนาดกันจะถูกสร้างให้มีตำแหน่งและหน้าที่ของขาใกล้เคียงกันให้มากที่สุด ทำให้เราสามารถออกแบบวงจรให้ใช้กับ memory ได้หลายขนาดโดยการใช้ DIP switch เลือก

6.2 การถอดรหัสตำแหน่ง

การเลือกตำแหน่งจะประกอบไปด้วยสัญญาณ 2 ส่วน คือ สัญญาณ high address (A_{15} - A_{n+1}) และ low address (A_n - A_o)



ส่วนของ low address สามารถต่อเข้ากับขา address ของ memory ได้โดยตรง แต่ส่วนของ high address ซึ่งมี 3 วิธีด้วยกัน คือ

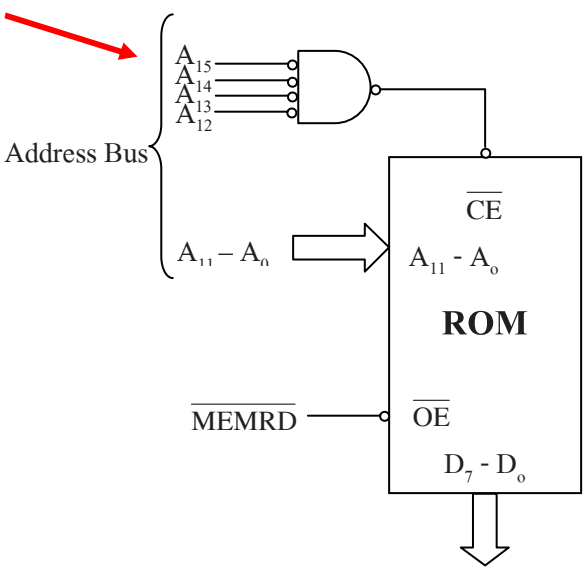
1. ใช้ logic gates
2. ใช้ decoder เช่น 2-line-to-4-line decoder 1-of-4 74LD139,155,156, 539
   3-line-to-8-line decoder 1-of-8 74LD138, 538, 137 (มี latch)
   4-line-to-16-line decoder 1-of-16 74LD154, 159
3. ใช้ magnitude comparator IC (เปลี่ยน address ได้)

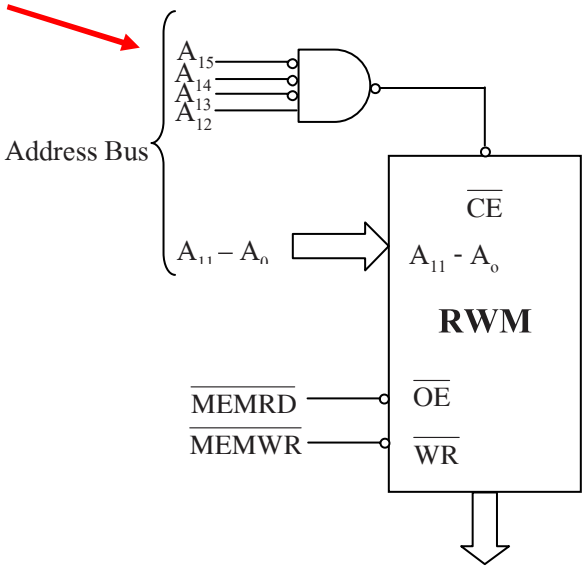
ตัวอย่าง ต้องการต่อ

	ขนาด	ตำแหน่ง	จำนวนสัญญาณ
ROM	4K	0000 – 0FFF	12 เส้น (A11 – A0)
RWM	4K	1000 – 1FFF	12 เส้น (A11 – A0)

	ROM	RWM
Begin address	0000 0000 0000 0000	0001 0000 0000 0000
End address	0000 1111 1111 1111	0001 1111 1111 1111

	15	14	13	12	11 - 0
ROM	0	0	0	0	X – X
RWM	0	0	0	1	X – X





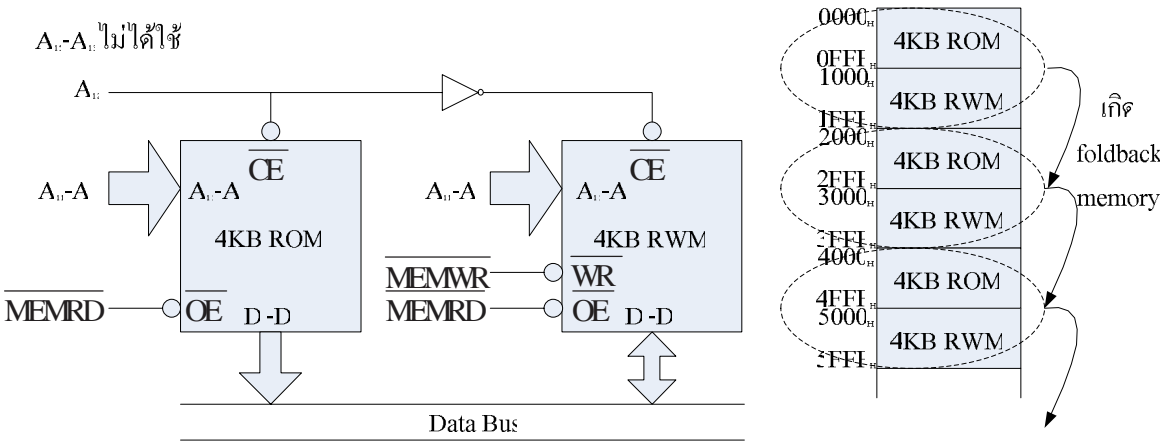
Address ที่ CPU ใช้

3000	1000	Address ภายใน memory chip
3FFF	1FFF	
4000	0000	
4FFF	0FFF	

การใช้ magnitude comparator เป็นอีกวิธีหนึ่งที่เราสามารถใช้ในการ decode high address ได้ เช่น 10k x 3 : comparator สามารถใช้กับการเลือก address ของ I/O ได้ด้วย

6.3 Partial Address Decoding และ Fold back Memory

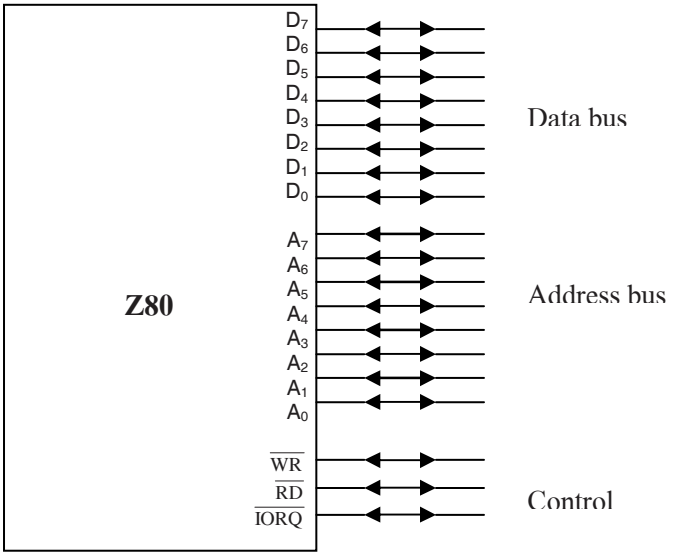
ในกรณีที่ระบบใช้ memory chips น้อย เราอาจไม่ต้องใช้ decoder สำหรับ high address เลยก็ได้ แต่โปรแกรมจะต้อง access code หรือ data ในขอบเขตของ memory ที่มีอยู่เท่านั้นเช่น



วงจรข้างบนจะทำให้สามารถ access ROM & RWM ตัวเดียวกัน จากหลาย ๆ ตำแหน่ง

บทที่ 7 การต่อประสานกับอุปกรณ์ I/O

ไมโครโพรเซสเซอร์ Z80 ติดต่อกับอุปกรณ์ภายนอกด้วยขาสัญญาณต่าง ๆ ดังในรูปที่ 7.1 และมีรายละเอียดตามตารางที่ 7.1



รูปที่ 7.1 ขาสัญญาณของ Z80 ที่ใช้ติดต่อกับอุปกรณ์ภายนอก

D_0-D_7	เป็น data bus ขนาด 8 บิต ใช้รับส่งข้อมูลระหว่าง Z80 กับอุปกรณ์ภายนอก
A_0-A_7	เป็น address bus ขนาด 8 บิตใช้กำหนดหมายเลขช่องสัญญาณ I/O ในการติดต่อกับอุปกรณ์ภายนอก Z80 ใช้เฉพาะ A_0-A_7 กำหนดตำแหน่ง ทำให้ติดต่ออุปกรณ์ภายนอกได้ด้วย address 00_H-FF_H
$\overline{IORQ}$	ทั้งสามสัญญาณนี้ active ด้วย logic "0" เมื่อ Z80 ทำคำสั่ง IN หรือ OUT ตามขั้นตอนดังที่จะกล่าวต่อไป
$\overline{RD}$ (ReaD) $\overline{WR}$ (WRite)	

ตารางที่ 7.1 รายละเอียดของขาสัญญาณของ Z80 ที่ใช้ติดต่อกับอุปกรณ์ภายนอก

ในการถอดรหัสเพื่อเรียกหมายเลขพอร์ตผู้ใช้จะต้องเลือกใช้สัญญาณ ($\overline{IORQ}$ กับ $\overline{RD}$) หรือ ($\overline{IORQ}$ กับ $\overline{WR}$) หรือ เฉพาะ $\overline{IORQ}$ แล้วแต่กรณี

การต่อพอร์ต I/O ให้กับ Z80 อาจใช้ไอซีดิจิตอล หรือใช้ชิพสนับสนุน เช่น 8255 หรือ Z80 PIO ก็ได้

7.1 การต่อประสานกับอุปกรณ์เอาท์พุต

คำสั่ง OUT (8-bit), A

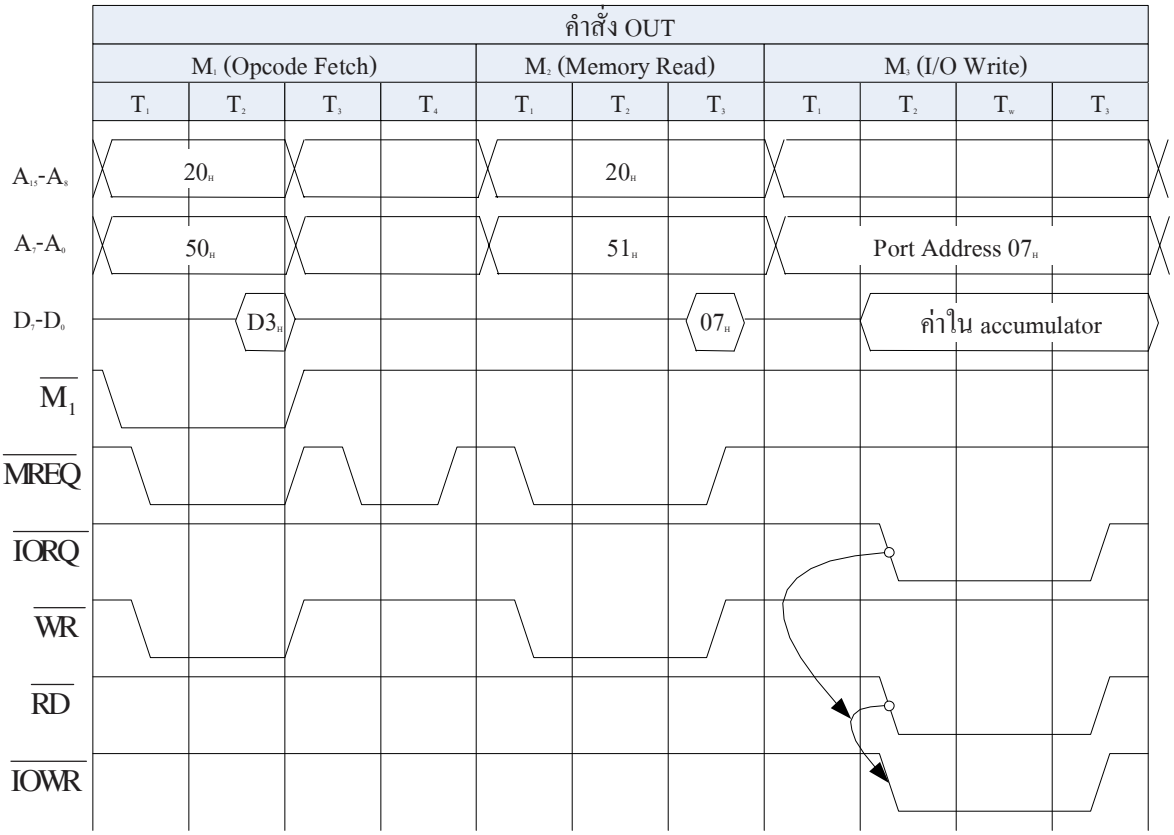
2050 : D307 OUT (07H), A

2052 :

output address
ขนาด 8 bits

รูปที่ 7.2 แสดงถึงการทำงานของคำสั่ง OUT

Z80 จะเพิ่ม T_W 1 state สำหรับ I/O โดยอัตโนมัติ



รูปที่ 7.2 Timing diagram แสดงการทำงานของคำสั่ง OUT

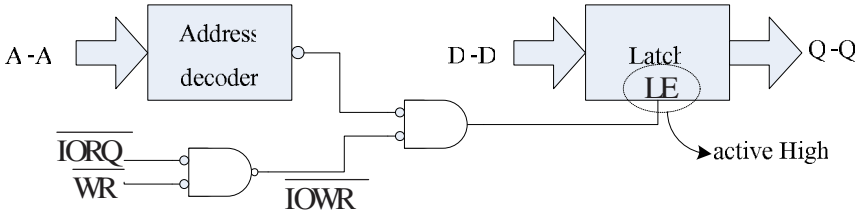
สิ่งที่จะต้องออกแบบมีดังนี้

1. decode low address (A_7- A_0) เพื่อเลือกตำแหน่ง output device ที่ต้องการ เพื่อสร้าง $\overline{\text{IOADDR}}$ pulse
2. รวมสัญญาณเลือกตำแหน่งจากข้อ 1 $\overline{\text{IORQ}}$, และ $\overline{\text{WR}}$ เพื่อสร้างสัญญาณเลือกอุปกรณ์เอาท์พุต ที่ต้องการ

อุปกรณ์เอาท์พุตจะเป็นอุปกรณ์ประเภท latch หรือ D flip-flop (74LS373, 74LS374 หรือ 7475) ดังในรูปที่ 7.3

ถ้า LE = 1 จะได้ว่า Q มีค่าเท่ากับ D

LE = 0 จะได้ว่า Q จะคงสถานะไว้



รูปที่ 7.3 การต่อประสานกับอุปกรณ์เอาท์พุต

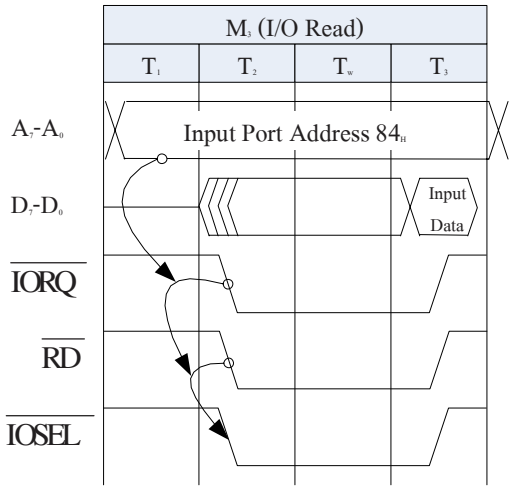
7.2 การต่อประสานกับอุปกรณ์อินพุต

คำสั่ง IN A, (8–bit) 2065 : DB84 IN A, (84H)

input address
ขนาด 8 bits

รูปที่ 7.3 แสดงถึงการทำงานของคำสั่ง IN

Z80 จะใส่ T_w 1 state โดยอัตโนมัติ



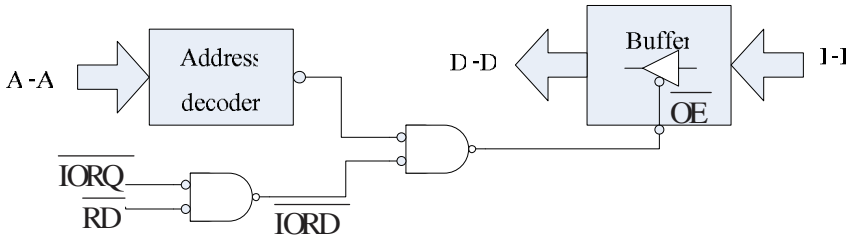
รูปที่ 7.4 Timing diagram แสดงการทำงานของคำสั่ง IN

สิ่งที่จะต้องออกแบบมีดังนี้

1. decode low address เพื่อเลือกตำแหน่งของ input device ที่ต้องการ
2. รวมสัญญาณเลือกตำแหน่ง $\overline{IORQ}$ และ $\overline{RD}$ เพื่อสร้างสัญญาณเลือกอุปกรณ์อินพุตที่ต้องการ

อุปกรณ์อินพุตจะเป็นอุปกรณ์ประเภท buffer (74LS244, 74LS245) ดังในรูปที่ 7.5

ถ้า $\overline{OE}$ ไม่ active จะได้ว่า เอาท์พุตของ buffer จะเป็น high impedance

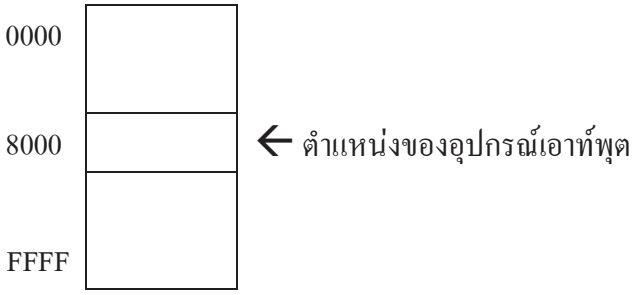


รูปที่ 7.5 การต่อประสานกับอุปกรณ์อินพุต

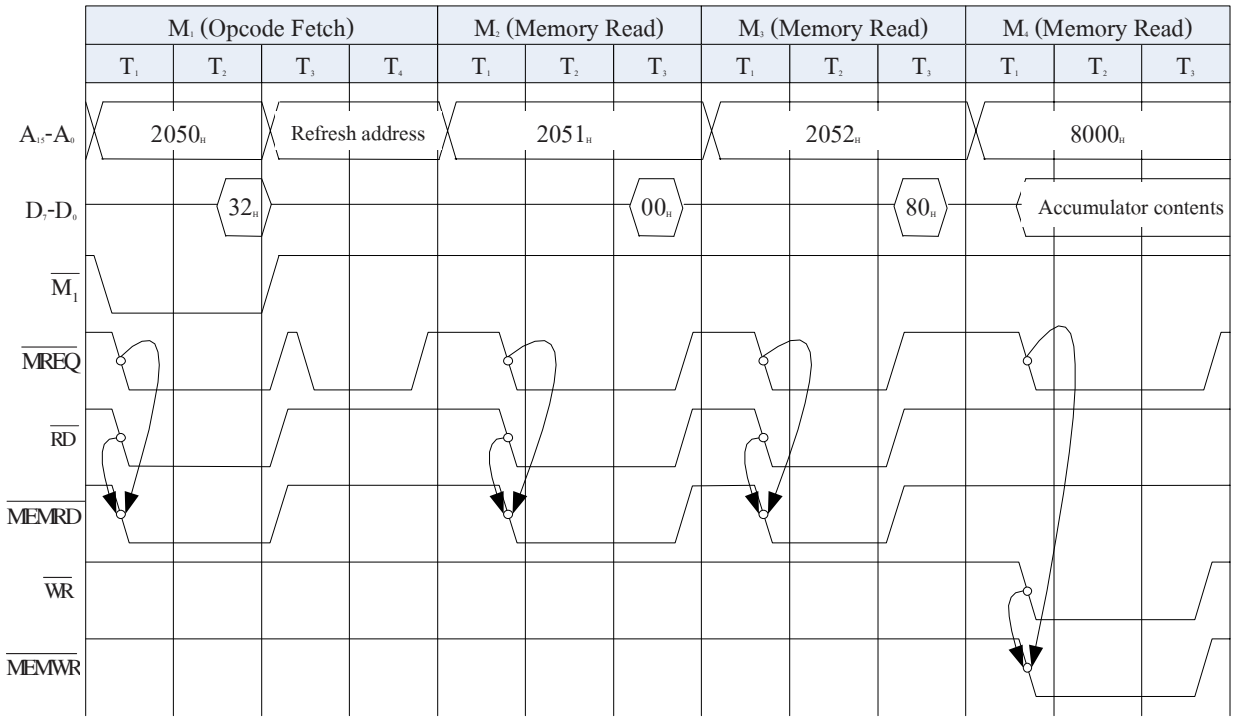
เราอาจสร้างอุปกรณ์อินพุตและเอาท์พุตไว้ที่ตำแหน่งเดียวกันได้ โดยไม่มีปัญหาในการทำงาน

7.3 Memory – Mapped I/O

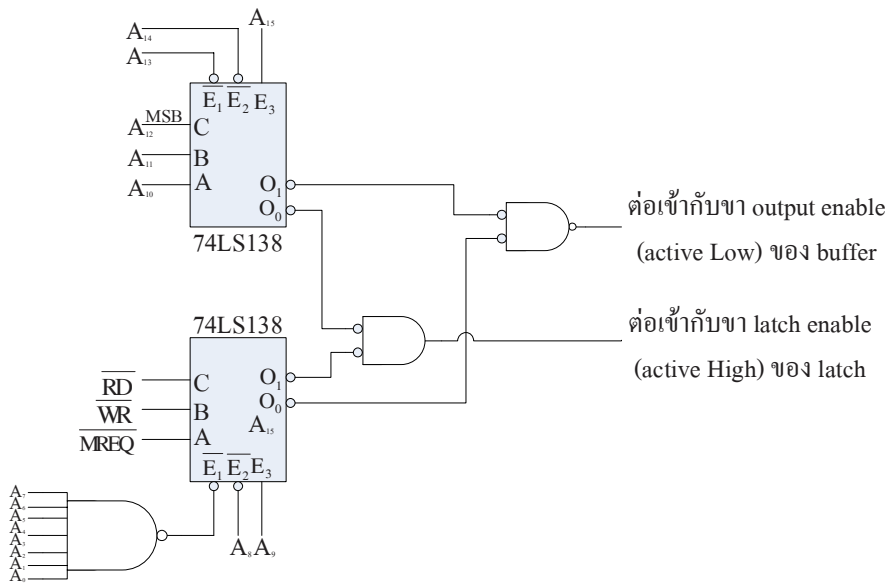
2050H: 32 00 80 LD (8000H), A



Timing diagram ของการต่อประสานกับอุปกรณ์อาท์พุตด้วย memory-mapped I/O เป็นดังแสดงรูปที่ 7.6



รูปที่ 7.6 Timing diagram แสดงการต่อประสานอุปกรณ์เอาท์พุตด้วย memory-mapped I/O



รูปที่ 7.7 ตัวอย่างการต่อประสานด้วย memory-mapped I/O

จากรูปที่ 7.7 ซึ่งแสดงตัวอย่างการต่อประสานอุปกรณ์อินพุตและเอาท์พุตด้วย memory-mapped I/O ซึ่งจะได้ว่าตำแหน่งของอุปกรณ์อินพุตและเอาท์พุตเป็นดังนี้

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ตำแหน่งเอาท์พุต	1	0	0	0	0	0	1	0	1	1	1	1	1	1	1	1	= 82FF
ตำแหน่งอินพุต	1	0	0	0	0	1	1	0	1	1	1	1	1	1	1	1	= 86FF

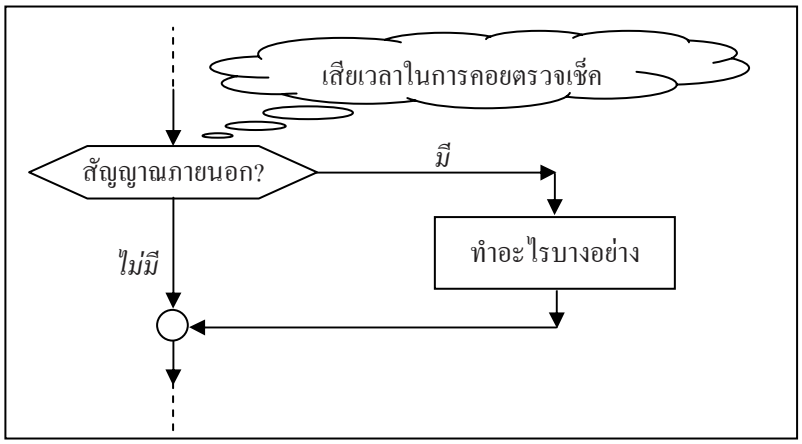
คำสั่งที่ใช้กับ memory-mapped I/O คือคำสั่ง LD ไม่ใช่ IN หรือ OUT

LD A, (86FFH) → input

LD (82FFH), A → output

การรับรู้สัญญาณภายนอกสามารถทำได้ 2 วิธี

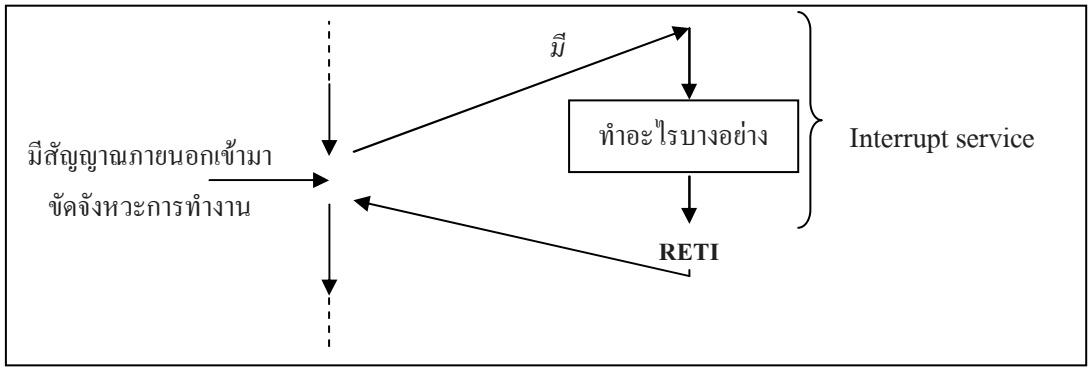
1. เขียนโปรแกรมตรวจสอบ



รูปที่ 8.1 การรับรู้สัญญาณภายนอกโดยการเขียนโปรแกรมตรวจสอบ

ถ้ามีหลายสัญญาณ สัญญาณที่ถูกตรวจสอบก่อนจะมีลำดับความสำคัญสูงกว่าโดยอัตโนมัติ

2. ใช้ Interrupt



รูปที่ 8.2 การรับรู้สัญญาณภายนอกโดยการเขียนใช้ interrupt

ข้อดีของการใช้ interrupt

- ไม่ต้องเขียนโปรแกรมให้คอยตรวจสอบสัญญาณภายนอก
- การตอบสนองแน่นอนกว่าการใช้โปรแกรมตรวจสอบ

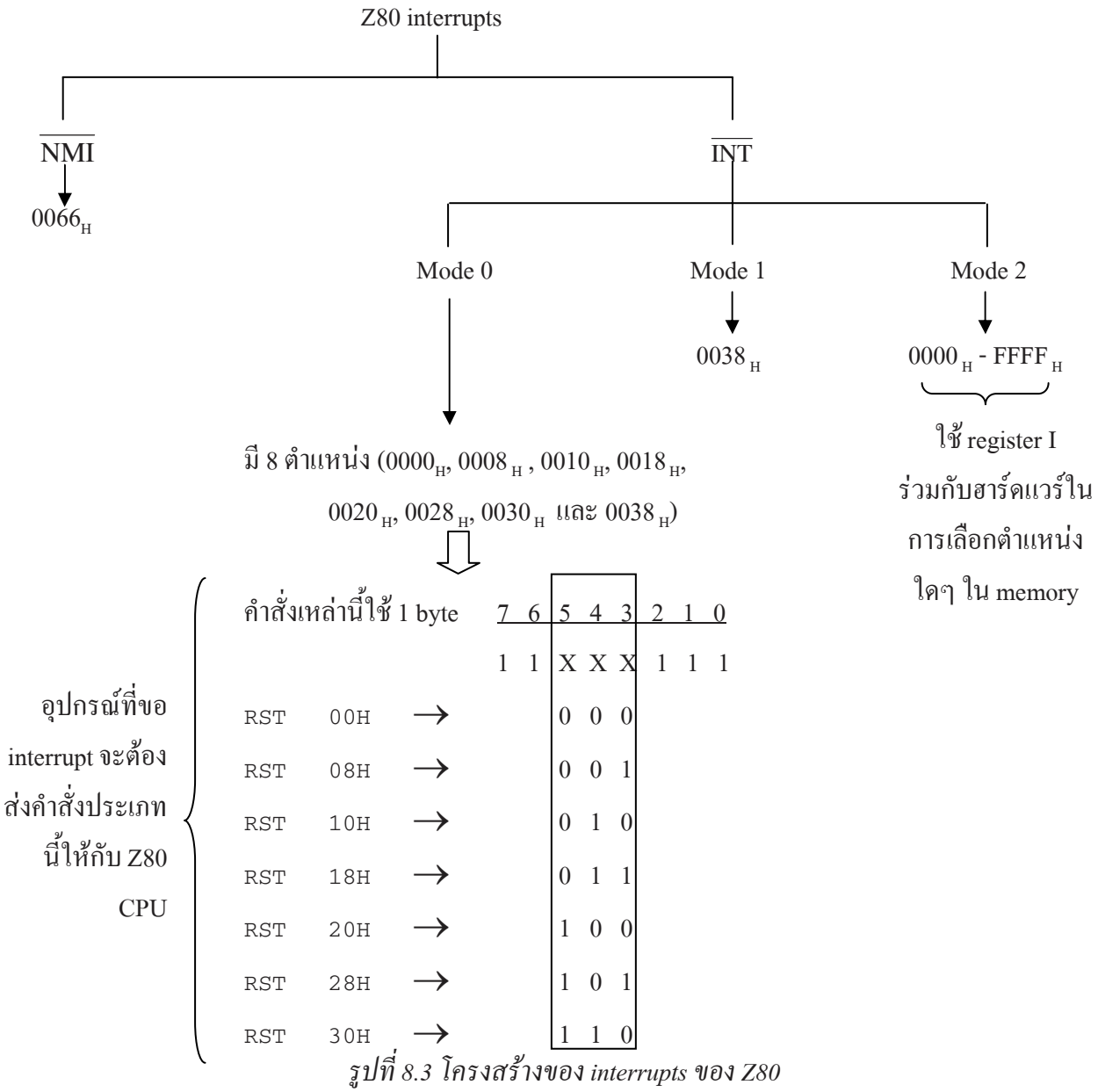
Interrupts มี 2 แบบ คือ

1. Maskable interrupt (INT) เป็น interrupt ที่จะถูกละเลยหรือชะลอได้ หากไมโครโพรเซสเซอร์กำลังทำงานอื่นที่สำคัญ คือเลือกที่จะ enable หรือ disable ได้

2. Nonmaskable interrupt (NMI) เป็น interrupt ที่จะต้องตอบสนองอย่างทันถ่วงที โดยที่ผู้ใช้ไม่สามารถหยุดยั้งหรือ disable ได้ด้วซอฟต์แวร์ควรใช้ในกรณีฉุกเฉิน

8.1 ความรู้เบื้องต้นสำหรับ Interrupt I/O ใน Z80

Z80 มีขาอินพุตสำหรับที่ใช้เกี่ยวกับการ interrupt อยู่ด้วยกัน 2 ขา คือ $\overline{INT}$ สำหรับ Maskable interrupt และ ขา $\overline{NMI}$ สำหรับ Nonmaskable interrupt



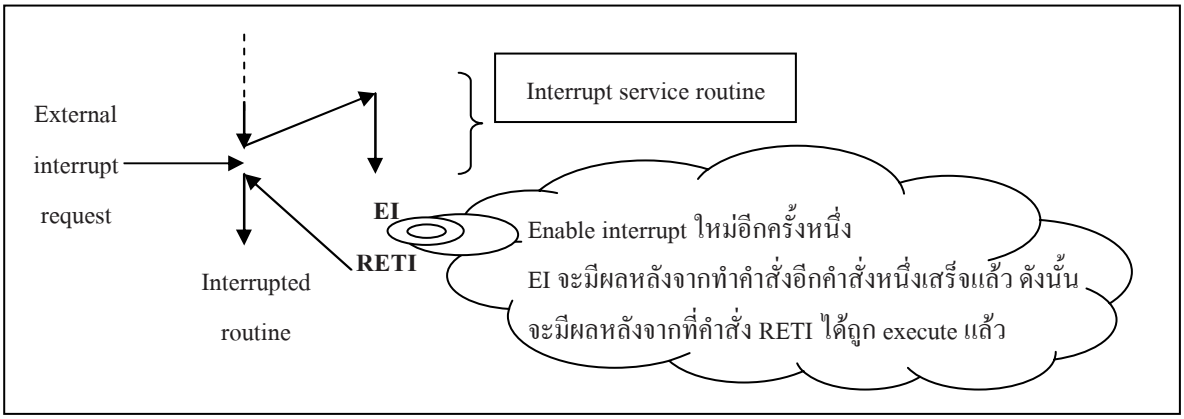
รูปที่ 8.3 โครงสร้างของ interrupts ของ Z80

8.2 การ enable และ disable interrupt

- คำสั่ง EI (Enable Interrupt)
  - มีขนาด 1 byte
  - ใช้ในการ reset ค่า interrupt flip-flops เพื่อให้สามารถรับสัญญาณ $\overline{INT}$ จากภายนอกได้ โดยที่รายละเอียดเกี่ยวกับ interrupt flip-flops ทั้งสอง (IFF1 และ IFF2) จะกล่าวในหัวข้อถัดไป

โดยที่ $\overline{INT}$ จะถูก disable เมื่อ

- ใช้คำสั่ง DI (Disable Interrupt) ซึ่งจะกล่าวในหัวข้อย่อยถัดไป
- System reset
- Z80 ตอบสนองต่อสัญญาณ interrupt (interrupt acknowledge) วิธีนี้ Z80 จะใช้ป้องกันการเกิด interrupt ซ้อน



รูปที่ 8.4 การใช้คำสั่ง EI เพื่อป้องกันการเกิด interrupt ซ้อน

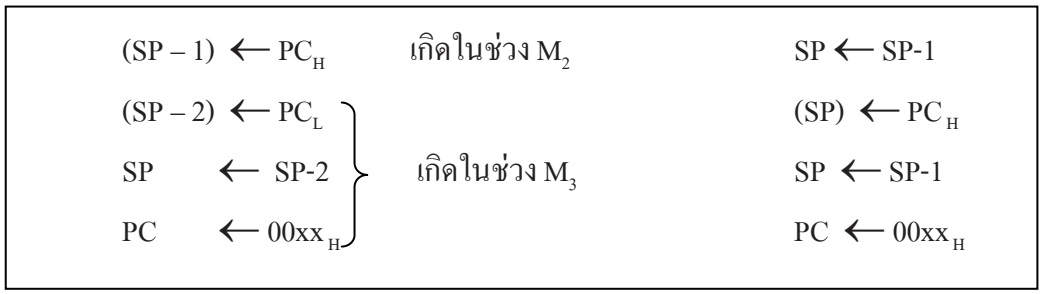
Z80 จะตรวจเช็คสัญญาณการ interrupt ใน T-state สุดท้ายของแต่ละ instruction cycle ถ้าพบสัญญาณ $\overline{INT}$ active ก็จะเข้าสู่ interrupt process ตามรูปที่ 8.4

$\underbrace{\overline{M1} \text{ กับ } \overline{IORQ}}_{\overline{INTA}}$ จะ active เพื่อเป็นการรับรู้ interrupt (interrupt acknowledge)

Z80 จะใส่ $T_{w1} + T_{w2}$ โดยอัตโนมัติ

อุปกรณ์ทางฮาร์ดแวร์เมื่อได้รับ $\overline{INTA}$ จะต้องส่งคำสั่ง RST XXH ไปไว้บน data bus จากนั้น Z80 จะเข้าสู่การ refresh

คำสั่ง RST XXH จะมี operation ดังต่อไปนี้



รูปที่ 8.5 การทำงานของคำสั่ง RST XXH

- คำสั่ง DI (Disable Interrupt)

  - มีขนาด 1 byte
  - ใช้ในการ reset ค่า interrupt flip-flops ทั้งสอง (IFF1 และ IFF2) ซึ่งจะกล่าวในหัวข้อถัดไป
  - ใช้เมื่อไม่ต้องการให้ Z80 รับรู้สัญญาณ $\overline{INT}$

คำสั่ง DI นี้ไม่มีผลต่อการ interrupt แบบ NMI

8.3 Interrupt flip-flops

CPU จะทำการตรวจสอบสถานะภาพต่อการตอบสนองการ interrupt ที่ IFF หรือ interrupt flip-flop ในกรณีของ Z80 จะมีที่แสดงสถานะภาพ ในการ interrupt อยู่ 2 บิต คือ IFF1 และ IFF2 โดยทั้งสองบิตนี้จะขึ้นอยู่กับผลที่ได้ด้วยจากการกระทำของ CPU หรือการเขียนโปรแกรมโดยใช้คำสั่งในการ set หรือ reset flip-flops

หลักการที่สำคัญคือ IFF1 ทำหน้าที่เป็นตัวกำหนดการ enable หรือ disable ของการ interrupt โดยที่ IFF2 จะมีหน้าที่หลักในการเก็บข้อมูลชั่วคราวของ IFF1 ในขณะที่มีการ reset CPU ทางขา reset ทั้ง IFF1 และ IFF2 จะได้รับการ reset ไปด้วย การแสดงสถานะ "0" ของ IFF1 จะเป็นการ disable การ interrupt กล่าวคือ IFF1 = "0" CPU จะไม่รับรู้ต่อการ interrupt ที่เข้ามาทาง $\overline{INT}$ การ set IFF สามารถกระทำได้ด้วยคำสั่ง EI โดยสถานะภาพของ IFF1 และ IFF2 ที่จะเปลี่ยนแปลงเนื่องจากการกระทำต่าง ๆ สรุปได้ดังตารางในรูปที่ 8.6

จากรูปที่ 8.6 พอสรุปได้ว่า การกำหนดการ enable จะต้องทำการ set flip-flop IFF1 หรือกล่าวอีกนัยหนึ่ง การยอมให้เกิดการ interrupt ได้ก็ต่อเมื่อ CPU ตรวจสอบ IFF1 ว่าอยู่ในสภาวะ enable หรือไม่การตรวจสอบสภาวะ enable ว่าได้รับการ enable หรือ disable ในบางกรณีทำได้โดยการตรวจสอบทาง parity bit นั่นคือ การกระทำคำสั่ง LD A, I และ LD A, R จะมีผลให้ค่าของ IFF2 ไปเก็บยัง parity flag

การทำงาน	IFF1	IFF2	หมายเหตุ
CPU reset	0	0	$\overline{INT}$ ถูก disable
คำสั่ง DI	0	0	$\overline{INT}$ ถูก disable
คำสั่ง EI	1	1	$\overline{INT}$ ถูก enable หลังคำสั่งถัดไป
LD A, I	●	●	parity flag → IFF2
LD A, R	●	●	parity flag → IFF2
$\overline{NMI}$ operation	0	●	$\overline{INT}$ ถูก disable
$\overline{INT}$ operation	0	0	-
คำสั่ง RETN	IFF2	●	IFF2→ IFF1
คำสั่ง RETI	●	●	-

หมายเหตุ: "●" หมายถึง ไม่เปลี่ยนแปลง

รูปที่ 8.6 สถานะของ interrupt flip-flops IFF1 และ IFF2

เมื่อมีการ interrupt แบบ NMI ขึ้นจะเกิดสภาวะ disable ทันทีที่ IFF1 กล่าวคือมันจะได้ รับการ reset นั่นคือระหว่างการinterrupt แบบ NMI นี้ การ interrupt แบบอื่นจะเข้ามาอีกไม่ได้ CPU จะไม่รับรู้ทั้งสิ้นสถานะเดิมก่อนการ interrupt แบบ NMI (สถานะการ disable หรือ enable) จะได้รับการเก็บรักษาไว้ที่ IFF2 ซึ่งระหว่างนี้จะได้รับการตรวจสอบได้เช่นกันว่า ก่อนการเข้าไปสู่โหมด NMI สภาวะการเป็นอย่างไร และเมื่อกลับเข้าโปรแกรมหลักด้วยคำสั่ง RETN จะทำให้สถานะเดิมเก็บรักษาไว้ใน IFF1 ใหม่การตอบสนองต่อ $\overline{INT}$ ก็จะทำให้ IFF1 และ IFF2 ได้รับการ reset เช่นกัน ดังนั้นเมื่อมีการ interrupt แบบ INT สัญญาณ $\overline{INT}$ ครั้งต่อไปจะไม่สามารถได้รับการตอบสนองจนกว่าจะมีคำสั่ง EI $\overline{INT}$ จึงจะได้รับการตอบสนองนั่นเอง ส่วนการ execute คำสั่ง RETI จะไม่มีผลทำให้ IFF1 และ IFF2 เกิดการเปลี่ยนแปลง

8.4 คำสั่ง Returns ต่างๆ

คำสั่ง RET - return จาก subroutine

RETI - return จาก interrupt service routine

RETN - return จาก non-maskable interrupt

คำสั่ง RET ใช้แทนคำสั่ง RETI ได้ แต่ในบางกรณี อุปกรณ์ I/O จะถูกสร้างให้รู้จักกับคำสั่ง RETI ซึ่งจะทำให้เราสามารถต่อพ่วงอุปกรณ์ที่ต้อง interrupt Z80 ได้หลายตัว ส่วนคำสั่ง RETN นั้นนอกจากจะทำหน้าที่ของ RET แล้ว จะทำหน้าที่ต่อไปนี้

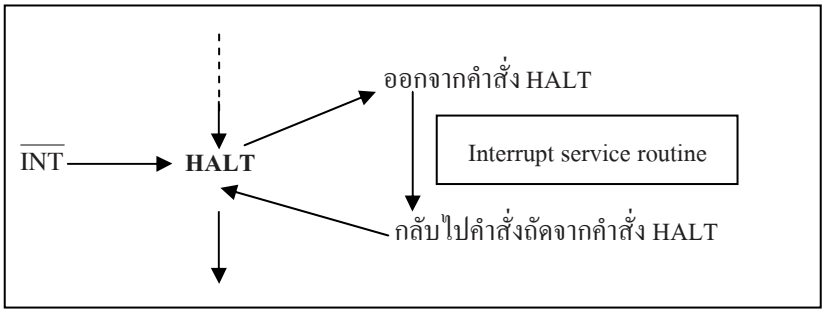
ตอนเกิด $\overline{NMI}$ IFF1 ← 0 ส่วนค่าของ IFF2 จะคงเดิม

RETN IFF1 ← IFF2

copy

Interrupt จะไม่เกิดขึ้นในขณะที่มีสัญญาณ $\overline{BUSRQ}$ หรือขณะที่กำลังทำงานให้กับ interrupt ที่มี priority สูงกว่า (เราต้องออกแบบให้ทำงานในลักษณะนี้)

มีอีกลักษณะหนึ่งคือตามรูปที่ 8.7 คือสัญญาณ $\overline{INT}$ ทำให้ออกจากคำสั่ง HALT ได้



รูปที่ 8.7 การออกจากคำสั่ง HALT เนื่องจากสัญญาณ $\overline{INT}$

8.5 Non-maskable Interrupt (NMI)

การ interrupt ชนิดนี้จะป้อนเข้าทางขา $\overline{NMI}$ ของ CPU โดยที่เมื่อ CPU รับสัญญาณนี้แล้ว CPU จะไม่กระทำคำสั่งถัด ไป แต่จะตอบสนองต่อการ interrupt โดยการเปลี่ยนค่าของ PC ให้เป็น 0066_H เพื่อให้ภาวะการ fetch ครั้งต่อไปเกิดขึ้นที่ address นี้ การตอบสนองในกรณีของ NMI นี้ CPU ถือว่าเป็นส่วนสำคัญมากที่จะต้องกระทำโดยที่คำสั่งทางซอฟต์แวร์หรือขบวนการอื่นใดไม่สามารถเข้ามาหยุดยั้งการ interrupt ชนิดนี้ได้ โดยที่จะมี priority สูงกว่า $\overline{INT}$ แต่ต่ำกว่า $\overline{BUSRQ}$ การ interrupt ด้วยวิธีนี้จึงมักใช้ในกรณีว่ามีเหตุการณ์สำคัญที่สุด จากหลักการทั่วไปของการ interrupt ค่าเดิมของ PC ที่ CPU จะกระทำต่อไปในโปรแกรมหลักจะได้รับการเก็บรักษาไว้ใน stack และการกลับคืนสู่โปรแกรมหลักเดิมนั้นกระทำได้ด้วยคำสั่ง RETN (Return from non-maskable interrupt)

NMI เป็นแบบ falling edge-sensitive เมื่อมีสัญญาณ $\overline{NMI}$ เข้ามาเมื่อใดสัญญาณ $\overline{NMI}$ นี้จะถูก latch ไว้ก่อนภายใน จนกระทั่งทำคำสั่งปัจจุบันเสร็จ จึงจะเกิดการ interrupt

NMI มีขั้นตอนดังต่อไปนี้

1. เมื่อเกิดขอบขาลง ที่ขา $\overline{NMI}$ Z80 จะค้าง (latch) สัญญาณ interrupt request ไว้ภายใน
2. ในช่วง T-state สุดท้ายของแต่ละคำสั่ง Z80 จะตรวจสัญญาณ Interrupt ที่เข้ามาถ้าพบว่ามีสัญญาณของขาลงที่ถูกค้างไว้ก่อนหน้านี้ และ $\overline{BUSRQ}$ ไม่ active จะเกิด interrupt
3. Z80 จะเก็บค่า PC ปัจจุบันไว้บน stack
4. IFF1 จะถูกเปลี่ยนค่าเป็น 0
5. Z80 เริ่มต้นทำงานใน interrupt service routine ที่ตำแหน่ง 66_H โดยไม่มีส่วนของฮาร์ดแวร์เพิ่มเติม (คล้าย IM0 ที่จะกล่าวต่อไป)

6. เมื่อทำงานจนกระทั่งเจอคำสั่ง retn Z80 จะคัดลอกค่าของ IFF2 ไปยัง IFF1 และกลับไปทำงานต่อในส่วนของโปรแกรมที่ถูกขัดจังหวะ (interrupted routine) (pop ค่า return address ใน stack มาใส่ใน PC)

8.6 Maskable Interrupt (INT)

ในการ interrupt ชนิดนี้ ผู้ใช้ต้อง interrupt ผ่านเข้ามาทาง ขา $\overline{INT}$ ของ CPU เมื่อ CPU ได้รับสัญญาณนี้แล้ว CPU จะตรวจสอบสถานะของตัวเองว่าจะตอบสนองต่อการ interrupt หรือไม่ การที่จะตอบสนองหรือไม่สามารถโปรแกรมได้ด้วยซอฟต์แวร์ ดังนั้นผู้โปรแกรมจึงสามารถกำหนดสภาวะการ interrupt ให้ได้รับการตอบสนองตรงเฉพาะส่วนใดส่วนหนึ่งของโปรแกรมได้ การ interrupt ด้วยวิธีนี้มีอยู่ด้วยกัน 3 โหมด ซึ่งการแยกโหมดก็ทำได้ด้วยการกำหนดในคำสั่งทางซอฟต์แวร์ โดยแยกเป็นโหมด 0 (IM0) โหมด 1 (IM1) และ โหมด 2 (IM2)

8.6.1 การ interrupt โหมด 0

ในโหมดนี้ผู้ออกแบบ CPU Z80 ได้ออกแบบมาเพื่อให้ Z80 ทำการตอบสนองต่อสัญญาณ interrupt เหมือน 8080 ทุกประการ กล่าวคือเมื่อมีการ interrupt เกิดขึ้นและโปรแกรมทางซอฟต์แวร์ได้ set mode การรับ interrupt เป็น โหมด 0 (IM0) และenable การ interrupt ไว้การทำงานของ CPU จะหยุดการ fetch คำสั่ง ถัดไป แต่จะตอบรับการ interrupt ด้วยการส่งสัญญาณตอบรับด้วย $\overline{M1}$ กับ $\overline{IORQ}$ อ่านข้อมูล 1 byte เข้ามาทาง data bus ข้อมูล 1 byte นี้ได้รับการส่งมาจาก อุปกรณ์ I/O ที่ interrupt มา เมื่อ CPU อ่านข้อมูล byte นี้มา CPU จะถือว่าเป็น opcode ทันทีและจะตีความหมายในการทำงาน คำสั่งขนาด 1 byte ที่เหมาะในการใช้ในการinterruptก็คือคำสั่ง RST เมื่อ CPU execute คำสั่ง RST CPU จะเก็บข้อมูลเดิมใน PC ไว้ที่ stack แล้วเปลี่ยนค่า PC ใหม่ตามลักษณะของการ RST นั้นๆ ดังนั้น CPU จะกระโดดข้ามไปทำงานตามที่ต้องการของการ interrupt ได้ การตอบสนองต่อการ interrupt ด้วยการส่งสัญญาณ $\overline{M_1}$ และ $\overline{IORQ}$ ทำการ fetch ข้อมูลจาก I/O มา execute นี้อาจทำได้โดยการให้ I/O ส่งคำสั่งอื่นที่ไม่ใช่คำสั่ง RST (คำสั่ง byte เดียว) แต่เป็น คำสั่งหลาย byte เช่นคำสั่ง CALL ซึ่งเป็นคำสั่งขนาด 3 byte โดย I/O ส่งข้อมูลขณะตอบรับการ interrupt ($\overline{M_1} + \overline{IORQ}$) ด้วยข้อความ CDH (CALL) เมื่อ CPU execute คำสั่งนี้ก็จะทราบว่าเป็นคำสั่ง CALL ซึ่งยังต้องการข้อมูลเพิ่มอีก 2 byte CPU จะสร้าง machine cycle ต่อไปในการอ่านหน่วยความจำเหมือนกระทำคำสั่ง CALL จริง ๆ ในการนี้เราจะต้องใช้วงจรทางฮาร์ดแวร์ประกอบเพื่อให้ I/O ส่งข้อมูลอีก 2 byte ตามไปให้ได้ซึ่งการ interrupt ด้วยวิธีนี้อาจกำหนดได้ด้วยเทคนิคที่แตกต่างกันตามการออกแบบของแต่ละบุคคล

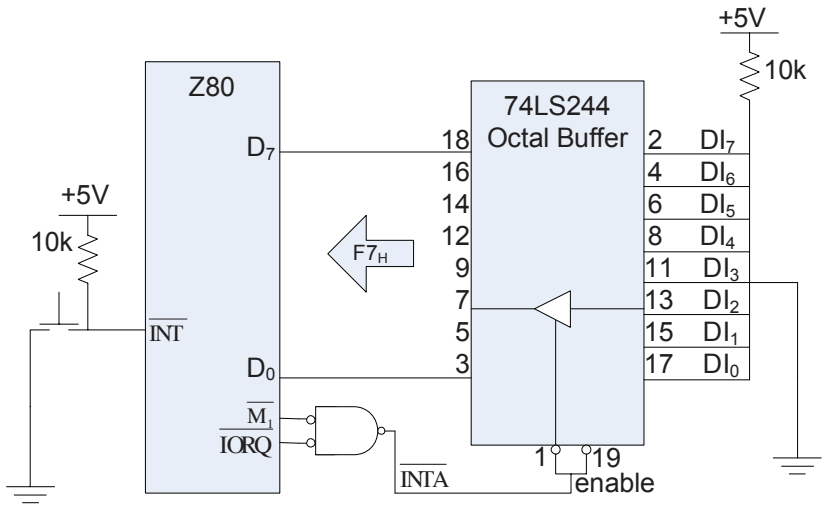
โดยปกติการ set ให้อยู่ในโหมดนี้กระทำได้ด้วยการสั่งทางซอฟต์แวร์ด้วยคำสั่ง IM1และการกำหนดให้รับการ interrupt ได้หรือไม่กระทำได้ด้วยคำสั่ง EI และ DI และเพื่อให้การทำงานเหมือน

อยู่ในโหมดของ 8080 ทุกประการ ดังนั้นหลังจากการ reset CPU CPU จะ set ตัวเองให้อยู่ในโหมด 0 นี้โดยอัตโนมัติ

ขั้นตอนการเกิด interrupt ในโหมด 0

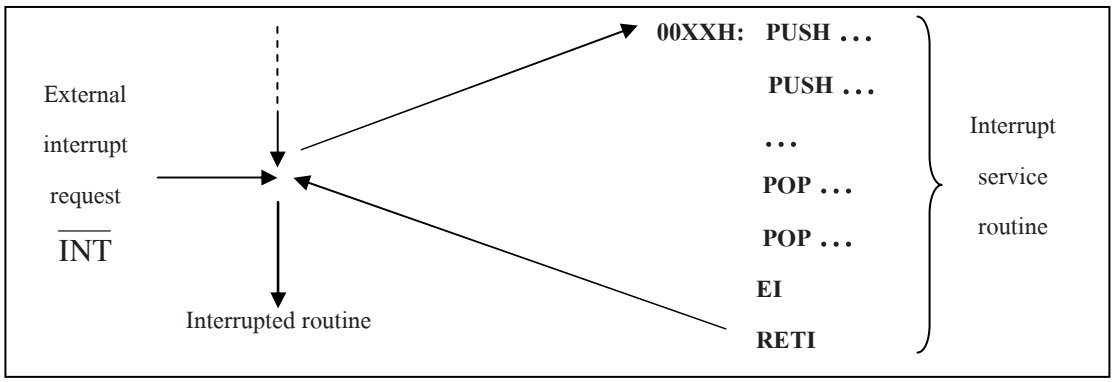
โหมด 0 เป็นวิธีการ interrupt ที่เหมือนกับ 8080 และ 8085

1. ใช้คำสั่ง IM 0
   EI
2. ในช่วงขาขึ้นของ T-state สุดท้ายของแต่ละคำสั่ง Z80 จะตรวจสอบสัญญาณ $\overline{INT}$
3. ถ้า $\overline{INT}$ active Z80 จะทำคำสั่งปัจจุบันจนเสร็จ แล้วทำการ disable IFF ทั้งสอง จากนั้นจึงส่งสัญญาณ $\overline{INTA}$ ($\overline{IORQ}$ และ $\overline{M1}$) โดยที่ไมโครโพรเซสเซอร์จะไม่รับ $\overline{INT}$ อีกจนกว่า IFF จะถูก enable ใหม่อีกครั้งหนึ่ง
4. ใช้สัญญาณ $\overline{INTA}$ เพื่อใส่คำสั่ง RST XXH ลงใน data bus ดังรูปที่ 8.8



รูปที่ 8.8 การใช้ Interrupt โหมด 0 (ที่มา: Gaonkar, 1992)

5. เมื่อ Z80 รับคำสั่ง RST XXH นี้ ก็จะเริ่มทำตามคำสั่ง โดยเก็บค่า PC ปัจจุบันไว้บน stack ก่อนจะไปเริ่มทำงานในตำแหน่งที่ระบุไว้ในคำสั่ง RST ดังรูปที่ 8.9

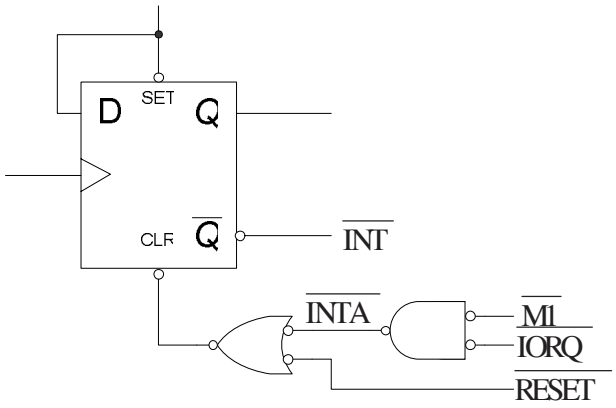


รูปที่ 8.9 การเก็บค่า PC ใน interrupt service routine

6. ปัญหาที่อาจเกิดขึ้นได้

- สัญญาณ $\overline{INT}$ สั้นเกินไป หรือมาไม่ถูกจังหวะ
- สัญญาณ $\overline{INT}$ นานเกินไป จะเกิด interrupt ซ้ำ เพราะว่า $\overline{INT}$ เป็นแบบ level sensitive

วิธีแก้ปัญหา คือการใช้ D Flip-Flop ดังในรูปที่ 8.10



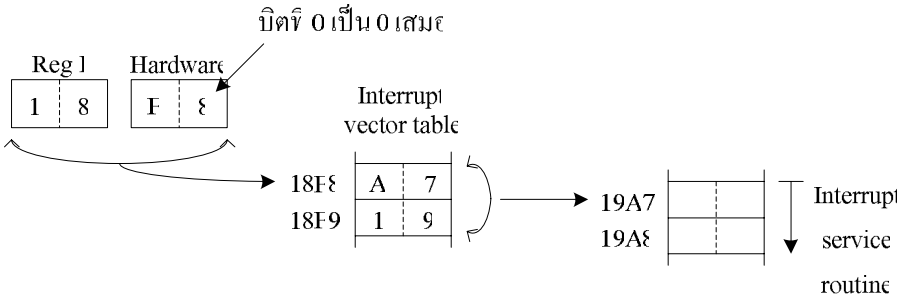
รูปที่ 8.10 การแก้ปัญหาโดยใช้ D Flip-Flop

8.6.2 การ interrupt โหมด 1

ในโหมดนี้ผู้โปรแกรมสามารถกำหนดได้ด้วยคำสั่ง IM1 การ interrupt ในโหมดนี้จะกระทำเหมือนกันกับ interrupt ทุกประการแต่จะแตกต่างกันก็เพียงการเริ่มการทำงาน (restart) มาที่ตำแหน่ง 0038_H (กรณี non-maskable ไปกระทำที่ 0066_H) และจำนวนคาบเวลาที่ใช้ใน โหมด 1 นี้มีมากกว่าใน non-maskable ทั้งนี้ เพราะในโหมดนี้ CPU ต้องเพิ่ม T_w ขึ้นอีก 2 states การ interrupt ในโหมดนี้สามารถ disable ได้ด้วยซอฟต์แวร์

8.6.3 การ interrupt โหมด 2

ในโหมดนี้ interrupt routine จะอยู่ที่ใดก็ได้ในหน่วยความจำ 0000_H – $FFFF_H$ โดยใช้ค่าที่เก็บใน register I และฮาร์ดแวร์ร่วมกัน เป็น pointer ชี้ไปยัง interrupt vector table ซึ่งเก็บ address ของ interrupt อีกทีหนึ่ง ดังตัวอย่างในรูปที่ 8.11



รูปที่ 8.11 ตัวอย่างของ interrupt โหมด 2

ในโหมดนี้ทำให้ Z80 มีขีดความสามารถเกี่ยวกับการ interrupt สูงขึ้นมาก การ interrupt ในโหมดนี้กำหนดได้ด้วยคำสั่ง IM 2 และการจะให้ CPU ตอบสนองหรือไม่ก็ยังใช้คำสั่งโปรแกรมได้เช่นกันโดยใช้คำสั่ง EI และ DI การกระโดดไปยังโปรแกรมอื่นในขณะเมื่อ CPU ตอบสนองต่อการ interrupt ในกรณีนี้จะไปที่ใดก็ได้ โดยใช้ addressในการกระโดดนี้ได้ถึง 16 บิต ซึ่งทำให้การ interrupt ทำได้สะดวกและรวดเร็วขึ้นอีกมาก

กรรมวิธีการตอบสนองต่อการ interrupt ในกรณีนี้คือ เมื่อมีสัญญาณ $\overline{INT}$ เข้ามาและ CPU ตรวจสอบได้ในตอนสุดท้ายของคำสั่ง CPU จะตอบสนองด้วยการส่ง $\overline{M1}$ กับ $\overline{IORQ}$ ออกไป สัญญาณ $\overline{M1}$ กับ $\overline{IORQ}$ จะเป็นตัวบอกอุปกรณ์ที่ส่ง $\overline{INT}$ มาให้ส่งข้อมูลขนาด 1 byte เข้าทาง data bus สำหรับในโหมดนี้ข้อมูลที่ส่งจาก I/O ขนาด 1 byte ที่เข้าทาง data bus นี้ CPU ถือว่าเป็น vector ของ การ interrupt โดยข้อมูลในบิต D_0 จะต้องเป็น "0" ส่วนบิตอื่นจะเป็นอะไรก็ได้ CPU จะนำ vector นี้ไปเป็นข้อมูล address byte ที่มีนัยสำคัญต่ำและข้อมูลจาก register I ภายใน CPU เป็นข้อมูล byte ที่มีนัยสำคัญสูง เรียกหาไปยังข้อมูลในหน่วยความจำเพื่ออ่านข้อมูลในหน่วยความจำ 2 bytes ติดกันนั้นมา load ใส่ PC หรือเป็นการอ้าง addressให้ PC แบบโดยทางอ้อม (Indirect addressing mode) นั่นเอง

สำหรับกรณีนี้จะเห็นว่าเราสามารถ set ค่าใน register I ให้เป็นอะไรก็ได้และ I/O จะส่งข้อมูล vector มาประกอบรวมเพื่อบอกถึงค่าตารางในหน่วยความจำที่ต้องการ ด้วยวิธีการเช่นนี้จะทำให้การกระโดดไปยังโปรแกรมย่อยเกิดขึ้นที่ใดก็ได้ อุปกรณ์ IC ที่ทำงานร่วมโดยใช้ interrupt mode นี้มีหลายเบอร์ด้วยกัน เช่น Z80PIO Z80CTC Z80SIO ฯลฯ ซึ่งอุปกรณ์ interface เหล่านี้สามารถส่ง vector ให้กับ CPU ได้อย่างมีประสิทธิภาพ การ interrupt ในโหมดนี้ CPU ต้องการเวลาถึง 19 states โดยใช้ 7 states ใน การ fetch ค่า vector ใช้ 6 states ถัดไปในการเก็บข้อมูล PC เดิม และใช้อีก 6 states ในการอ่านข้อมูลจากหน่วยความจำมายัง PC

ตัวอย่างการต่อประสาน A/D converter ใน interrupt โหมด 1

เมื่อเกิด interrupt ในโหมด 1 Z80 จะไปเริ่มต้นทำงานที่ตำแหน่ง 0038_H โดยไม่ต้องมีฮาร์ดแวร์เพิ่ม

<u>การทำงานของ ADC 0801</u>

- รับอินพุตที่ขา Vin (+) และ Vin (-)
- มี internal clock กำหนดความถี่โดยใช้ RC โดยที่ conversion time จะขึ้นอยู่กับความถี่ของสัญญาณนาฬิกา

การแปลงเริ่มจากการให้สัญญาณ $\overline{CS}$ และ $\overline{WR}$ (ใช้เป็น start signal) เมื่อแปลงเสร็จจะส่งสัญญาณ INTR ให้ Z80 Z80 จะมาเอาข้อมูลโดยให้สัญญาณ $\overline{CS}$ และ $\overline{RD}$ แล้วอ่านข้อมูลจาก data bus การอ่านข้อมูลจะทำเป็นการ reset การทำงานของ A/D โดยอัตโนมัติ (หยุดส่งสัญญาณ $\overline{INTR}$) โดยไม่ต้องใช้ฮาร์ดแวร์เพิ่ม

ตัวอย่างโปรแกรม

```
            ORG   0
            JP    START
            ORG   38H
            JP    ADC
START:      ORG   2000H
            LD    SP, 3000H
            LD    HL, BUFFER
            LD    B, จำนวนbyte
            IM    1
            EI
            OUT   (0F8H), A    ;เริ่มต้นการแปลงสัญญาณ
 WAIT:      HALT
            JP    NZ, WAIT
            …
 ADC:       IN    A, (0F8H)
            LD    (HL), A
            INC   HL
            OUT   (0F8H), A
            DEC   B
            EI
            RETI
BUFFER:     DFS   จำนวนbyte
```

8.7 Timing diagram สำหรับการตอบสนองต่อการ interrupt ของ CPU

การสนองต่อการ interrupt ของ CPU มีลักษณะสำคัญที่เราจะต้องพิจารณาในแง่ของไดอะแกรมเวลา เพื่อการเชื่อมต่อกับอุปกรณ์ I/O ในแง่ทางฮาร์ดแวร์จะได้เป็นไปอย่างถูกต้อง โดยการตอบสนองจะเกิดตามลำดับดังนี้

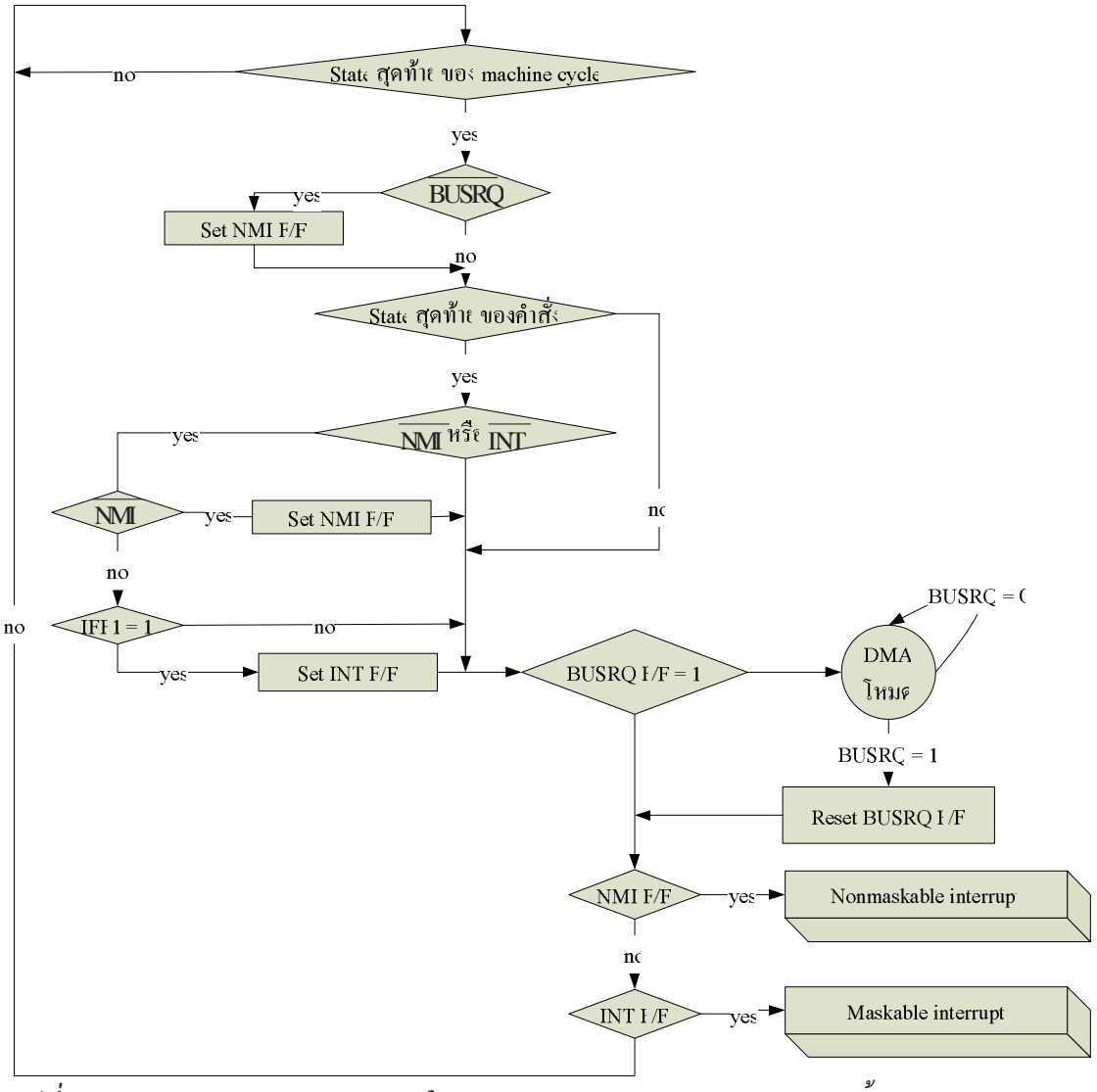
1. อุปกรณ์ I/O ส่งสัญญาณ interrupt มา โดยการทำให้ขา $\overline{\text{INT}}$ อยู่ในลอจิก "0"

2. CPU จะตอบสนองต่อการ interrupt ด้วยการส่ง $\overline{\text{M1}}$ ลอจิก "0" ก่อนเพื่อให้อุปกรณ์ I/O เตรียมจัดการเกี่ยวกับขบวนการจัดลำดับก่อน แล้ว $\overline{\text{IORQ}}$ จาก CPU จึงส่งตามมา การให้ $\overline{\text{IORQ}}$ มาทีหลัง $\overline{\text{M1}}$ ก็เนื่องจากให้ช่วงเวลา delay ระหว่างนี้เป็นตัวกำหนด IEI และ IEO ของการกระทำ daisy chain เมื่อ $\overline{\text{IORQ}}$ ออกไปที่ I/O และถ้า IEI ของอุปกรณ์นั้นเป็นลอจิก "1" ตัว I/O นั้นก็จะส่ง vector เข้ามาทาง data bus และ CPU จะใช้ $\overline{\text{IORQ}}$ และ $\overline{\text{M1}}$ เป็น pulse สั่งอ่านเข้าทาง data bus และการทำ daisy chain อุปกรณ์ที่ส่ง interrupt ได้ จะให้ IEO เป็นลอจิก "0" เพื่อป้อนเข้า IEI ของตัวอื่นเป็นการบล็อกการส่ง vector จาก I/O ตัวอื่น

3. การเคลียร์ interrupt โดยปกติอุปกรณ์ I/O จะ active ในการส่ง interrupt ได้ต้องให้ IEI = 1 และ IEO = 0 ดังนั้นเมื่อเสร็จสิ้นการกระทำการinterrupt แล้วสถานะของการ I/O จะต้อง เคลียร์ตัวเองได้ นั้นคือเมื่อมีการกระทำคำสั่ง RETI (ED 4D) ก็จะมีการบอกให้อุปกรณ์ I/O ทราบว่าจบสิ้นการ interrupt แล้ว เพื่อทำให้ IEO เป็น "1" เพื่อการenable ตัวI/O ที่มีความสำคัญน้อยกว่า ความสัมพันธ์ของ $\overline{\text{INT}}$ $\overline{\text{NMI}}$ และ $\overline{\text{BUSRQ}}$

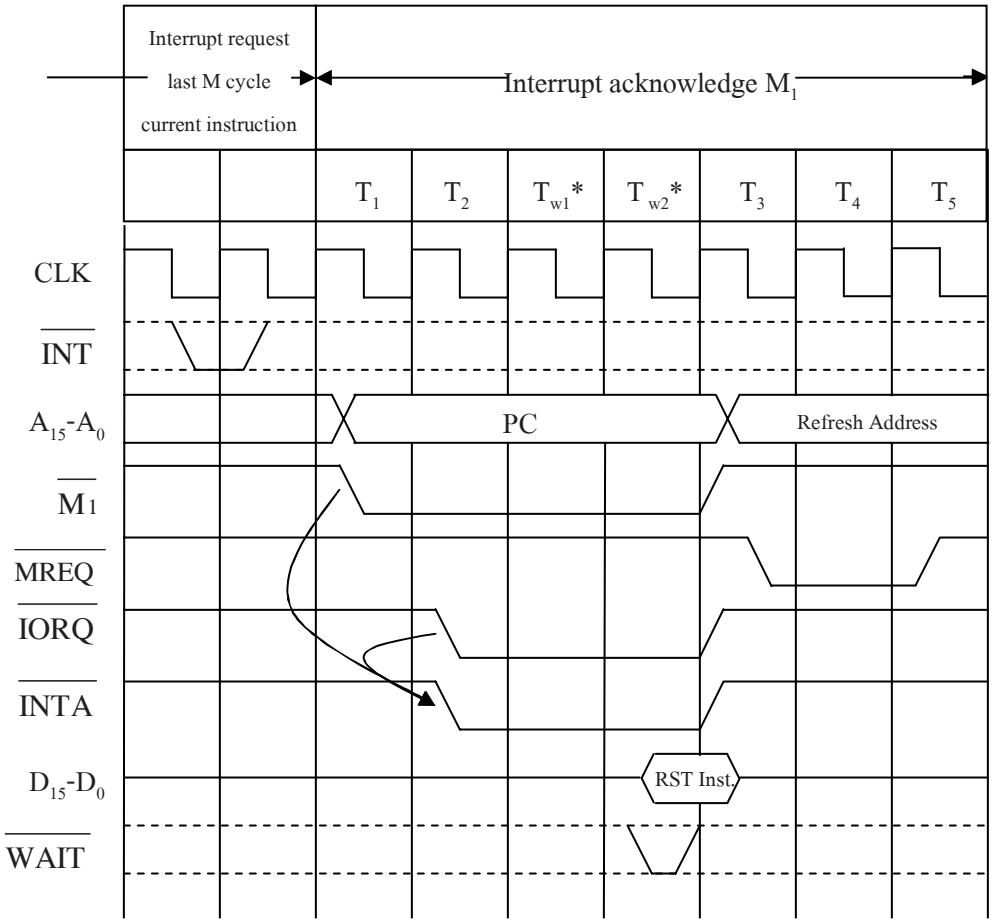
รูปที่ 8.12 แสดงการจัดลำดับการทำงานภายในสำหรับการตรวจสอบการ interrupt ทั้งแบบ INT และ NMI รวมทั้งขบวนการขอใช้บัส (BUSRQ) โดยมีลักษณะการทำงานที่น่าสนใจดังนี้

1. การตรวจสอบสัญญาณ $\overline{INT}$ และ $\overline{NMI}$ CPU จะทำการตรวจสอบทุกๆไซเคิลสุดท้ายของสัญญาณนาฬิกาของแต่ละคำสั่ง
2. สัญญาณ $\overline{BUSRQ}$ จะได้รับการตรวจสอบทุก ๆ ตอนสุดท้ายของ machine cycle
3. เมื่อ CPU อยู่ในสภาวะ DMA จะไม่ยอมรับรู้การขอ INT หรือ NMI
4. การจัดการตอบสนองจะเป็นไปตามลำดับดังนี้ $\overline{BUSRQ}$ $\overline{NMI}$ และ $\overline{INT}$



รูปที่ 8.12 การจัดลำดับการทำงานภายในสำหรับการตรวจสอบการ interrupt ทั้งแบบ INT และ NMI

Interrupt request and acknowledge cycle



* T_{w1} และ T_{w2} คือ wait state ซึ่ง Z80 จะใส่เพิ่มให้โดย

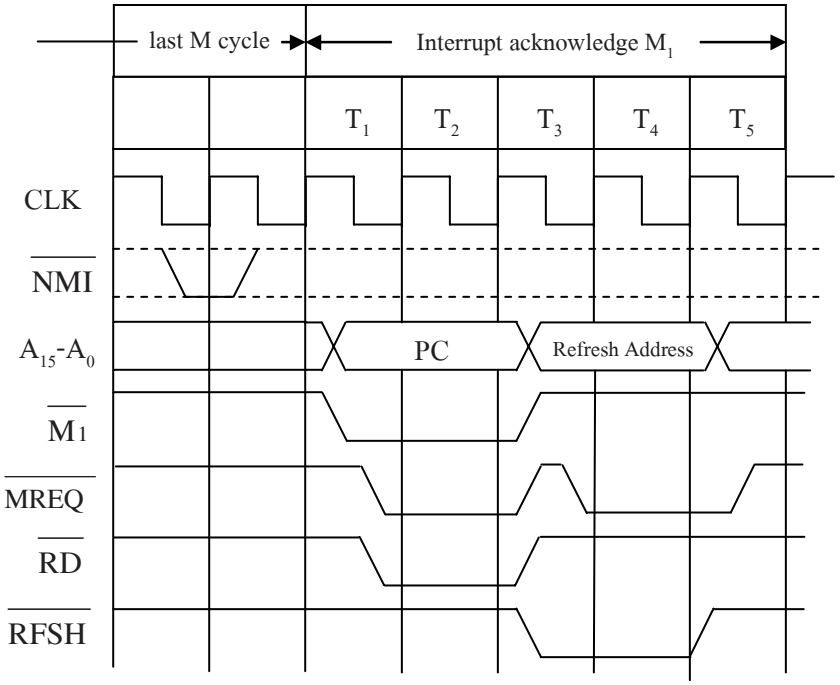
รูปที่ 8.13 Timing diagram ของการตอบสนองต่อการ interrupt (ที่มา: Gaonkar, 1992)

จากรูปที่ 8.13 เป็นการแสดง timing diagram ของการ interrupt โดยที่ CPU จะทำการตรวจสอบสัญญาณ $\overline{INT}$ ทุก ๆ pulse สัญญาณนาฬิกาลูกสุดท้ายของทุกคำสั่งที่กระทำแต่อย่างไรก็ตามการตรวจสอบ นี้จะไม่เกิดผลหากซอฟต์แวร์ได้สั่ง DI ซึ่งจะทำให้ interrupt flip-flop ภาย ในไม่ได้รับการเซท นอกจากนี้ สัญญาณ $\overline{BUSRQ}$ active การ interrupt ก็ไม่ได้รับการตอบสนอง เช่นกัน เมื่อการตอบสนองต่อ $\overline{INT}$ เกิดขึ้น CPU จะสร้างสถานะพิเศษใน $\overline{M1}$ ต่อมา ใน สถานะพิเศษของ $\overline{M1}$ นี้ $\overline{IORQ}$ จะ active แทน $\overline{MREQ}$ เพื่อเป็นการตอบสนองต่อการ interrupt และตอนท้ายของ $\overline{IORQ}$ CPU จะทำการอ่านข้อมูล 8 บิตเป็นเวคเตอร์สำหรับการ interrupt จาก data bus สังเกตว่าในกรณีนี้ CPU จะสร้างสัญญาณ T_w 2 states ขึ้น อย่าง อัตโนมัติ การสร้าง T_w ขึ้นมาก็เพื่อชะลอเวลาให้อุปกรณ์ภายนอกตรวจสอบตัวอุปกรณ์ที่สร้าง interrupt และ จัดลำดับความ สำคัญ (priority) ของสัญญาณที่ interrupt

เรายังสามารถเพิ่ม state การ $\overline{WAIT}$ ขึ้นอีกได้เพื่อให้ CPU รออุปกรณ์ที่จะส่ง vector ของการ interrupt ทาง data bus

Nonmaskable interrupt response

จากรูปที่ 8.14 เป็น diagram เวลาของการตอบสนองต่อการ interrupt แบบ non-maskable เมื่อถึง pulse ของสัญญาณนาฬิกาลูกสุดท้ายก่อนการกระทำการจบคำสั่ง แต่ละคำสั่ง CPUจะทำการตรวจสอบ ขา $\overline{NMI}$ และถ้า $\overline{NMI}$ มีค่า active (=0) CPU ก็จะ latch ไว้แล้วตอบสนองต่อการ interrupt ทันที การ interrupt ด้วยวิธีนี้ถือว่า CPU ให้ความสำคัญสูงสุด และขบวนการทาง ซอฟต์แวร์ก็ไม่สามารถที่จะป้องกันไม่ให้ CPU ตอบสนองได้ การตอบสนองจะเกิดขึ้นทันทีดังนั้นการ interrupt ด้วยวิธีนี้จึงนำมาใช้ในกรณีที่ สำคัญเป็นพิเศษ การ interrupt วิธีนี้ CPU จะส่งข้อมูล เดิมที่อยู่ใน program counter ไปเก็บไว้ที่ stack แล้วเซทค่า program counter เป็นค่า 0066_H เพื่อ กระทำในส่วนของโปรแกรมย่อยที่ตำแหน่ง 0066_H



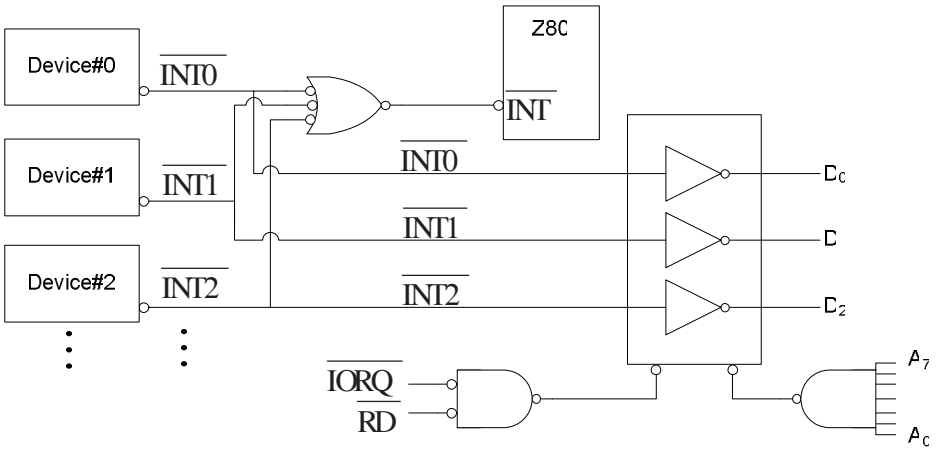
รูปที่ 8.14 Timing diagram ของการตอบสนอง Interrupt แบบ NMI (ที่มา: Gaonkar, 1992)

8.8 Multiple Interrupts และ Priorities

เราสามารถออกแบบวงจรให้อุปกรณ์ I/O หลายอุปกรณ์ขอขัดจังหวะการทำงานของ Z80 ได้ วิธีการที่ Z80 จะแยกแยะว่าอุปกรณ์ใดที่ขอ interrupt ทำได้ 2 วิธี คือ

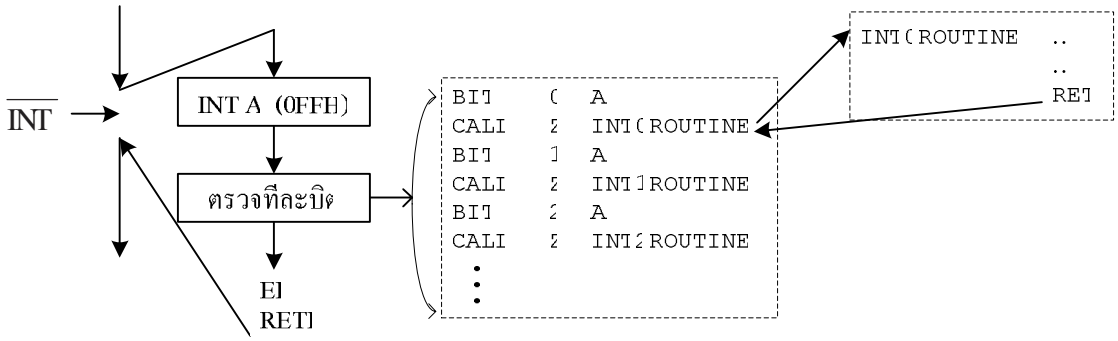
- polling method
- interrupt vector method

8.8.1 Polling Method



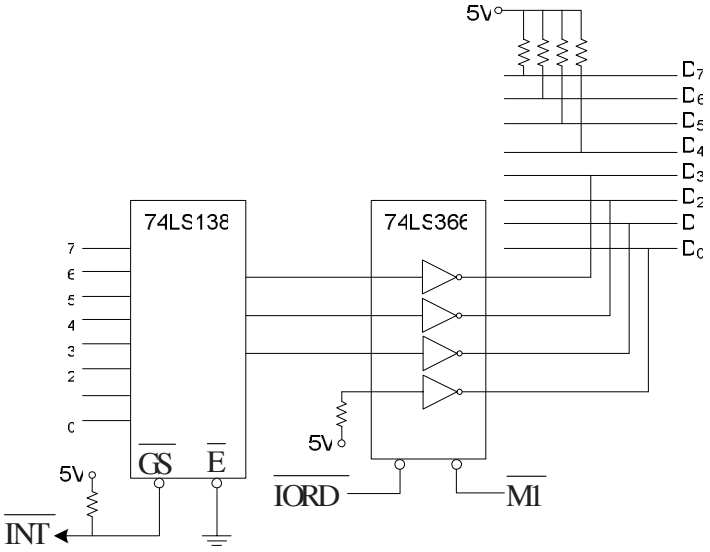
รูปที่ 8.16 Polling Method (ที่มา: Gaonkar, 1992)

กำหนดให้มีพอร์ตหนึ่ง รับอินพุตที่เป็นสัญญาณ $\overline{INTX}$ จากอุปกรณ์ I/O เราสามารถเขียนโปรแกรมให้ interrupt service routine ตรวจที่ตำแหน่งของ input port นี้ว่า มีสัญญาณ $\overline{INTX}$ จากอุปกรณ์ใด สัญญาณ $\overline{INTX}$ ใดที่ตรวจก่อน จะมี priority สูงกว่าโดยอัตโนมัติ



รูปที่ 8.17 การเขียนโปรแกรมตรวจบิตต่างๆของ Polling Method

<u>ตัวอย่างของการพัฒนา Polling Method</u>



รูปที่ 8.18 ตัวอย่างPolling Method (ที่มา: Gaonkar, 1992)

ส่วนของโปรแกมเป็นดังนี้

```
            ORG    0
            JR     START
            ORG    38H
            JP     ADC_INTR
START:      LD     SP, STACK
            LD     HL, DATA1
            LD     DE, DATA2
            IM     1
            EI
            OUT    (ADC1), A   } เริ่มการแปลง ADC
            OUT    (ADC2), A   }
MAINLOOP:   :
            :
            :
            JP     MAINLOOP
ADC_INTR:   PUSH AF
            IN     A, (0FFH)
            AND    3
            RRA
            CALL   C, DEVICE1
            RRA
            CALL   C, DEVICE2
            POP    AF
            EI
            RETI
DEVICE1:    PUSH   AF
            IN     A, (ADC1)
            LD     (HL), A
            OUT    (ADC1), A
            POP    AF
            RET
 DEVICE2:   PUSH   AF
            IN     A, (ADC2)
            LD     (DE), A
            OUT    (ADC2), A
            POP    AP
            RET
DATA1:      DFS    1
DATA2:      DFS    1
            END
```

8.8.2 Interrupt Vector Method

วิธีนี้ใช้ interrupt โหมด 2 วงจรในรูปที่ 8.19 ใช้ 74LS148 (8-to-3 Priority Encoder)

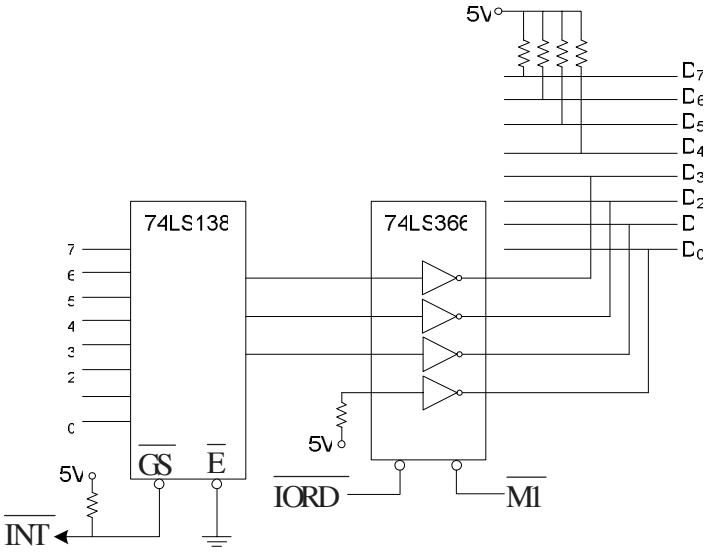
ถ้า input I_7 เป็น active low output $A_2 - A_0 = \bar{1}\ \bar{1}\ \bar{1} = 0\ 0\ 0$

ถ้า input I_4 เป็น active low output $A_2 - A_0 = \bar{1}\ \bar{0}\ \bar{0} = 0\ 1\ 1$

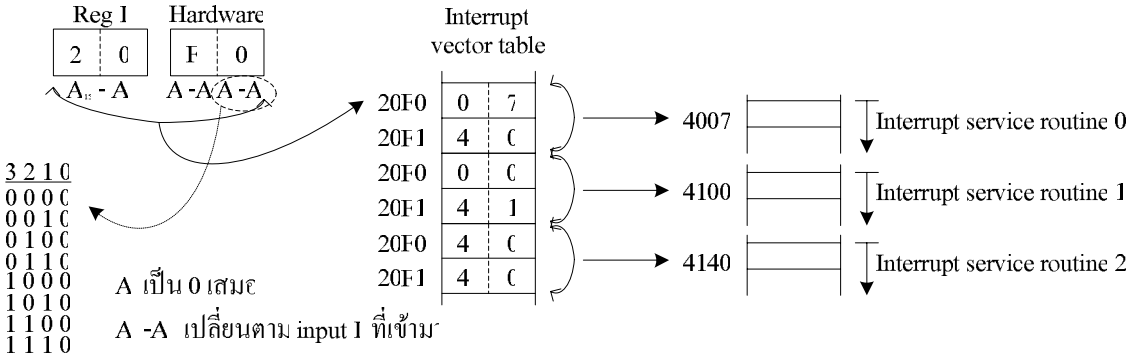
ถ้า I_7 และ I_4 เป็น active พร้อมกัน $A_2 - A_0 = \bar{1}\ \bar{1}\ \bar{1} = 0\ 0\ 0$

} ใช้ 74LS366 invert กลับมาอีกครั้ง

ในโหมด 2 นี้ Z80 จะรับ interrupt vector address เป็นเลขคู่ ดังนั้นเราจึงให้บิต D_0 เป็น 0



รูปที่ 8.19 Interrupt Vector Method (ที่มา: Gaonkar, 1992)



รูปที่ 8.20 Interrupt Vector Method

บทที่ 9 อุปกรณ์ต่อประสานที่สามารถโปรแกรมได้

อุปกรณ์ต่อประสานที่สามารถโปรแกรมได้ (Programmable Interface Devices) เป็นอุปกรณ์ที่มีหน้าที่ต่อประสานอุปกรณ์อินพุตหรือเอาท์พุต โดยที่หน้าที่ของอุปกรณ์ต่อประสานที่สามารถโปรแกรมได้นี้เปลี่ยนแปลงได้โดยการสั่งงานผ่าน Control register ที่อยู่ภายในอุปกรณ์ต่อประสานที่สามารถโปรแกรมได้

ในบทนี้จะกล่าวถึงอุปกรณ์ต่อประสานที่สามารถโปรแกรมได้ 2 ชนิดคือ

- Z80PIO
- 8255A

9.1 แนวคิดพื้นฐานของอุปกรณ์ต่อประสานที่สามารถโปรแกรมได้

การต่อประสานกับอุปกรณ์อินพุตหรือเอาท์พุตในบทที่ผ่านมา เป็นเพียงการต่อประสานอย่างง่ายโดยกระทำภายใต้สมมติฐานที่ว่า อุปกรณ์ I/O เหล่านั้นมีความพร้อมสำหรับการถ่ายโอนข้อมูลตลอดเวลา ซึ่งในความเป็นจริงสมมติฐานนั้นไม่ได้เป็นจริงเสมอไป

อุปกรณ์ I/O บางตัวจะทำงานช้ากว่า CPU มาก เช่น เครื่องพิมพ์แบบ dot matrix หรือแป้นพิมพ์ เพื่อป้องกันการสูญเสียข้อมูลหรืออ่านข้อมูลเดิมซ้ำ จำเป็นต้องมีสัญญาณ handshake ระหว่างอุปกรณ์ I/O กับ Programmable I/O(PIO) chip

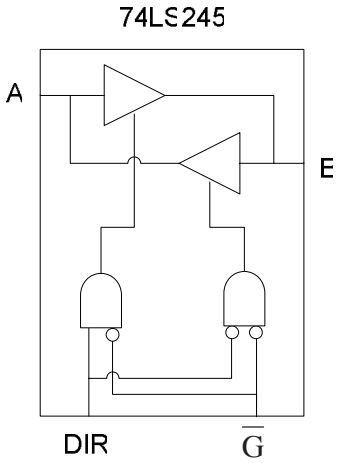
นอกจากนี้ อุปกรณ์ I/O บางตัวไม่สามารถสร้างสัญญาณ $\overline{\text{INT}}$ อุปกรณ์ I/O บางตัว อาจทำหน้าที่ทั้งรับและส่งข้อมูลได้ ดังนั้น programmable I/O device จะต้องมีสิ่งต่อไปนี้

1. Input และ Output registers (Latches สำหรับคงข้อมูลเอาไว้)
2. Tri-state buffers ที่มีความสามารถให้ข้อมูลผ่านได้ 2 ทิศทาง
3. Handshake และ interrupt signals
4. Control logic
5. Chip select logic
6. Interrupt Control logic

9.2 การสร้างตัวรับส่งที่สามารถโปรแกรมได้อย่างง่าย (Programmable Transceiver)

เพื่อให้เข้าใจถึงการเขียนโปรแกรมควบคุมอุปกรณ์ต่อประสานที่สามารถโปรแกรมได้ หัวข้อนี้อธิบายถึงการสร้างอุปกรณ์รับส่งอย่างง่ายซึ่งทำมาจาก buffer แบบ 2 ทิศทางที่สามารถโปรแกรมได้โดยใช้ชิป 74LS245 (bidirectional tri-state octal buffer)

ชิป 74LS245 ประกอบด้วย buffer 2 ทิศทางแบบ 3 สถานะจำนวน 8 ตัว โดยการไหลของข้อมูลภายใน buffer จะเกิดขึ้นเมื่อสัญญาณ $\overline{G}$ active (มีค่าเป็น 0) และ มีสัญญาณ DIR (direction) ทำหน้าที่กำหนดทิศทางการไหลของข้อมูล รูปที่ 9.1 แสดงถึงลอจิกของ buffer ภายในชิป 74LS245 เพียง 1 ตัว ซึ่งในความจริงแล้วนั้น ชิป 74LS245 ประกอบด้วย buffer ในลักษณะนี้ทั้งหมด 8 ตัว



รูปที่ 9.1 แผนภาพทางลอจิกของ buffer เพียง 1 ตัวภายในชิป 74LS245

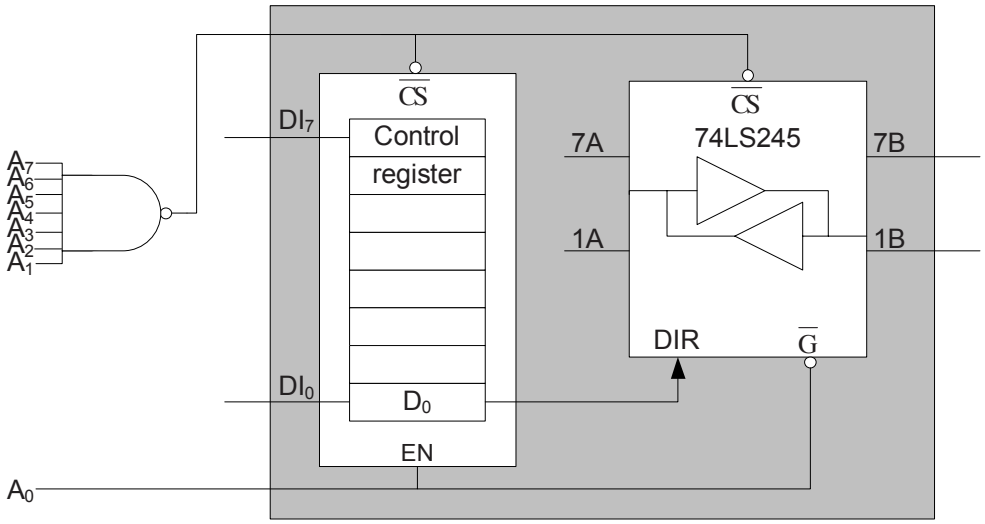
(ที่มา: Gaonkar, 1992)

การควบคุมการไหลของข้อมูลภายในชิป 74LS245 เป็นดังตารางที่ 9.1

$\overline{G}$	DIR	การไหลของข้อมูล
0	0	ข้อมูลไหลจาก B ไป A (buffer ทำหน้าที่เป็น input)
0	1	ข้อมูลไหลจาก A ไป B (buffer ทำหน้าที่เป็น output)
1	X	ไม่มีการไหลของข้อมูล (A และ B เป็น high impedance-Z)

ตารางที่ 9.1 รายละเอียดของค่าควบคุมต่างๆของชิป 74LS245

ในการนี้เนื่องจากเป็นการมุ่งเน้นไปยังการพัฒนาอุปกรณ์ที่สามารถโปรแกรมได้โดยการเขียนคำสั่ง ผ่านไมโครโพรเซสเซอร์ Z80 จึงจำเป็นต้องเพิ่ม *control register* เพื่อรองรับคำสั่งจากไมโครโพรเซสเซอร์โดยกำหนดโครงสร้างการพัฒนาระบบเป็นดังในรูปที่ 9.2



รูปที่ 9.2 ระบบการส่งรับข้อมูลอย่างง่ายโดยใช้ชิป 74LS245 (ที่มา: Gaonkar, 1992)

จากรูปที่ 9.2 ได้มีการใช้ A_7-A_1 จาก address bus ผ่าน NAND gate เพื่อไปเลือกใช้ control register และ ชิป 74LS245 ผ่านทาง $\overline{CS}$ (chip select) และได้มีการใช้ A_0 เพื่อแยกระหว่างตำแหน่งของ control register และ ชิป 74LS245 รายละเอียดเป็นดังในตารางที่ 9.2

$\overline{CS}$ (เพื่อเลือกใช้ control register และ 74LS245)							EN/$\overline{G}$	เลขฐาน 16	ตำแหน่งของ
A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0		
1	1	1	1	1	1	1	0	FE_H	port
1	1	1	1	1	1	1	1	FF_H	control register

ตารางที่ 9.2 รายละเอียดของตำแหน่งของ control register และชิป 74LS245

Control word คือค่าที่อยู่ใน control register สำหรับการกำหนดอุปกรณ์รับส่งนี้ให้ทำหน้าที่เป็นตัวรับข้อมูลหรืออินพุตคือ 00_H (D_0 = 0 และ D_7-D_1 เป็น “don’t care” หรือในที่นี้ให้เป็น 0) ส่วน control word สำหรับการกำหนดให้เป็นตัวส่งข้อมูลหรือเอาท์พุตคือ 01_H (D_0 = 1)

ชุดคำสั่งข้างล่างนี้เป็นการเขียนโปรแกรมเพื่อสั่งงานระบบในรูปที่ 9.2 เริ่มต้นโดยกำหนด buffer ให้เป็น input port เพื่อทำหน้าที่รับข้อมูลขนาด 1 byte

```
LD      A,00H           ;set control word ⟹ input port
OUT     (0FFH),A        ;write the control word to control register
IN      A, (0FEH)       ;get data via input port
```

ส่วนชุดคำสั่งต่อไปนี้เป็นการเขียนโปรแกรมเพื่อกำหนด buffer ให้เป็น output port ผ่านการส่ง control word 01_H ออกไปยัง control register ที่มีตำแหน่งอยู่ที่ FF_H เพื่อทำหน้าที่ส่งข้อมูล OutData ที่มีขนาด 1 byte ออกไปยัง output port ที่มีตำแหน่งอยู่ที่ FE_H

```
LD      A,01H           ;set control word ⇒ output port
OUT     (0FFH),A        ;write the control word to control register
LD      A,OutData       ;load output data
OUT     (0FEH),A        ;send output data via output port
```

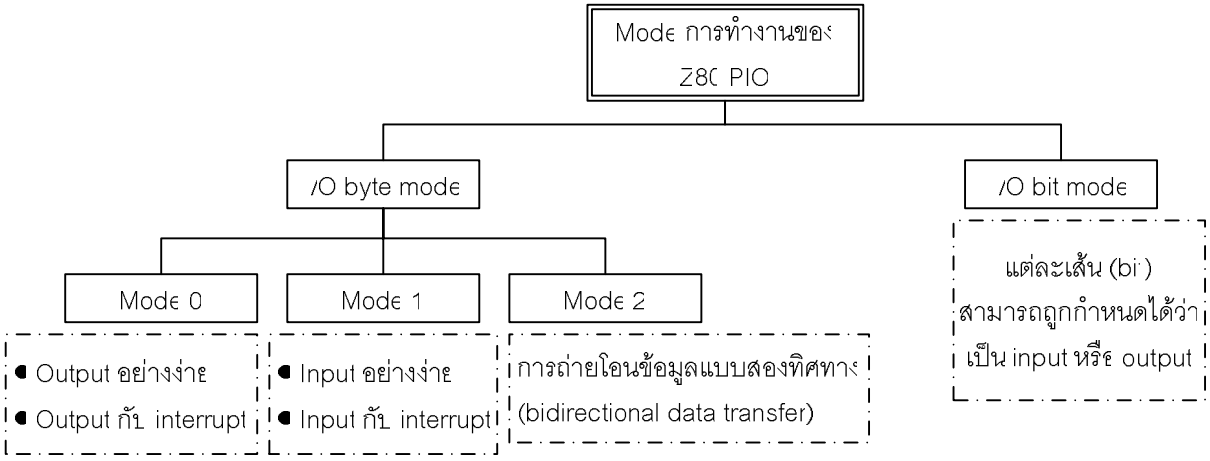
9.3 อุปกรณ์ I/O แบบขนานสำหรับ Z80

Z80 PIO *(Z80 Parallel I/O Device)* เป็นอุปกรณ์การต่อประสาน I/O ที่สามารถโปรแกรมได้ ซึ่งออกแบบมาเพื่อให้ใช้ร่วมกับไมโครโพรเซสเซอร์ Z80 โดยเฉพาะ

Z80 PIO มี port I/O จำนวน 2 พอร์ต (พอร์ต A และ B) แต่ละ port มีขนาด 8 บิต

9.3.1 โหมดการทำงานของ Z80 PIO

โหมดการทำงานของ Z80 PIO มีดังนี้



รูปที่ 9.3 โหมดการทำงานของ Z80 PIO

Mode 0: โหมดเอาท์พุตโดยที่มีอยู่ด้วยกัน 2 ลักษณะคือ

- แบบเอาท์พุตอย่างง่าย ไม่มีการใช้สัญญาณ handshaking
- แบบมีการใช้ร่วมกับ interrupt โดยมีการใช้สัญญาณ handshaking (สัญญาณ strobe และ ready)

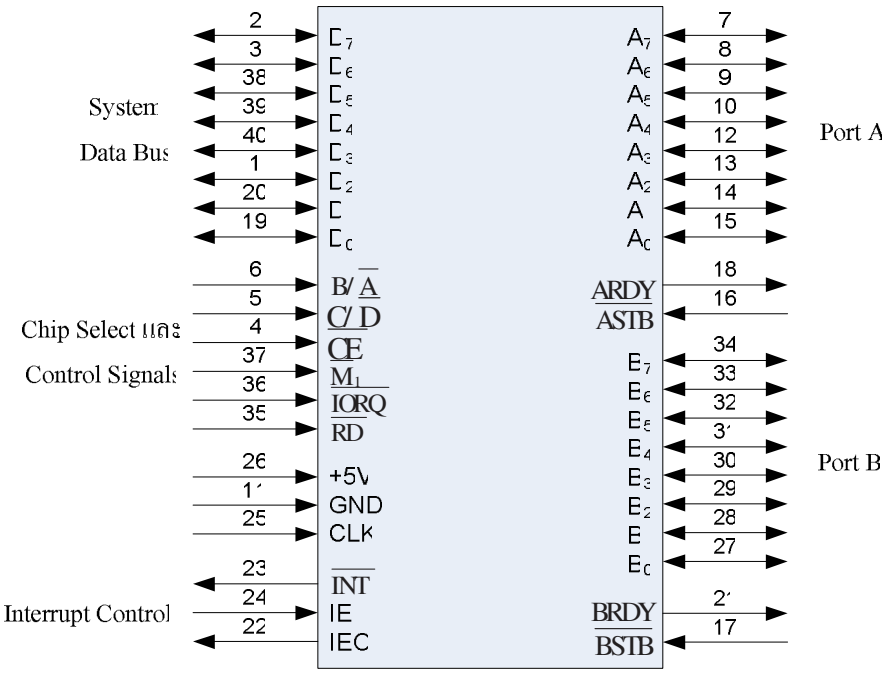
Mode 1: โหมดอินพุตโดยที่มีอยู่ด้วยกัน 2 ลักษณะคือ

- แบบอินพุตอย่างง่าย ไม่มีการใช้สัญญาณ handshaking
- แบบมีการใช้ร่วมกับ interrupt โดยมีการใช้สัญญาณ handshaking (สัญญาณ strobe และ ready)

Mode 2: โหมด 2 ทิศทางคือสามารถกำหนดให้เป็นอินพุตหรือเอาท์พุตก็ได้

Mode 3: โหมดบิต คือสามารถกำหนดแต่ละบิตของทั้งสอง port ให้ทำหน้าที่เป็นอินพุตหรือเอาท์พุตได้ โหมดนี้ใช้สัญญาณ handshaking ไม่ได้

9.3.2 กลุ่มสัญญาณของ Z80 PIO



รูปที่ 9.4 Z80 PIO Logic Signals (ที่มา: Gaonkar, 1992)

จากรูปที่ 9.4 แสดงถึงกลุ่มสัญญาณทั้ง 6 กลุ่มของ Z80 PIO ซึ่งมีคำอธิบายดังนี้

1. **Data bus (D_7 - D_0)**: เป็น data bus แบบ 2 ทิศทาง 3 สถานะ ขนาด 8 บิต ใช้ในการโอนย้ายข้อมูลระหว่าง Z80 MPU กับ PIO
2. **สัญญาณ I/O**: เป็นสายสัญญาณ I/O แบบ 2 ทิศทาง 3 สถานะ ใช้ในการโอนย้ายข้อมูลระหว่าง PIO กับอุปกรณ์รอบข้าง (peripheral) มีอยู่ด้วยกัน 2 ชุดโดยแต่ละชุดมีขนาด 8 บิตสำหรับพอร์ตแต่ละพอร์ต
    - **Port A: A_7 - A_0**
    - **Port B: B_7 - B_0**
3. **สัญญาณ handshaking**: ใน PIO มีสัญญาณ handshaking ทั้งหมด 4 สัญญาณ โดยแต่ละพอร์ตจะสัญญาณ ใช้ได้แก่
    - $\overline{\text{ASTB}}$ **(สัญญาณ stroke ของ พอร์ต A)** - เป็นสัญญาณอินพุตของพอร์ต A โดยรับมาจากอุปกรณ์รอบข้างเข้าไปยัง PIO โดยที่
        i. ถ้ากำหนดให้พอร์ต A เป็น output port อุปกรณ์รอบข้างจะส่งสัญญาณมายัง $\overline{\text{ASTB}}$ เพื่อแสดงว่าได้รับข้อมูลเรียบร้อยแล้ว
        ii. แต่ถ้ากำหนดให้พอร์ต A เป็น input port อุปกรณ์รอบข้างจะส่งสัญญาณมายัง $\overline{\text{ASTB}}$ เพื่อแสดงว่าได้ส่งข้อมูลมาเรียบร้อยแล้ว

- **ARDY (สัญญาณ ready ของ port A)** - เป็นสัญญาณเอาท์พุตของ port A ส่งออกจาก PIO ไปยังอุปกรณ์รอบข้าง
  i. ถ้ากำหนดให้ port A เป็น output port PIO จะใช้สัญญาณ ARDY นี้เพื่อบอกอุปกรณ์รอบข้างว่าข้อมูลพร้อมแล้วมาเอาไปได้
  ii. แต่ถ้ากำหนดให้ port A เป็น input port PIO จะใช้ สัญญาณ ARDY นี้ในการบอกอุปกรณ์รอบข้างว่าได้รับข้อมูลแล้วส่งมาใหม่ได้
- $\overline{\text{BSTB}}$ และ **BRDY (สัญญาณ stroke และ ready ของ port B)** - เป็นสัญญาณ handshake ของ port B โดยทั่วไปทำหน้าที่คล้ายกับสัญญาณ $\overline{\text{ASTB}}$ และ ARDY ของ port A ยกเว้นในกรณีที่ port A ถูกใช้ใน mode 2 ซึ่งเป็นโหมดที่ port A ทำหน้าที่รองรับการโอนย้ายข้อมูลแบบ 2 ทิศทาง ซึ่งในโหมดนั้นสัญญาณ $\overline{\text{BSTB}}$ และ BRDY จะถูกใช้ร่วมกับ port A

4. **ไฟเลี้ยงและสัญญาณนาฬิกา**
5. **Interrupt control logic**: สัญญาณที่ใช้ในการควบคุม interrupt มีอยู่ด้วยกัน 3 สัญญาณ ได้แก่
   - $\overline{\text{INT}}$ **(Interrupt)** - เป็นสัญญาณส่งออกจาก PIO ใช้ในการ interrupt Z80
   - **IEI (Interrupt Enable In)** – เป็นสัญญาณอินพุตใช้เมื่อต่อ Z80 PIO พ่วงเข้าด้วยกันแบบ daisy chain โดยจะใช้ร่วมกับสัญญาณ IEO รายละเอียดจะอธิบายในภายหลัง
   - **IEO (Interrupt Enable Out)** - เป็นสัญญาณเอาท์พุตใช้ร่วมกับสัญญาณ IEI ในการต่อ interrupt แบบ daisy chain
6. **สัญญาณควบคุม**: ประกอบด้วย 6 สัญญาณ แต่เมื่อแบ่งตามหน้าที่การทำงานแล้วมีอยู่ด้วยกันเพียง 2 ประเภทคือ
   - สัญญาณควบคุมที่ใช้ในการกำหนดตำแหน่งของ data port และ control register ของทั้ง port A และ B (ตารางที่ 9.3) สัญญาณประเภทนี้ได้แก่

$\overline{\text{CE}}$	C/$\overline{\text{D}}$	B/$\overline{\text{A}}$	พอร์ตที่เลือก
0	0	0	Data Port A
0	0	1	Data Port B
0	1	0	Control Register A
0	1	1	Control Register B
1	X	X	PIO ไม่ได้ถูกเลือก

ตารางที่ 9.3 สรุปการเลือก port ของ Z80 PIO

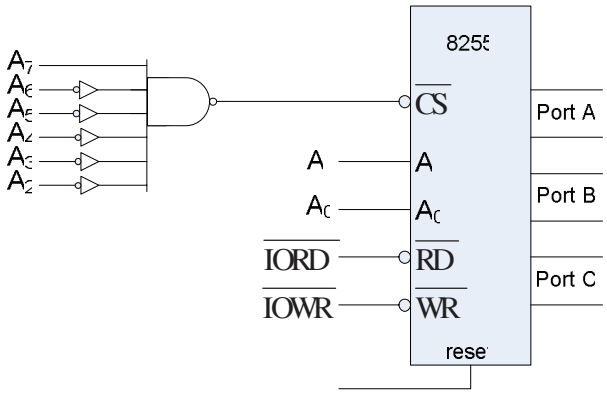
$\overline{CE}$ (Chip enable) - ใช้ในการเลือกชิป

C/$\overline{D}$ - ใช้ในการเลือกว่าเป็น control register หรือ data port

B/$\overline{A}$ - ใช้ในการเลือกว่าเป็นของ port B หรือ A

- สัญญาณควบคุมที่ใช้ในการกำหนดประเภทของการทำงาน อันได้แก่สัญญาณ $\overline{M_1}$, $\overline{RD}$ และ $\overline{IORQ}$ โดยสัญญาณเหล่านี้จะถูกต่อกับ Z80 โดยตรง (สังเกตว่า Z80 PIO ไม่มีสัญญาณ $\overline{WR}$) โดยรายละเอียดในการกำหนดการทำงานเป็นดังนี้
  - การอ่าน: Z80 จะอ่านข้อมูลจาก register ที่เลือกไว้ เมื่อสัญญาณ $\overline{RD}$ และ $\overline{IORQ}$ active
  - การเขียน: Z80 จะเขียนข้อมูลลงใน register ที่เลือกไว้เมื่อสัญญาณ $\overline{IORQ}$ active อย่างเดียวโดยที่สัญญาณ $\overline{RD}$ ไม่ active (เป็นเงื่อนไขที่ว่าสัญญาณ $\overline{RD}$ และ $\overline{WR}$ ของ Z80 จะไม่ active พร้อมกัน นั่นหมายความว่าในกรณีที่เป็นการเขียนข้อมูล สัญญาณ $\overline{RD}$ จะไม่ active)
  - การตอบสนองการ interrupt (Interrupt Acknowledge): Z80 จะตอบสนองการ interrupt จาก PIO เมื่อ สัญญาณ $\overline{M_1}$ และ $\overline{IORQ}$ active
  - การ reset: PIO จะได้รับการ reset เมื่อสัญญาณ $\overline{M_1}$ active ในขณะที่สัญญาณ $\overline{RD}$ และ $\overline{IORQ}$ ไม่ active

ตัวอย่างการ decode address ให้กับ Z80 PIO



รูปที่ 9.5 วงจรการต่อประสาน Z80 PIO กับไมโครโพรเซสเซอร์ Z80 (ที่มา: Gaonkar, 1992)

จากรูปที่ 9.5 จะได้ว่า

	A_7	A_6	A_5	A_4	A_3	A_2	A_1 C/$\overline{D}$	A_0 B/$\overline{A}$	
Data port A	1	0	0	0	0	0	0	0	$=80_H$
Data port B							0	1	$=81_H$
Control register A							1	0	$=82_H$
Control register B							1	1	$=83_H$

ในการจัดเตรียมการทำงานต่างๆ ของ Z80 PIO จะต้องเขียนคำสั่งการควบคุมต่างๆ ไปที่ control register ดังนี้

Control Word สำหรับควบคุมโหมดการทำงานของ Z80 PIO ซึ่งผู้ผลิตกำหนดมาให้แล้วนั้นมีรูปแบบตามรูปที่ 9.6

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Mode		X	X	1	1	1	1
ใช้ในการเลือก mode การทำงาน 00: mode 0 01: mode 1 10: mode 2 11: mode 3		Don't care		ชี้ให้เห็นว่าเป็น control word			

รูปที่ 9.6 Control word ของ Z80 PIO

Interrupt Enable Word เป็นการ enable flip-flop ของพอร์ทที่ใช้ โดยมีรูปแบบตามรูปที่ 9.7

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1/0	X	X	X	0	0	1	1
1: enable port interrupt 0: disable port interrupt	ใช้สำหรับโหมด 3			ชี้ให้เห็นว่าเป็น interrupt enable word			

รูปที่ 9.7 Interrupt enable word ของ Z80 PIO

Interrupt vector เป็นการกำหนด byte ล่างของ interrupt vector ซึ่งแสดงถึงตำแหน่ง service routine โดยมีรูปแบบตามรูปที่ 9.8

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
ใช้สำหรับการกำหนด บิต							0

D_7 - D_0 เป็นการกำหนด byte ล่างของ interrupt vector

โดยที่มี $D_0 = 0$ ชี้ให้เห็นว่าเป็น interrupt vector

รูปที่ 9.8 Interrupt vector ของ Z80 PIO

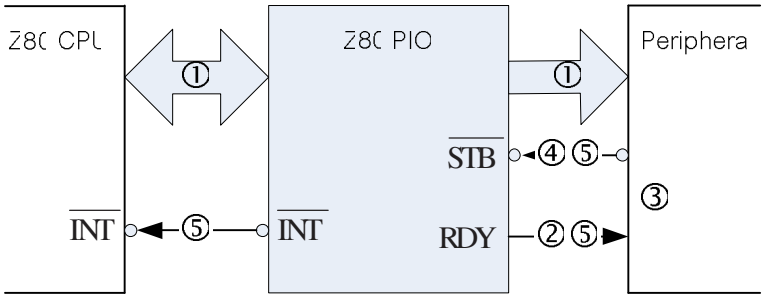
9.3.3 โหมด 0 1 และ 2 พร้อมด้วยสัญญาณ handshake และ Interrupt I/O

Z80 PIO สามารถ interrupt Z80 CPU ได้ ก็ต่อเมื่อเราได้เขียนคำสั่งให้ enable interrupt flip-flop ภายใน Z80 PIO ด้วย

9.3.3.1 Output Mode 0 พร้อมด้วยสัญญาณ handshake

ลำดับเหตุการณ์ที่จะเกิดขึ้นเป็นดังนี้

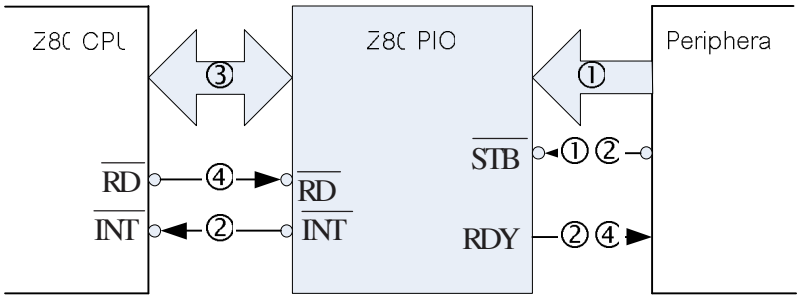
1. เมื่อ Z80 ทำคำสั่ง OUT ข้อมูลจะถูกส่งจาก Z80 CPU ไปปน data bus หลังจากนั้น Z80 PIO ส่งข้อมูลต่อไปยังอุปกรณ์ภายนอก
2. Z80 PIO ส่งสัญญาณ RDY ไปยังอุปกรณ์ภายนอก (peipheral) ว่าข้อมูลมาแล้ว
3. อุปกรณ์ภายนอกรับข้อมูลไป
4. อุปกรณ์ภายนอกส่งสัญญาณไปยัง $\overline{STB}$ เพื่อบอกว่าได้รับข้อมูลเรียบร้อยแล้ว
5. เมื่อ Z80 PIO ได้รับขอบขาขึ้นของสัญญาณ $\overline{STB}$ ก็จะหยุดส่งสัญญาณ RDY ต่อจากนั้น Z80 PIO จะส่งสัญญาณ $\overline{INT}$ ไปยัง Z80 CPU เพื่อให้ส่งข้อมูลถัดไปมาให้



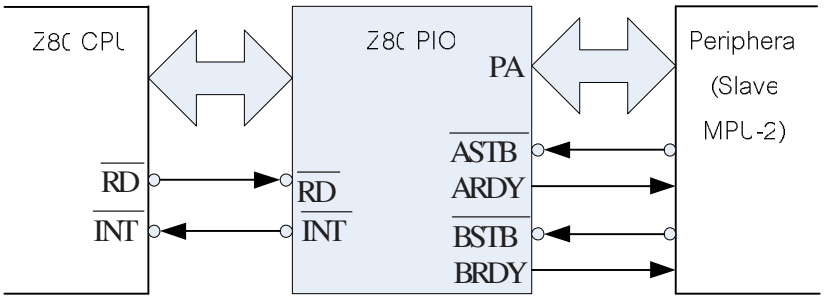
9.3.3.2 Input Mode 1 พร้อมด้วยสัญญาณ handshake

ลำดับเหตุการณ์ที่จะเกิดขึ้นเป็นดังนี้ (RDY เริ่มจาก active)

1. อุปกรณ์ภายนอกส่งข้มูลมายัง Z80 PIO และให้สัญญาณ $\overline{STB}$ ออกมาด้วย
2. ขอบขาขึ้นของสัญญาณ $\overline{STB}$ จะทำให้สัญญาณ RDY ไม่ active เพื่อบอกอุปกรณ์ภายนอกว่าได้รับข้อมูลแล้วและส่งสัญญาณ $\overline{INT}$ ให้ CPU
3. ใน interrupt routine ต้องอ่านข้อมูลนี้ไป
4. เมื่อมีสัญญาณ $\overline{RD}$ มายัง Z80 PIO เพื่ออ่านข้อมูล สัญญาณ RDY จะกลับมา ใหม่หลังจากขอบขาขึ้นของสัญญาณ $\overline{RD}$



9.3.3.3 Mode 2: Bidirectional Data Transfer



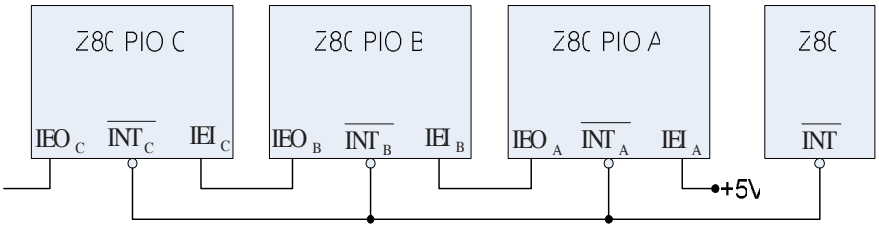
โดยที่ PA – เป็น bidirectional

PB – ทำงานในโหมด 3

ARDY, $\overline{\text{ASTB}}$ - เป็นสัญญาณ handshake สำหรับ output

BRDY, $\overline{\text{BSTB}}$ - เป็นสัญญาณ handshake สำหรับ input

9.3.3.4 Interrupt Piority



ลำดับความสำคัญ (Priority): PIO A > PIO B > PIO C (ต่อพ่วงกันได้ไม่เกิน 4 ตัว)

เมื่อ PIO ตัวที่ interrupt ได้ตรวจสอบว่า Z80 CPU ทำคำสั่ง RETI แล้วจะหยุด disable interrupt PIO ที่มีลำดับความสำคัญต่ำกว่า

IEI เป็น high – ไม่มี PIO อื่นที่มีลำดับความสำคัญสูงกว่ากำลัง service จาก Z80 CPU

เป็น low – มี PIO อื่นที่มีลำดับความสำคัญสูงกว่ากำลัง service จาก Z80 CPU และ PIO ตัวนี้จะขัดจังหวะม่ได้

IEO ธรรมดาจะเป็น high หรือ low ตาม IEI แต่ถ้า IEI เป็น high และ IEO เป็น low แสดงว่า PIO นี้ กำลัง service อยู่

9.3.4 Mode 3: Bit Mode

ลักษณะพิเศษของโหมดนี้คือ

1. แต่ละบิตของ PA และ PB ถูกกำหนดเป็น input หรือเอาท์พุตก็ได้ โดยการเขียนคำสั่งไปที่ control register
2. ไม่มีการใช้สัญญาณ handshake (RDY เป็น low และ $\overline{\text{STB}}$ ถูก disable)
3. เราสามารถกำหนด bit combination สำหรับ input เพื่อทำให้เกิด interrupt ได้ โดยเราสามารถเลือก logic combination (AND/OR) กับบางบิตและเลือกว่าจะเป็น active high หรือ active low ได้ด้วย

โดยในโหมด 3 นอกจากจะต้องเขียน control word ลง control register ของพอร์ตที่ใช้แล้วนั้น ยังต้องเขียน word อื่นๆที่ใช้ในการควบคุมอีก 3 words คือ

- I/O Register Control Word
- Interrupt Control Word
- Mask Control Word

ลง control register อีกด้วยตามรูปแบบดังต่อไปนี้

I/O Register Control Word เพื่อกำหนดว่าแต่ละ bit ทำหน้าที่เป็น input หรือเอาท์พุตโดยมีรูปแบบตามรูปที่ 9.9

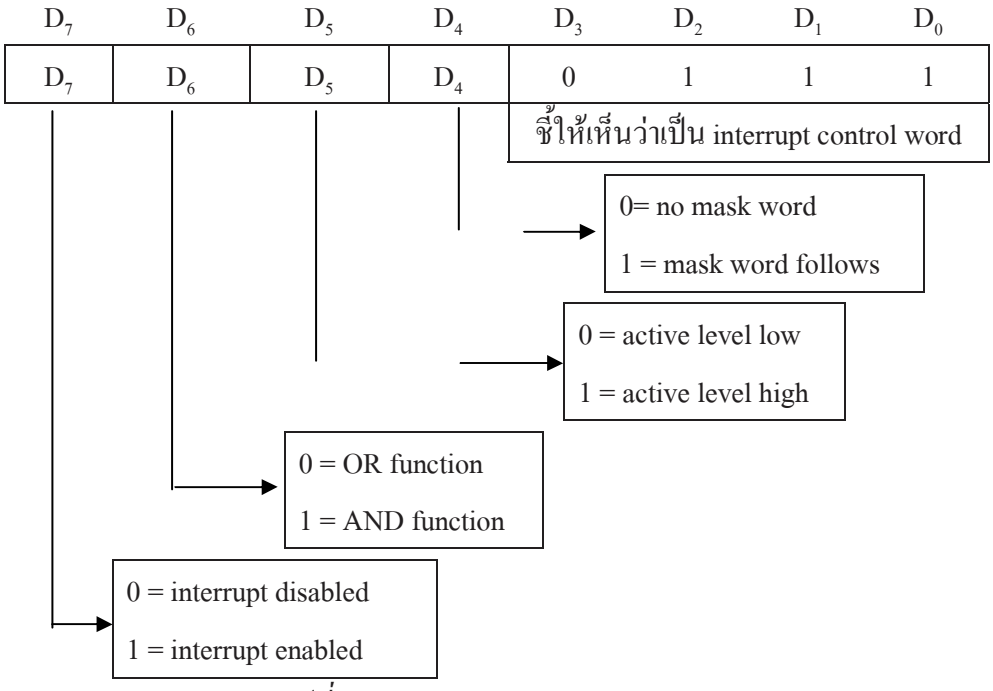
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

0= output

1 = input

รูปที่ 9.9 I/O register control word ของ Z80 PIO

Interrupt Control Word เพื่อกำหนดสภาวะตรรกะที่ใช้ในการสร้าง interrupt โดยมีรูปแบบตามรูปที่ 9.10



รูปที่ 9.10 Interrupt control word ของ Z80 PIO

Mask Control Word เพื่อกำหนดว่าบิตใดบ้างที่ถูกตรวจสอบ โดยมีรูปแบบตามรูปที่ 9.11

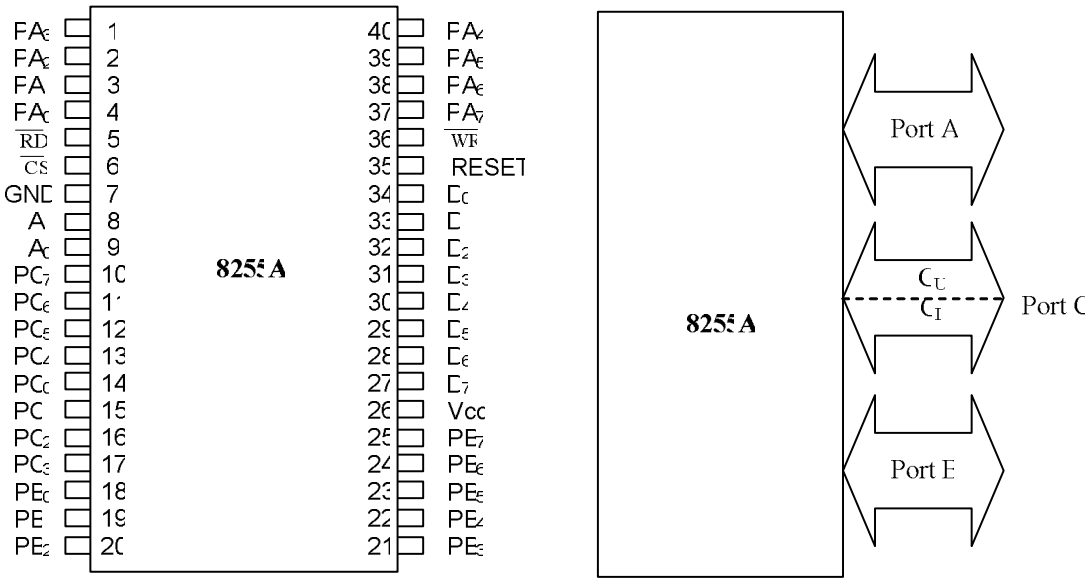
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

=0 $\Longrightarrow$ บิตนั้นจะถูกตรวจสอบเวลารับค่าเข้ามา

รูปที่ 9.11 Mask control word ของ Z80 PIO

9.4 ชิป 8255A

ชิป 8255A เป็นอุปกรณ์การต่อประสาน I/O แบบขนานที่สามารถโปรแกรมได้อีกประเภทหนึ่งที่ใช้งานทั่วไปกับไมโครโพรเซสเซอร์ได้หลากหลายรุ่น รูปที่ 9.12 แสดงการจัดวางขาของชิป 8255A



รูปที่ 9.12 8255A Pin layout (ที่มา: Gaonkar, 1992)

ชิป 8255A มี I/O ports ด้วยกันทั้งหมด 3 ports (port A, B และ C) แต่ละ port มีขนาด 8 บิต โดยที่ port C มีคุณลักษณะพิเศษคือ แบ่งเป็น port ย่อยขนาด 4 บิต จำนวน 2 ports คือ port C_U และ C_L ดังแสดงในรูปที่ 9.12 ทางขวามือ

Port A และ B ของชิป 8255A จะคล้ายกับ port A และ B ของ Z80 PIO ส่วน port C ของชิป 8255A จะคล้ายกับบิตโหมดของ Z80 PIO

ส่วนคำอธิบายของขาสัญญาณต่างๆของ 8255A นั้นแสดงในตารางที่ 9.4

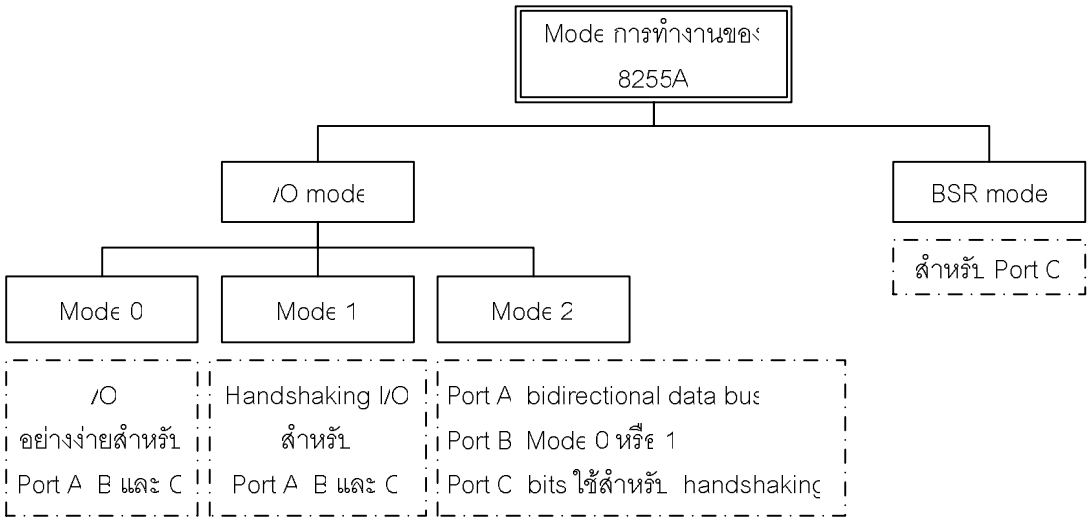
ขาสัญญาณ	คำอธิบาย
D_7 - D_0	Data bus แบบ 2 ทิศทาง
RESET	Reset input
$\overline{CS}$	Chip select
$\overline{RD}$	Read input
$\overline{WR}$	Write input
A_1 และ A_0	Port address
PA_7 - PA_0	Port A (bit)
PB_7 - PB_0	Port B (bit)

PC_7 - PC_0	Port C (bit)
V_{CC}	+ 5 Volts
GND	0 Volt

ตารางที่ 9.4 คำอธิบายของขาสัญญาณของ 8255A

ชิป 8255A มีโหมดการทำงานอยู่ด้วยกัน 2 โหมดคือ BSR mode และ I/O mode โดยที่ I/O mode ยังแบ่งได้เป็น 3 โหมดย่อยได้แก่ โหมด 0 โหมด 1 และ โหมด 2 รายละเอียดของโหมดการทำงานทั้งหมดของ 8255A มีดังนี้ (รูปที่ 9.13)

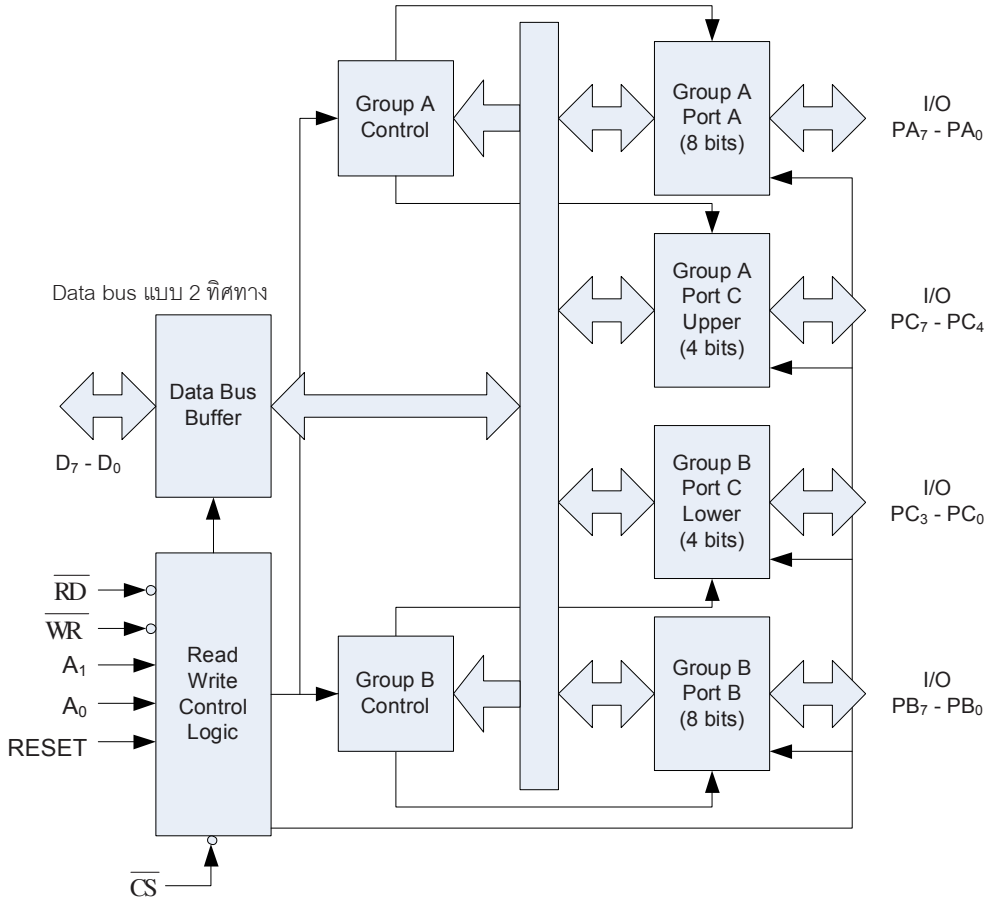
- **BSR (Bit Set Reset) mode**: ใช้เพื่อกำหนดบิต (set/reset) ใน port C
- **I/O mode**: แบ่งได้เป็น 3 โหมดย่อยได้แก่
  - **Mode 0**: ports ทั้งสาม (พอร์ต A B และ C) ทำหน้าที่เป็น I/O port อย่างง่าย
  - **Mode 1**: การโอนย้ายข้อมูลเป็นแบบ handshake สำหรับพอร์ต A และ B โดยที่ใช้บิตจากพอร์ต C เป็นสัญญาณ handshake
  - **Mode 2**: ใช้พอร์ต A เพื่อการโอนย้ายข้อมูลแบบ 2 ทิศทางโดยใช้บิตจากพอร์ต C เป็นสัญญาณ handshake ส่วนพอร์ต B สามารถถูกใช้ในโหมด 0 หรือ 1 ก็ได้



รูปที่ 9.13 โหมดการทำงานของ 8255A

รูปที่ 9.14 แสดงแผนภาพโครงสร้างภายในของ 8255A โดยประกอบด้วย port ขนาด 8 บิต 2 พอร์ต (พอร์ต A และ B) พอร์ตขนาด 4 บิต 2 พอร์ต (พอร์ต C_U และ C_L), Data bus buffer และ control logic ต่างๆ สังเกตจัดแบ่งกลุ่มของพอร์ต ด้วย ว่ามีการจัดแบ่ง 2 กลุ่มคือ

- Group A ซึ่งประกอบด้วยพอร์ต A (PA_7 - PA_0) และ C_U (PC_7 – PC_4)
- Group B ซึ่งประกอบด้วยพอร์ต B (PB_7 - PB_0) และ C_L (PC_3 – PC_0)



รูปที่ 9.14 โครงสร้างภายในของ 8255A (ที่มา: Gaonkar, 1992)

ขั้นตอนที่สำคัญในการติดต่อระหว่าง MPU กับอุปกรณ์รอบข้างผ่านอุปกรณ์ต่อประสานที่โปรแกรมได้นั้น มีอยู่ด้วยกัน 3 ขั้นตอนดังนี้

1. กำหนด address ของแต่ละ I/O port รวมทั้งของ control register ของอุปกรณ์ต่อประสานนั้นๆ
2. เขียน control word ที่เหมาะสมลงใน control register
3. เขียนคำสั่ง I/O ที่เหมาะสม (ขึ้นอยู่กับประเภทของการติดต่อ – อ่าน/เขียน) เพื่อติดต่อกับอุปกรณ์รอบข้าง

Control logic

8255A มีสัญญาณควบคุมอยู่ด้วยกัน 6 สัญญาณ ดังนี้

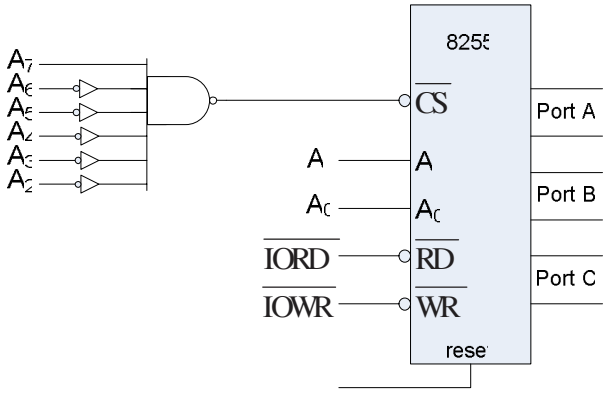
- $\overline{RD}$: ใช้ในการ enable การอ่านข้อมูล คือ เมื่อ active MPU จะอ่านข้อมูลจาก I/O port ที่เลือก
- $\overline{WR}$: ใช้ในการ enable การเขียนข้อมูล คือ เมื่อ active MPU จะเขียนข้อมูลลงใน I/O port ที่เลือกหรือ control register
- $\overline{CS}$ **A_1 และ A_0**: ใช้ในการเลือกอุปกรณ์ (ตารางที่ 9.5) โดยที่ $\overline{CS}$ จะเป็นสัญญาณเลือกชิปหลัก ส่วน A_1 และ A_0 จะกำหนด I/O port หรือ control register ที่ต้องการ โดยทั่วไป

จะต่อ $\overline{CS}$ เข้ากับ address ที่ได้ถอดรหัสแล้ว ส่วน A_1 และ A_0 จะต่อเข้ากับ 2 บิตล่าง (A_1 และ A_0) ของ address จาก MPU ตามลำดับ

$\overline{CS}$	A_1	A_0	พอร์ตที่เลือก
0	0	0	พอร์ต A
0	0	1	พอร์ต B
0	1	0	พอร์ต C
0	1	1	Control Register
1	X	X	8255A ไม่ได้ถูกเลือก

ตารางที่ 9.5 สรุปการเลือก port ของ 8255A

ตัวอย่างการ decode address ให้กับ 8255



รูปที่ 9.15 วงจรการต่อประสาน 8255 กับไมโครโพรเซสเซอร์ Z80 (ที่มา: Gaonkar, 1992)

จากรูปที่ 9.15 จะได้ว่า

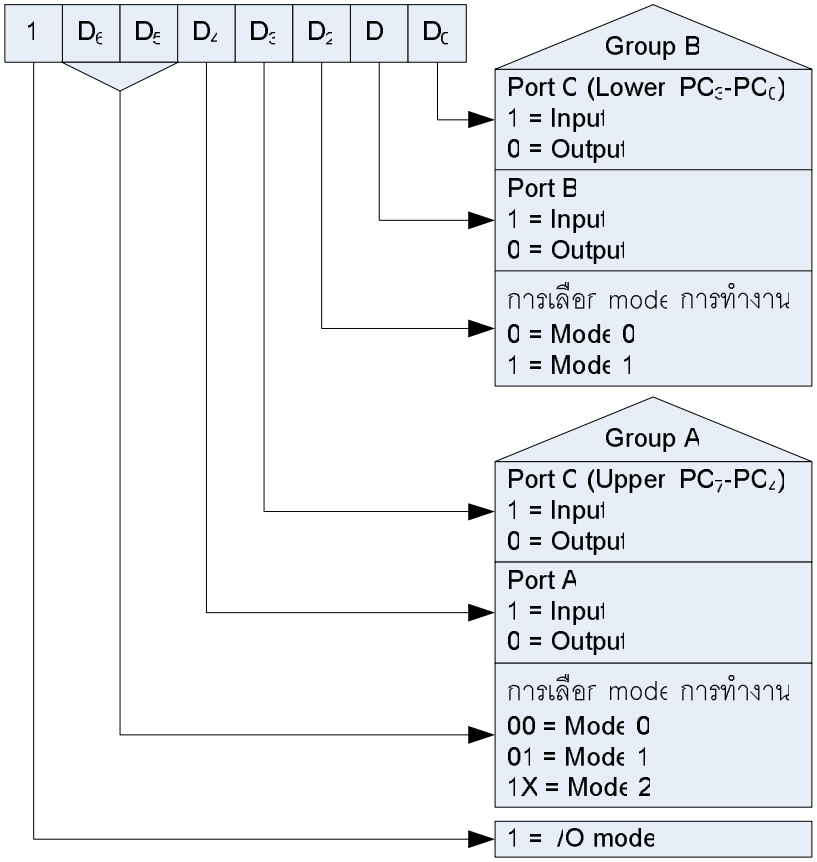
	$\overline{CS}$								
	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
Port A	1	0	0	0	0	0	0	0	$=80_H$
Port B							0	1	$=81_H$
Port C							1	0	$=82_H$
Control register							1	1	$=83_H$

Control Word

คือค่าที่อยู่ใน control register ของ 8255A การควบคุมการทำงานของ 8255A ทำได้โดยเขียน control word ที่เหมาะสมตามโหมดการทำงาน (รูปที่ 9.16 และ 9.17) ลงใน control register (A_1 และ A_0 มีค่าเป็น 1)

Mode 0: โหมด I/O อย่างง่าย

สำหรับในโหมดนี้แต่ละพอร์ตจะทำหน้าที่เป็น I/O port อย่างง่ายๆ โดยไม่มีการ interrupt หรือสัญญาณ handshake เข้ามาเกี่ยวข้อง โดยที่ทั้งพอร์ต A และ B จะเป็น I/O port ขนาด 8 บิต ในขณะที่พอร์ต C_U และ C_L จะเป็น I/O port ขนาด 4 บิต รูปที่ 9.16 แสดงรูปแบบของ control word สำหรับ I/O mode ทั้งสามโหมด (โหมด 0 1 และ 2)



รูปที่ 9.16 Control word สำหรับ I/O mode ของ 8255A (ที่มา: Gaonkar, 1992)

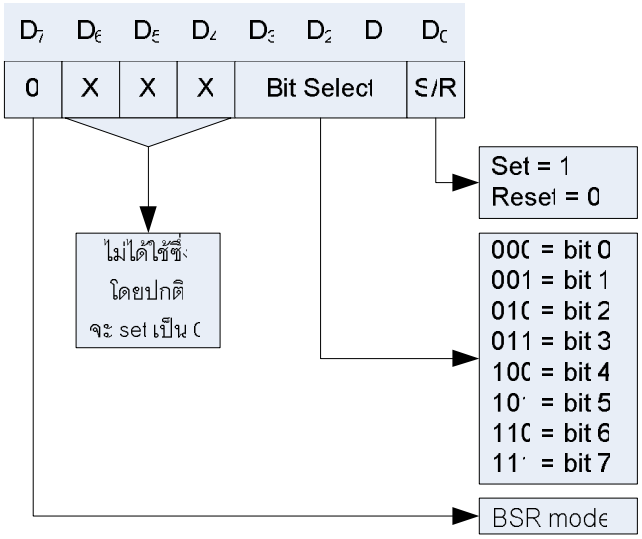
โหมดนี้มีลักษณะเฉพาะดังนี้

- Output จะค้าง
- Input จะไม่ค้าง
- ไม่มี handshake

ในกรณีที่ C_u และ C_L ทำงานกันคนละหน้าที่ เช่น C_u เป็นอินพุตส่วน C_L เป็นเอาท์พุตหรือกลับกัน การเขียนข้อมูลไปยังพอร์ต C นี้ จะทำให้ข้อมูลไปปรากฏในส่วนที่เป็นเอาท์พุตไม่มีผลกระทบต่อส่วนที่เป็นอินพุตส่วนการอ่านข้อมูลจากพอร์ต C นี้ จะอ่านข้อมูลส่วนที่เป็นทั้ง input และเอาท์พุตที่ถูกเขียนออกมา

BSR mode: โหมด bit set/reset

โหมดนี้สามารถ set หรือ reset แต่ละบิตของพอร์ต C ได้ด้วยการเขียน control word ที่เหมาะสมลงใน control register โดยที่รูปแบบของ control word สำหรับ BSR mode นี้แสดงในรูปที่ 9.17



รูปที่ 9.17 Control word สำหรับ BSR mode ของ 8255A (ที่มา: Gaonkar, 1992)

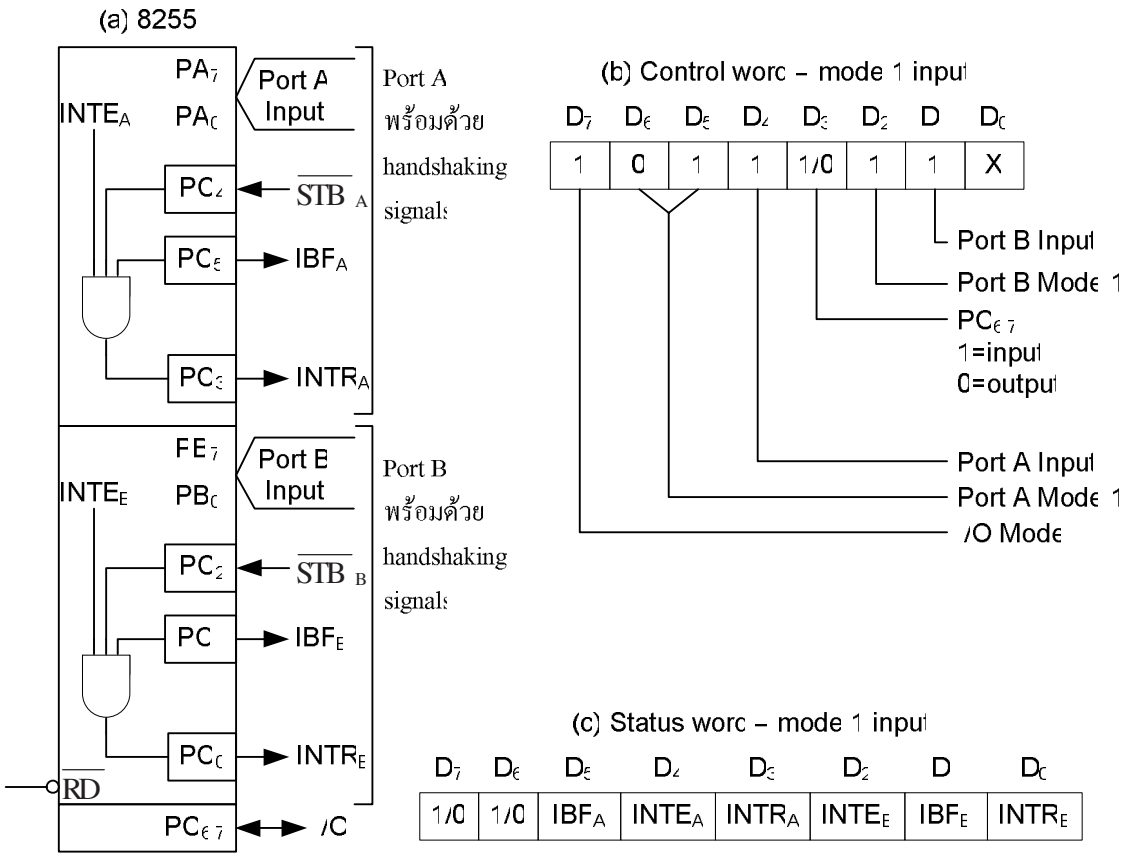
Mode 1: Input/Output mode และสัญญาณ Handshake

ลักษณะของโหมด 1 เป็นดังนี้ พอร์ต A และ B ทำหน้าที่เป็นได้ทั้งพอร์ตอินพุตหรือเอาท์พุต ขนาด 8 บิต โดยที่แต่ละพอร์ตใช้สายสัญญาณของพอร์ต C จำนวน 3 สายเป็นสัญญาณ handshake ในขณะที่สายสัญญาณอีก 2 สายที่เหลืออาจทำหน้าที่เป็น I/O อย่างง่ายได้ โดยก่อนการถ่ายโอนข้อมูลจะมีการแลกเปลี่ยนสัญญาณ handshake ระหว่าง MPU กับอุปกรณ์รอบข้าง และข้อมูลทั้งที่เป็นอินพุตและเอาท์พุตจะค้างไว้ได้

ในชิป 8255A สายของพอร์ต C ที่ใช้เป็นสัญญาณซึ่งกันและกันนั้น แปรผันสอดคล้องกับฟังก์ชัน I/O ของพอร์ต ดังนั้น จึงต้องพิจารณาฟังก์ชัน (หน้าที่) อินพุตและเอาท์พุตในโหมด 1 แยกกันดังนี้

<u>Mode 1 : สัญญาณควบคุม input</u>

รูปที่ 9.18 แสดงสัญญาณควบคุมขณะจัดพอร์ต A และพอร์ต B เป็นพอร์ตอินพุต



รูปที่ 9.18 Mode 1 ของ 8255A (input) (ที่มา: Gaonkar, 1992)

พอร์ต A จะใช้สัญญาณ 3 สัญญาณด้านบน คือ PC_3 PC_4 และ PC_5 ในขณะที่พอร์ต B จะใช้สัญญาณ PC_2 PC_1 และ PC_0 หน้าที่ของสัญญาณเหล่านี้ คือ

(a) $\overline{STB}$ (Strobe) : สัญญาณนี้ (active low) เกิดจากอุปกรณ์รอบข้างเพื่อแสดงว่าได้ส่งไบต์ข้อมูลแล้ว โดยที่ ชิป 8255A จะสร้างสัญญาณ IBF และ INTR ในการตอบสนองต่อ $\overline{STB}$

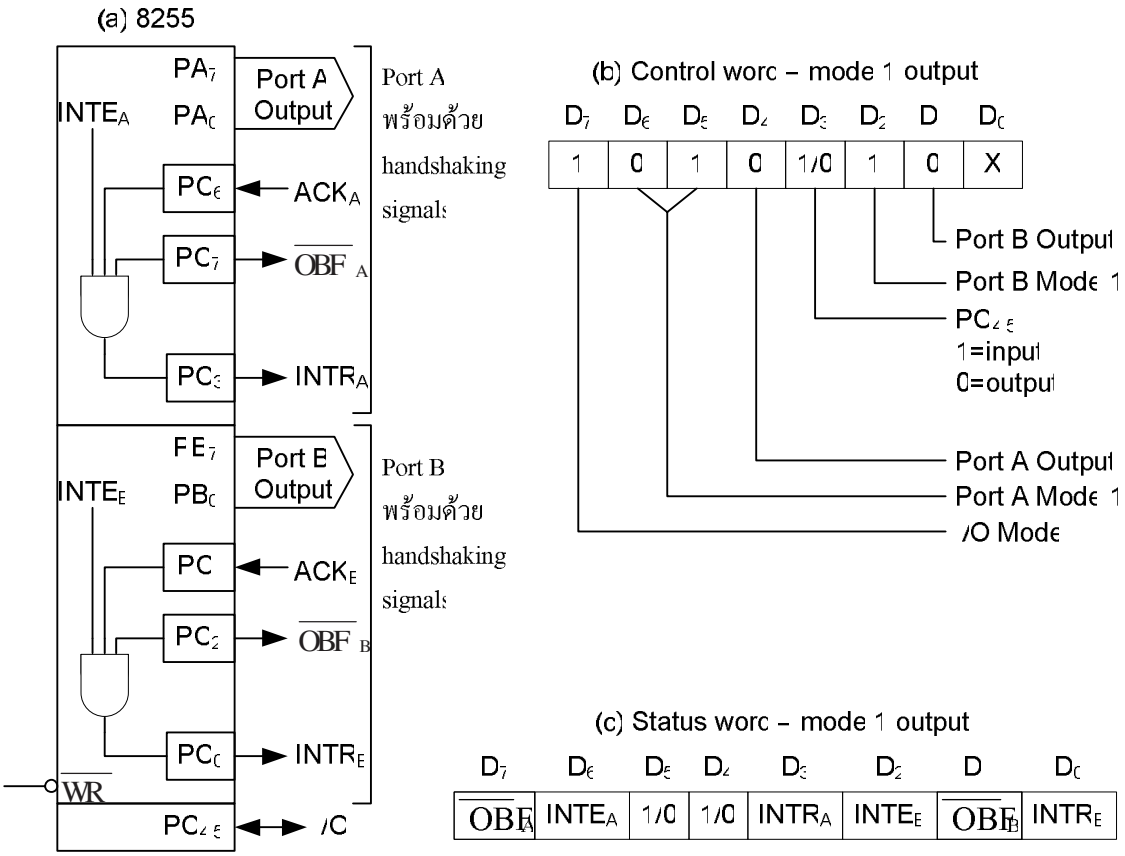
(b) IBF (Input Buffer Full) : เป็นสัญญาณตอบรับโดยที่ชิป 8255A ใช้เพื่อแสดงว่าแลตช์ที่อินพุตได้รับไบต์ข้อมูล สัญญาณนี้รีเซ็ตเมื่อ MPU อ่านข้อมูลนั้น

(c) INTR (Interrupt Request) : เป็นสัญญาณเอาท์พุตซึ่งใช้เพื่อขัดจังหวะ MPU สัญญาณนี้เกิดขึ้นเมื่อ $\overline{STB}$ IBF และ INTE (flip-flop ภายใน) อยู่ที่ลอจิก 1 ทั้งหมด สัญญาณนี้รีเซ็ตโดยขอบขาลงของสัญญาณ $\overline{RD}$

(d) INTE (Interrupt Enable) : เป็น flip-flop ภายในซึ่งใช้เพื่อ Enable หรือ Disable การสร้างสัญญาณ INTR flip-flop 2 ตัว คือ $INTE_A$ และ $INTE_B$ ได้รับการเซ็ต/รีเซ็ตผ่านโหมด BSR flip-flop $INTE_A$ ได้รับการ Enable/Disable ผ่าน PC_4 ส่วน flip-flop $INTE_B$ ได้รับการ Enable/Disable ผ่าน PC_2

Mode 1: สัญญาณควบคุม input

รูปที่ 9.18 แสดงสัญญาณควบคุมขณะจัดพอร์ต A และพอร์ต B เป็นพอร์ตเอาท์พุต



รูปที่ 9.19 Mode 1 ของ 8255A (output) (ที่มา: Gaonkar, 1992)

(i) $\overline{\text{OBF}}$ (Output Buffer Full) : เป็นสัญญาณเอาท์พุตซึ่งเป็น low เมื่อ MPU เขียนข้อมูลลงในแลตช์ที่เอาท์พุตของชิป 8255A สัญญาณนี้แสดงแก่อุปกรณ์รอบข้างที่เอาท์พุตว่าพร้อมที่จะอ่านข้อมูลใหม่ สัญญาณนี้จะเป็น high อีกครั้งหลังจากชิป 8255A รับสัญญาณตอบรับ $\overline{\text{ACK}}$ จากอุปกรณ์รอบข้าง

(ii) $\overline{\text{ACK}}$ (Acknowledge) : เป็นสัญญาณอินพุตของชิป 8255A จากอุปกรณ์รอบข้าง ซึ่งจะได้เอาท์พุตออกมาเป็นค่า LOW เมื่ออุปกรณ์รับข้อมูลจากพอร์ตของชิป 8255A

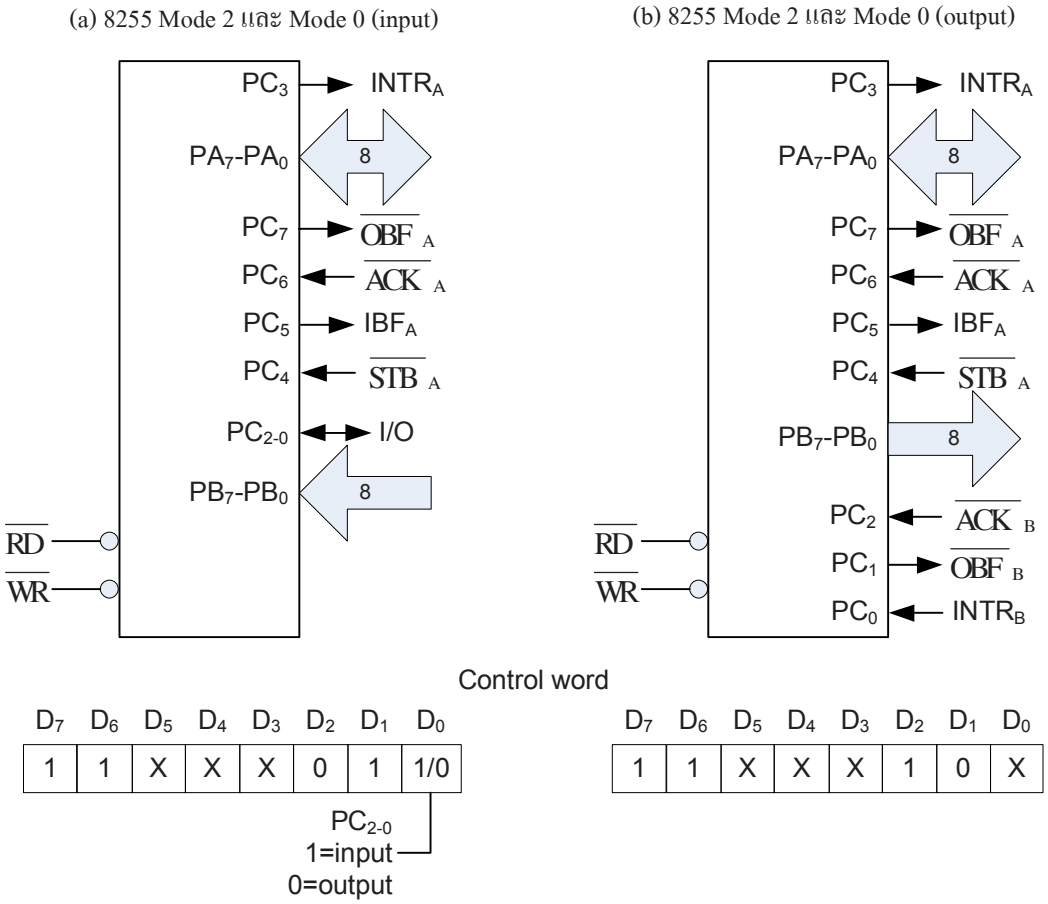
(iii) INTR (Interrupt Request) : เป็นสัญญาณเอาท์พุตซึ่งรีเซ็ตโดยขอบขาขึ้นของสัญญาณ $\overline{\text{ACK}}$ เราใช้สัญญาณนี้ในการขัดจังหวะ MPU เพื่อขอไบต์ข้อมูลถัดไปสำหรับเอาท์พุต INTR นี้ได้รับการเซ็ตเมื่อ $\overline{\text{OBF}}$ $\overline{\text{ACK}}$ และ INTE เป็น 1 ทั้งหมด โดยขอบขาลงของ $\overline{\text{WR}}$ จะรีเซ็ตสัญญาณนี้

(iv) INTE (Interrupt Enable) : เป็นฟลิปฟลอปภายในที่ต่อไปยังพอร์ตใดพอร์ตหนึ่ง เราควบคุม flip-flop 2 ตัว คือ INTE_A และ INTE_B โดยใช้บิต PC_6 และ PC_2 (ตามลำดับ) ผ่านโหมด BSR

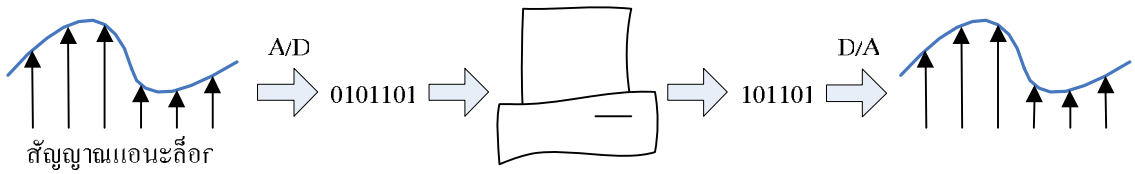
(v) PC_4 และ PC_5 : จัดเตรียมสายเหล่านี้เป็นอินพุตหรือเอาท์พุตก็ได้

Mode 2: การถ่ายโอนข้อมูล 2 ทิศทาง

โหมดนี้ใช้งานได้มากมาย เช่น การถ่ายโอนข้อมูลระหว่างไมโครคอมพิวเตอร์ 2 ตัว หรือการเชื่อมประสานตัวควบคุมแผ่นบันทึก โหมดนี้เป็นการจัดเตรียมพอร์ต A เพื่อถ่ายโอนข้อมูล 2 ทิศทาง และจัดเตรียมพอร์ต B ในโหมด 0 หรือ โหมด 1 ได้ พอร์ต A จะใช้สัญญาณ ของพอร์ต C จำนวน 5 สัญญาณเป็นสัญญาณควบคุมเพื่อการถ่ายโอนข้อมูล สัญญาณของพอร์ต C ที่เหลืออีก 3 สัญญาณใช้เป็น I/O อย่างง่ายหรือใช้เป็นสัญญาณซึ่งกันและกันสำหรับพอร์ต B ก็ได้ สำหรับรูปที่ 9.19 แสดงโครงสร้างทั้งสองโครงสร้างของโหมด 2



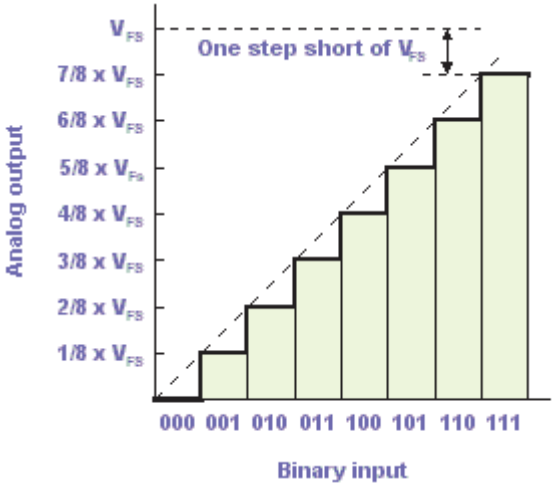
รูปที่ 9.19 Mode 2 ของ 8255A (bidirectional) (ที่มา: Gaonkar, 1992)



10.1 Digital-to-Analog Conversion (DAC)

อุปกรณ์ทางไฟฟ้า/อิเล็กทรอนิกส์โดยทั่วไปที่เป็นแอนะล็อกสามารถควบคุมการทำงานโดยการให้อินพุตเป็นระดับแรงดันที่แตกต่างกัน ตัวอย่างเช่น มอเตอร์กระแสตรงซึ่งควบคุมความเร็วโดยเปลี่ยนระดับแรงดัน (หรือกระแส) ของขดลวดสนาม

เมื่อนำระบบดิจิตอล หรือไมโครคอนโทรลเลอร์ปมาใช้ควบคุมอุปกรณ์ทางแอนะล็อกเหล่านี้ จึงต้องมีวงจรซึ่งสามารถแปลงสัญญาณทางดิจิตอลเป็นระดับแรงดันต่อเนื่องแบบแอนะล็อก ตั้งแต่ 0V จนถึงระดับสูงสุดที่กำหนดไว้ เรียกว่าวงจร Digital-to-Analog Converter (DAC)



รูปที่ 10.1 Transfer Curve ในอุดมคติของ DAC 3 บิต (ที่มา: http://www.cpe.ku.ac.th/~yuen/204471/)

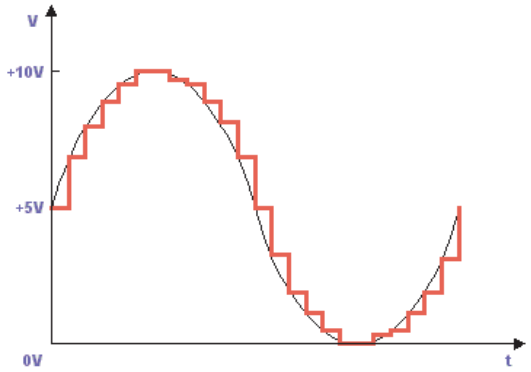
รูปที่ 10.1 เป็นกราฟแสดงถึงความสัมพันธ์ระหว่างเอาท์พุตที่เป็นแอนะล็อกกับอินพุตที่เป็นดิจิตอลขนาด 3 บิตเรียกว่า transfer curve สังเกตว่าเมื่ออินพุตไบนารีเพิ่มขึ้น เอาท์พุตแอนะล็อกจะเพิ่มในลักษณะขั้นบันได โดยที่ขนาดของแต่ละขั้นจะหาได้จาก

$$\text{stepsize} = V_{FS}/2^n$$

เมื่อให้ V_{FS} คือระดับแรงดันเอาท์พุตสูงสุด

n คือจำนวนบิตของอินพุต

เนื่องจากเอาท์พุตของ DAC จะเพิ่มเป็นขั้นๆ รูปคลื่นสัญญาณ ที่ได้จาก DAC จึงมีลักษณะไม่เรียบ ดังตัวอย่าง ในรูปที่ 10.2 ซึ่งแสดงถึงสัญญาณไซน์ ที่สร้างจาก DAC



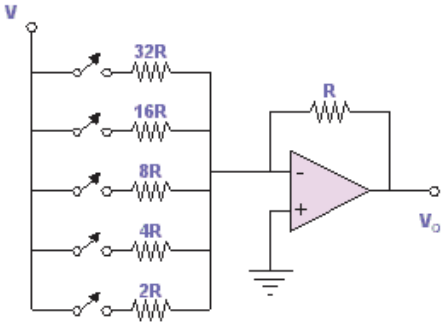
รูปที่ 10.2 คลื่นไซน์ที่สร้างจาก DAC (ที่มา: http://www.cpe.ku.ac.th/~yuen/204471/)

ถ้าเพิ่มจำนวนบิต ความละเอียดของ DAC จะเพิ่มขึ้น เช่น เมื่อ ใช้ DAC 12 บิต และ V_{FS} = 5.0 V ความละเอียดคือ 5.0 V / 4096 = 1.22 mV ซึ่งจะ ละเอียดกว่า DAC 8 บิตถึง 16 เท่า

ความถูกต้องของ DAC ขึ้นอยู่กับหลายส่วน

1. **Quantization error** DAC บิต V_{FS} = 5.0 V เอาท์พุตจะมีความละเอียด 19.53 mV ถ้าต้องการเอาท์พุต 4.00 V DAC จะให้เอาท์พุตได้ใกล้เคียง ที่สุดคือ 4.04 V (19.53 mV x 205) ผิดพลาด 4 mV โดยทั่วไป ค่าผิดพลาดจะเท่ากับ +/- 0.5 LSB (least significant bit) ตัวอย่างเช่น DAC 8 บิต ความผิดพลาดจะเป็น 1 ใน 512 หรือ +/- 0.195 %
2. **Offset and gain errors** เมื่ออินพุตไบนารีเท่ากับ 0 แต่เอาท์พุตของ DAC ไม่เป็น 0 เรียก ว่า offset error และอาจเกิดร่วมกับ gain error ความผิดพลาดเหล่า นี้จะทำให้ tranfer curve ในรูปที่ 10.1 โค้งขึ้นหรือลงจะขึ้นอยู่กับความไม่สมดุลย์ภายใน DAC อย่างไรก็ตาม offset error และ gain error จะแก้ได้โดยใช้ความต้าน ทานปรับค่าได้ต่อไว้ภายนอก
3. **Nonlinearity** คือค่าความคลาดเคลื่อนสูงสุดของ transfer curve เทียบกับเส้นตรงจากจุดศูนย์และจุดสูงสุด ซึ่งจะขึ้นอยู่กับความผิดพลาดของส่วนประกอบภายใน DAC ใน data sheet ของ DAC จะระบุเป็นเปอร์เซ็นต์เทียบกับค่าสูงสุด หรือ ระบุเป็นเศษส่วนของ LSB (โดยทั่วไปคือ +/- 0.5 LSB)
4. **Settling time** คือช่วงเวลานับแต่ให้อินพุตจนกระทั่ง DAC ให้เอาท์พุต วัดเมื่อเอาท์พุตที่ได้ผิดพลาดจากค่าจริง น้อยกว่า 0.5 LSB ค่าเวลานี้อาจน้อยกว่า 100 ns สำหรับ DAC ความเร็วสูง และอาจมากกว่า 100 µs สำหรับ DAC ราคาถูก

Summed Source DAC

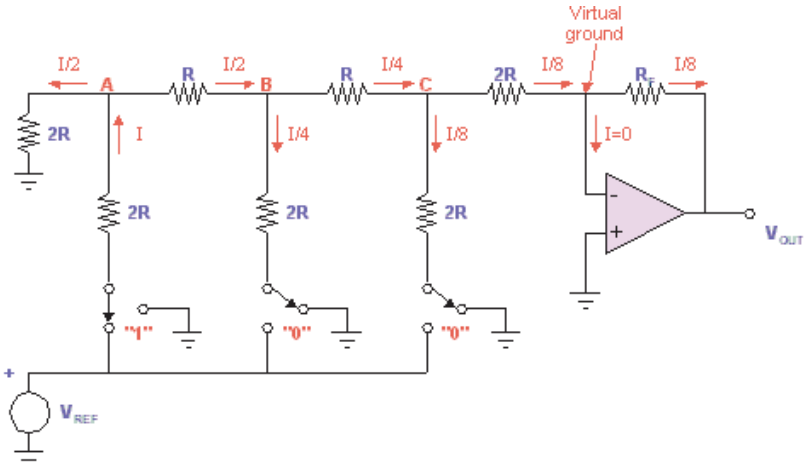


รูปที่10.3 Summed Source DAC (ที่มา: http://www.cpe.ku.ac.th/~yuen/204471/)

รูปที่ 10.3 แสดงถึงวงจร Summed Source DAC ซึ่งเป็นวงจรอย่างง่ายในการแปลงสัญญาณดิจิตอลเป็นแอนะล็อก โดยความต้านทานมีค่าเป็น 2R 4R และ 8R เพื่อทำให้กระแสที่ผ่านความต้านทานแต่ละตัวมีค่าลดลงเป็น 2 เท่า ความต้านทานตัวล่างสุด (2R) จะ เป็น MSB ส่วนตัวบนสุดจะเป็น LSB

ข้อเสียของการใช้วงจรลักษณะนี้ ในทางปฏิบัติ ค่าความต้านทานที่ต่างกันเป็น 2 เท่า คือ 2R 4R 8R … จะ ไม่สามารถหาได้ง่าย จึงมีการปรับปรุงเป็น วงจร R-2R

Switched Voltage R-2R DAC

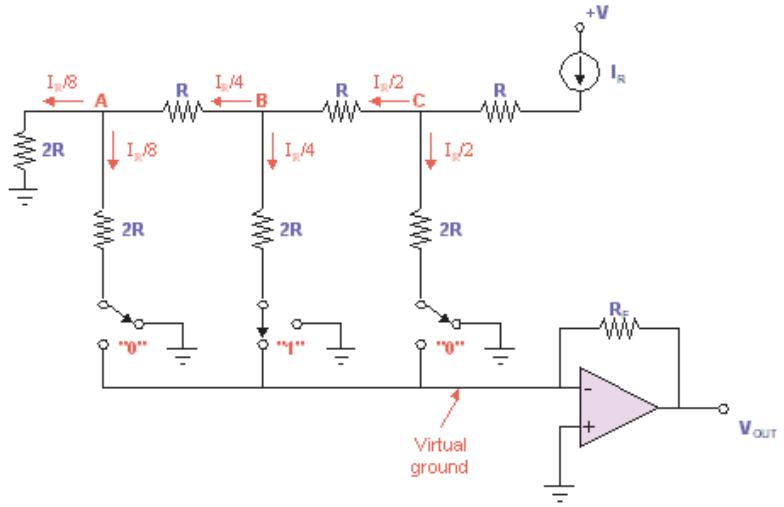


รูปที่ 10.4 Swiched Voltage R-2R DAC (ที่มา: http://www.cpe.ku.ac.th/~yuen/204471/)

รูปที่ 10.4 เป็น DAC 3 บิต ใช้ออปแอมป์และความต้านทาน เพียง 2 ค่าคือ R และ 2R สังเกตว่าอินพุตดิจิตอลจะมาจากสวิตช์ทั้ง 3 ซึ่งอาจต่อกับกราวด์ (ลอจิก 0) หรือต่อกับ VREF (ลอจิก 1) ตัวอย่างนี้อินพุตเป็น 001

พิจารณากระแส I เมื่อผ่านจุด A จะถูกแบ่งเป็นสองส่วนเท่าๆ กัน เหลือ I/2 เมื่อผ่านจุด B และ C จะถูกแบ่งอีกครั้ง เหลือ I/4 และ I/8 ตามลำดับ ดังนั้นกระแสที่ป้อนให้กับออปแอมป์จะเหลือ I/8 เมื่อพิจารณาที่สวิตช์ตัวอื่นๆ ก็จะมีลักษณะคล้ายกัน ดังนั้นกระแสที่ผ่านออปแอมป์เมื่อปิดสวิตช์อื่นนับจากซ้ายไปขวา จะมีขนาด I/8 I/4 และ I/2 ตามลำดับ สวิตช์ซ้ายสุดจะเป็น LSB ส่วนขวาสุดจะเป็น MSB

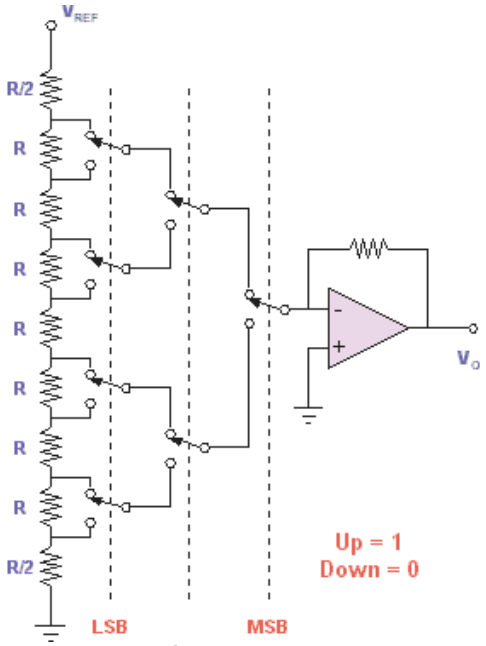
Switched Current R-2R DAC



รูปที่ 10.5 Swiched Current R-2R DAC (ที่มา: http://www.cpe.ku.ac.th/~yuen/204471/)

วงจรนี้เปลี่ยนจากการใช้แรงดันอ้างอิง (V_{REF}) มาเป็นกระแสอ้างอิง (I_R) กระแสที่ ผ่านสวิตช์แต่ละตัวจากขวาไปซ้ายจะเป็น $I_R/2$, $I_R/4$ และ $I_R/8$ ตามลำดับ วงจรลักษณะนี้จะมีความเร็วสูงกว่าวงจร Switched Voltage เนื่องจากคาปาซิแตนซ์ที่รอยต่อ (junction capacitance) ของความต้านทานแต่ละตัวจะไม่ถูกชาร์จและดิสชาร์จเหมือนวงจร Switched Voltage

Switched Pole DAC



รูปที่ 10.6 Swiched Pole DAC (ที่มา: http://www.cpe.ku.ac.th/~yuen/204471/)

จะมีการใช้ความต้านทานต่ออนุกรมกันหลายตัว เนื่องจากวงจรนี้ต้องการความต้านทานค่าเท่าๆ กัน ดังนั้นจึงเป็นที่นิยมสำหรับผู้ผลิต Integrated Circuit สังเกตว่าจะมีความต้านทานที่ปลายทั้งสองของอนุกรม เพื่อปรับ offset ของเอาท์พุต

10.2 Analog-to-Digital Conversion (ADC)

สัญญาณที่ใช้ในอุปกรณ์อิเล็กทรอนิกส์ ฯมี 2 ชนิด คือ สัญญาณแอนะล็อกและสัญญาณดิจิตอล โดยที่สัญญาณแอนะล็อกจะใช้ในอุปกรณ์ทั่วๆ ไปและใช้ในการควบคุมแบบเก่า

ในปัจจุบันมีไมโครโพรเซสเซอร์และไมโครคอนโทรลเลอร์เข้ามาช่วยในการควบคุมอุปกรณ์ต่างๆ มากมาย ซึ่งทำให้การควบคุมนั้นทำได้ง่ายและรวดเร็วยิ่งขึ้น แต่ในการควบคุมนั้นเราจำเป็นต้องใช้สัญญาณดิจิตอลในการติดต่อกับไมโครโพรเซสเซอร์หรือไมโครคอนโทรลเลอร์ แต่ในความเป็นจริงนั้นเราใช้สัญญาณแอนะล็อกในการควบคุม ดังนั้นเราจึงจำเป็นต้องมีการเปลี่ยนสัญญาณแอนะล็อกเป็นสัญญาณดิจิตอลแล้วจึงนำสัญญาณนั้นเข้ามาสู่ไมโครโพรเซสเซอร์หรือไมโครคอนโทรลเลอร์เพื่อใช้ควบคุมระบบต่อไป

แม้ว่าสัญญาณแอนะล็อกนั้นมีความแน่นอนและแม่นยำสูง แต่สัญญาณแอนะล็อกนั้นก็ควบคุมได้ยาก เนื่องจากในสภาพแวดล้อมมีสัญญาณรบกวนอยู่มาก และการที่จะทำให้การควบคุมแบบแอนะล็อก มีความสามารถควบคุม เท่ากับการควบคุมแบบดิจิตอลนั้นทำได้ยาก เนื่องจากวงจรควบคุมแบบแอนะล็อกจะต้องมีความซับซ้อนสูง

อย่างไรก็ตาม สัญญาณดิจิตอลก็ไม่สามารถทดแทนความละเอียดของสัญญาณแอนะล็อกได้อย่างสมบูรณ์ แต่ทำให้การควบคุมนั้นทำให้ง่าย และสะดวกยิ่งขึ้น

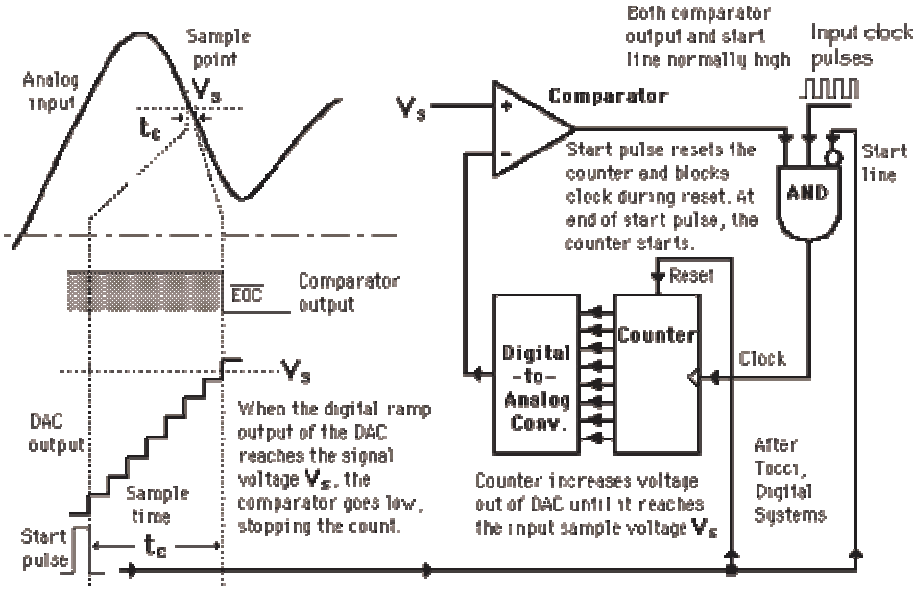
อินพุตจะถูกแบ่งเป็นหลายระดับ แต่ละระดับต่างกัน 1 quantum (Q) จุดกึ่งกลางของแต่ละระดับเรียกว่า threshold ถ้าอินพุตเปลี่ยนอยู่ในช่วง ±½ LSB จาก threshold จะไม่ทำให้ digital output เปลี่ยน ดังนั้น ADC จะมี inherent quantization uncertainty ขนาด ±½ LSB เสมอ (ไม่ว่าจะเพิ่มจำนวน bit เป็นเท่าใดก็ตาม)

Digital Ramp ADC

รูปที่ 10.7 แสดงถึงลักษณะการทำงานของ Digital Ramp ADC ซึ่งเป็นวิธีที่ง่ายที่สุดของการแปลงสัญญาณแอนะล็อกให้เป็นสัญญาณดิจิตอล โดยใช้อัลกอริทึม การนับค่าเพิ่มขึ้นเรื่อยๆ แล้วนำผลที่ได้จากการนับไปเปรียบเทียบกับค่าที่ต้องการที่ตั้งไว้

จากวงจร Counter เป็นอุปกรณ์นับค่าที่เพิ่มขึ้นทีละหนึ่ง แล้วส่งค่าที่ได้ให้ D/A มีขา Reset รับสัญญาณ Reset เมื่อต้องการให้เริ่มนับใหม่

D/A เมื่อรับค่าที่นับเพิ่มขึ้นทีละหนึ่งจากตัวนับ ก็แปลงค่าให้เป็นสัญญาณ แอนะล็อกที่มีค่าความต่างศักย์ค่าๆ หนึ่ง แล้วส่งต่อเข้าไปที่อุปกรณ์ตัวเปรียบเทียบ (Comparator)

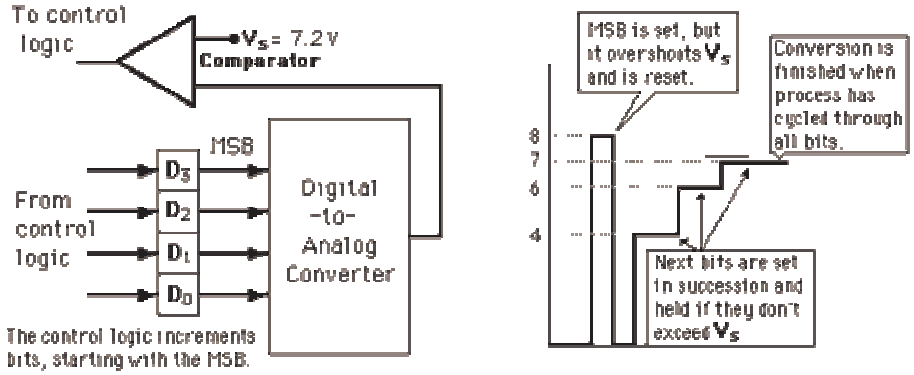


รูปที่ 10.7 Digital Ramp ADC (ที่มา: http://hyperphysics.phy-astr.gsu.edu/hbase/hframe.html)

Comparator จะเป็นอุปกรณ์ตัวเปรียบเทียบค่าความต่างศักย์ ของอินพุต และค่าจากที่ตัวนับ ถ้าหากทั้งสองสัญญาณมีค่าเท่ากันส่งค่าความต่างศักย์ 0 โวลต์ออกมา(ลอจิก 0) ถ้าไม่เท่ากันก็จะส่งความต่างศักย์ที่ไม่ใช่ 0 โวลต์ออกมา(ลอจิก 1) ซึ่งค่าความต่างศักย์ที่ออกมา จะนำมาเข้าลอจิกเกต "และ" กับสัญญาณนาฬิกา จะได้ค่าลอจิกออกมา ถ้าผลลัพธ์ออกมาเป็นสัญญาณนาฬิกาแสดงว่ายังไม่ได้ผลลัพธ์เท่าที่ต้องการสัญญาณนาฬิกาก็จะไปทำให้ตัวนับนับเพิ่มขึ้นต่อไป และเมื่อได้ค่าผลลัพธ์ดิจิตอลที่ต้องการแล้วค่าที่ได้จากตัวเปรียบเทียบจะให้ค่าความต่างศักย์เป็น 0 (ลอจิก 0) ซึ่งเมื่อนำมาเข้าลอจิกเกต "AND" กับสัญญาณนาฬิกาแล้ว ก็จะให้ลอจิก 0 ซึ่งทำให้ตัวนับไม่นับเพิ่มอีก ก็จะได้ค่าดิจิตอลจากตัวนับที่ต้องการ

ข้อเสียของวิธีนี้คือ การนับต้องเริ่มนับที่ 0 เสมอ และนับเพิ่มขึ้นเรื่อยๆ ทำให้ช้า เอาท์พุตที่ได้จะมี delay จึงไม่ค่อยนิยมใช้เท่าที่ควร จึงได้เปลี่ยนตัวนับเป็นแบบนับลงได้ด้วย ซึ่งจะอ้างอิงระดับจากระดับเก่า ทำให้ไม่จำเป็นต้องนับ 0 ใหม่เมื่อมีการเปลี่ยนอินพุตใหม่แต่ให้อ้างอิงกับผลลัพธ์เดิม ทำให้ได้ผลลัพธ์เร็วขึ้น

Successive Approximation ADC

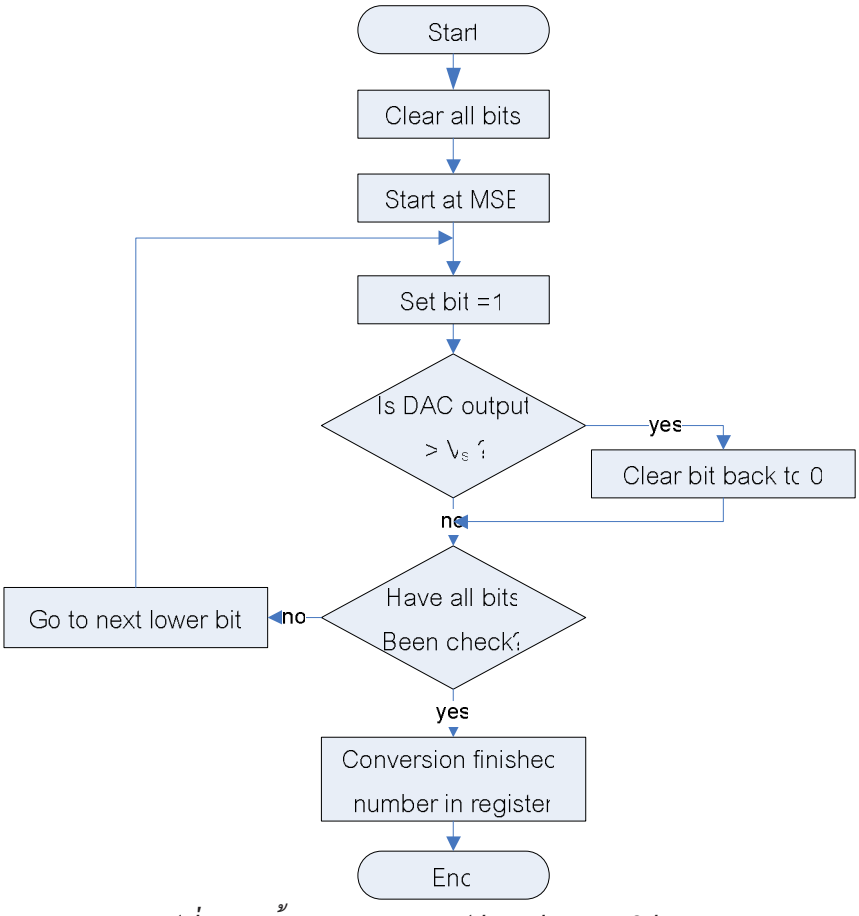


รูปที่ 10.8 4-bit Successive ADC (SAC) with 1 volt step size

(ที่มา: http://hyperphysics.phy-astr.gsu.edu/hbase/hframe.html)

วิธี Successive approximation ADC (SAC) ใช้หลักการของ "binary search" ในการหาคำตอบ โดยนำค่าผลลัพธ์มาเปรียบเทียบกับค่ากึ่งกลางของช่วง เพื่อให้ทราบว่าค่านั้นๆ มากกว่าหรือน้อยกว่า โดยจะปรับช่วงให้แคบลงมาเรื่อยๆ แล้วเปรียบเทียบผลลัพธ์กับค่ากึ่งกลางของช่วงไปเรื่อยๆ จนได้ผลลัพธ์ที่ต้องการ เช่น เลขที่เป็นคำตอบคือ 3 จากช่วงของคำตอบที่ 0-7 ครั้งแรกเอาค่า (0+7)/2 = 4 มาเปรียบเทียบ ได้ผลว่า คำตอบที่ต้องการอยู่ในช่วงที่น้อยกว่า 4 ครั้งที่ 2 ก็เลือกค่า (0+4)/2 = 2 มาเปรียบเทียบ ได้ผลว่าคำตอบที่ต้องการอยู่ในช่วงที่มากกว่า 2 แต่น้อยกว่า 4 ครั้งที่ 3 ก็เลือกค่า (2+4)/2 = 3 มาเปรียบเทียบ ได้ผลว่าคำตอบที่ต้องการ

ขั้นตอนของการเปรียบเทียบที่ใช้ในวิธี SAC นี้แสดงในรูปที่ 10.9



รูปที่ 10.9 ขั้นตอนของการเปรียบเทียบของวิธี SAC

(ที่มา: http://hyperphysics.phy-astr.gsu.edu/hbase/hframe.html)

Converter Error

- Offset error
- Gain error
- Linearity error
- Differential linearity error

Converter Resolution

- หมายถึง การเปลี่ยนแปลงทาง analog ที่ทำให้ ADC output เปลี่ยนไป 1 ระดับ
- บอกเป็น %FS, mV ของ input range, จำนวน bit ของ converter

Converter Accuracy

- หมายถึงความแตกต่างระหว่าง input voltage กับเอาท์พุตที่ได้มาเมื่อเทียบกลับมาเป็นค่าทาง input ซึ่งจะรวมผลของ error ทุกอย่าง
- อาจบอกเป็นจำนวน voltage หรือ จำนวน LSB

Conversion Time และ Converter Throughput Rate

- ADC ใช้เวลาในการแปลงจากสัญญาณแอนะล็อกเป็นสัญญาณดิจิตอลซึ่งเรียกว่า conversion time (t_c)
- ADC จะมีวงจร S/H (sample and hold circuit) โดยทั่วไปจะไม่ต่อ analog input เข้ากับ ADC โดยตรง โยไม่มีวงจร S/H เนื่องจากถ้าทำเช่นนั้น สัญญาณ analog input จะเปลี่ยนเร็วเกินกว่า ½ LSB ไม่ได้ ถ้าเปลี่ยนเร็วเกินกว่านี้จะทำให้การแปลงสัญญาณเกิดความผิดพลาด ดังนั้นการต่อ input เข้ากับ ADC โดยตรงได้นั้น input จะต้องเป็น input ที่ช้ามากๆ เกินกว่าจะนำไปใช้ประโยชน์ได้
- ส่วน Converter Throughput Rate มีค่าเท่ากับ $1/t_c$

Converter Input และ Output

สัญญาณ analog input: ADCโดยทั่วไปจะรับแรงดันแบบ unipolar จึงจำเป็นที่จะต้องแปลงสัญญาณ analog input ที่มีค่าเป็นทั้ง + และ – ให้เป็น + อย่างเดียว โดยอาจใช้วงจร op-amp ช่วย

Interfacing ADC to ระบบไมโครโพรเซสเซอร์

มีอยู่ด้วยกันหลายแบบ เช่น

1. Most-recent-data scheme
    - ADC ทำงานไม่หยุด และ update ข้อมูลใน buffer
    - ไมโครโพรเซสเซอร์อ่าน buffer เมื่อต้องการ
2. Start-and-wait scheme
    - เมื่อต้องการข้อมูลไมโครโพรเซสเซอร์จะสั่ง ADC
3. Using microprocessor interrupt
    - ADC จะเริ่มทำงานโดยคำสั่งจากไมโครโพรเซสเซอร์หรือ clock
    - เมื่อ ADC ทำงานเสร็จ จะเกิด interrupt ให้ไมโครโพรเซสเซอร์นำข้อมูลไป
    - วิธีนี้ไมโครโพรเซสเซอร์จะไม่ต้องเสียเวลาคอย
    - การจัดการ interrupt มีทั้งแบบ polling และแบบ vectored

Interface Software

วิธีการส่งผ่านข้อมูลจาก ADC มายังไมโครโพรเซสเซอร์มี 3 วิธี คือ

1. Memory-mapped transfer – มี address lines ที่จะต้อง decode มาก
2. I/O-mapped transfer – มี address lines ที่จะต้อง decode น้อยกว่า
3. DMA (Direct Memory Access) – ADC ส่งข้อมูลเข้าเก็บใน memory โดยตรง โดยมี DMA controller เป็นตัวควบคุม ในช่วงนี้ไมโครโพรเซสเซอร์จะหยุดการ access ไปยัง memory ในส่วนที่ DMA ใช้อยู่

บทที่ 11 การส่งข้อมูลผ่าน Parallel Printer Port ของ PC

พอร์ตขนาน (Parallel Port) หรือที่มักเรียกว่า LPT Port สามารถรับ/ส่งข้อมูลในลักษณะ Parallel ได้ ซึ่งทำให้ส่งข้อมูลได้ทีละมากๆ ส่วนใหญ่แล้วจะพบว่าเครื่องพรินเตอร์เกือบทุกรุ่นจะมีใช้พอร์ตดังกล่าวนี้ นอกจากเครื่องพรินเตอร์แล้วก็ยังมีอุปกรณ์อีกหลายประเภทด้วยกันที่ใช้พอร์ตขนาน อย่างเช่น สแกนเนอร์ ฮาร์ดดิสก์แบบติดตั้งภายนอก Zip ไดร์ฟ และ Tapeไดร์ฟ เป็นต้น

เมื่อเครื่องพีซีมีการส่งข้อมูลไปยังเครื่องพรินเตอร์ หรืออุปกรณ์ชนิดอื่น ที่ใช้พอร์ตขนาน จะส่งได้ครั้งละ 8 บิต (1ไบต์) โดยข้อมูลทั้ง 8 บิตนี้ จะถูกส่งไปแบบขนานกันไป เหมือนการเดินเรียงแถวแบบหน้ากระดานตรงข้ามกับการทำงานของพอร์ตอนุกรม (Serial Port) ที่จะส่งข้อมูล 8 บิต โดยนำแต่ละบิตมาเรียงต่อกันเป็นแถวตอน แล้วส่งไปเป็นชุด มาตรฐานของพอร์ตขนานสามารถส่งข้อมูลได้ 50-100 กิโลไบต์ต่อวินาที

Parallel Ports ของ PC มีจุดประสงค์หลักในการออกแบบคือ เพื่อเชื่อมต่อ printer เข้ากับ PC ได้ ไม่ได้ใช้ในการรับส่งข้อมูลธรรมดาทั่วไป ดังนั้น ในการดัดแปลงให้ parallel port สามารถรับส่งข้อมูลธรรมดา จำเป็นต้องศึกษารายละเอียดให้เข้าใจ เพื่อจะไม่ได้เกิดความเสียหายกับ parallel ports

ใน PC โดยทั่วไป จะมี parallel ports อยู่แล้วอย่างน้อย 1 พอร์ต คือ LPT1 ซึ่งประกอบไปด้วย I/O ports 3 พอร์ต ซึ่งจะมีหน้าที่เป็นอินพุตหรือเอาท์พุตต่างกันออกไป นอกจากนี้ ระดับแรงดันของ logic 1 ของพอร์ตหนึ่งอาจจะมีระดับแรงดันตรงกันข้ามกับอีกพอร์ตหนึ่ง ซึ่งทำงานร่วมใน LPT1 เดียวกันนี้ ตารางที่ 11.1 แสดงตำแหน่งและหน้าที่ของพอร์ตใน LPT1

ตำแหน่งของพอร์ต	หน้าที่การทำงานของพอร์ต
0x378	8-bit data port (output port ที่เขียนข้อมูลออกไปได้ และอ่านข้อมูลที่เขียนออกไปกลับมาได้ แต่ต่อวงจรให้รับอินพุตอื่นไม่ได้ เพราะอาจทำให้พอร์ตเสียหาย)
0x379	5-bit status port (input port ที่อ่านข้อมูลได้อย่างเดียว)
0x37a	4-bit control port (เป็นเอาท์พุตหรืออินพุตก็ได้ วงจรทางด้านเอาท์พุตเป็นแบบ open-collector ดังนั้นถ้าจะให้พอร์ตทำงานเป็นอินพุตจะต้องเขียนข้อมูล เพื่อให้ output transistor อยู่ในสภาวะ off เสียก่อน จึงต่อกับอินพุตได้ อย่าต่อกับสัญญาณอินพุตถ้าไม่แน่ใจว่า output transistor อยู่ในสภาวะ off หรือไม่ ยกเว้นถ้าแน่ใจว่า สัญญาณอินพุตได้มาจากวงจรที่มีเอาท์พุตเป็นแบบ open collector)

ตารางที่ 11.1 ตำแหน่งและหน้าที่ของ ports ของ LPT1

ความยาวของสายไฟที่ต่อออกจาก connector แบบ 25 ขา ไม่ควรเกิน 75 ซม. เนื่องจากเป็นการส่งสัญญาณ TTL ที่มีความเร็วค่อนข้างสูง ถ้าความยาวของสายมากเกินไป อาจมีปัญหาการรบกวนในการส่งสัญญาณผ่านสายได้

หัวข้อที่ 11.1 ถึง 11.3 จะกล่าวถึงพอร์ตย่อยทั้งสามของ LPT1 ตลอดจนแสดงถึงความสัมพันธ์ของตำแหน่งบิตของข้อมูลสัมพันธ์กับตำแหน่งของขาของ connector แบบ 25 ขา รวมทั้งค่า logic สำหรับการเขียนและอ่าน

11.1 8-bit Data Port (0x378)

- ใช้เป็นเอาท์พุตได้อย่างเดียว
- ไม่ได้สร้างให้รับอินพุต แต่ถูกสร้างให้อ่านข้อมูลที่เขียนออกไปกลับมาได้
- 5V = logic 1 และ 0V = logic 0

Bit	ระดับแรงดันเมื่อเขียน logic 1	ระดับแรงดันเมื่ออ่าน logic 1	ตำแหน่งขา
0	High	High	2
1	High	High	3
2	High	High	4
3	High	High	5
4	High	High	6
5	High	High	7
6	High	High	8
7	High	High	9

ตารางที่ 11.2 ข้อมูลรายละเอียดของพอร์ต 0x378 (8-bit output)

11.2 5-bit Status Port (0x379)

- ใช้เป็นอินพุตได้อย่างเดียว

Bit	ระดับแรงดันเมื่ออ่าน logic 1	ตำแหน่งขา
0	NOT USED	
1	NOT USED	
2	NOT USED	
3	High	15
4	High	13
5	High	12
6	High	10
7	Low	11

ตารางที่ 11.3 ข้อมูลรายละเอียดของพอร์ต 0x379 (5-bit input)

11.3 4-bit Control Port (0x37A)

- ใช้เป็นอินพุตหรือเอาท์พุตได้
- ถ้าใช้เป็นอินพุตต้องเขียน logic ออกมาให้ output transistor ของ 7406 OFF เสียก่อน จึงจะรับอินพุตได้

Bit	ระดับแรงดันเมื่อเขียน logic 1	ระดับแรงดันเมื่ออ่าน logic 1	ตำแหน่งขา
0	Low	Low	1
1	Low	Low	14
2	High	High	16
3	Low	Low	17
4	ENABLE IRQ 7	ENABLED IRQ 7 BIT	
5	NOT USED		
6	NOT USED		
7	NOT USED		

ตารางที่ 11.4 ข้อมูลรายละเอียดของพอร์ต 0x37a (4-bit input/output)

11.4 Ground Signals

- สาย ground มีตำแหน่งอยู่ที่ขา 18-25
- Protective ground ไม่ควรต่อเข้ากับ ground signals

11.5 การเขียนโปรแกรมภาษา C รับส่งข้อมูลกับ I/O Ports

การรับส่งข้อมูลกับ I/O ports ในภาษา C ทำได้หลายวิธี มีทั้งแบบ 8-bit และ 16-bit และเป็น function calls หรือ macro ก็ได้ ในการทดลองนี้เราจะเลือกใช้ macro รับส่งข้อมูลขนาด 8-bit ตารางที่ 11.5 แสดงชื่อ macro ที่สามารถใช้ในการรับส่งข้อมูลผ่าน parallel port ในภาษา C (สำหรับโปรแกรมในภาษาอื่น ก็สามารถเขียนให้รับส่งข้อมูลกับ ports ได้เช่นกัน)

การเรียกใช้ macro	หน้าที่
`unsigned char inportb(int portid);`	อ่านข้อมูลแบบ byte จากพอร์ตในตำแหน่ง `portid`
`void outportb(int portid, unsigned char value);`	เขียนข้อมูล `value` ไปที่พอร์ตในตำแหน่ง `portid`

ตารางที่ 11.5 Macro ที่ใช้ในการรับส่งข้อมูลกับพอร์ต (ต้องมี `#include <dos.h>` ด้วย)

ตัวอย่างเช่น

```
outportb (int port_id, unsigned char udata);

udata = inportb (int port_id);
```

ถ้าเราใช้ inportb หรือ outportb โดยไม่มี #include <dos.h> จะทำให้ Turbo C Compiler ใช้ function calls แทนที่จะเป็น macro

Bitwise operation	Operator	ตัวอย่างการใช้
OR	|	เช่น data | 0x80
AND	&	เช่น data &0xFE
XOR	^	เช่น data^11
NOT	~	เช่น ~data

11.6 ความเร็วของการรับ/ส่งข้อมูลผ่าน Parallel Printer Port

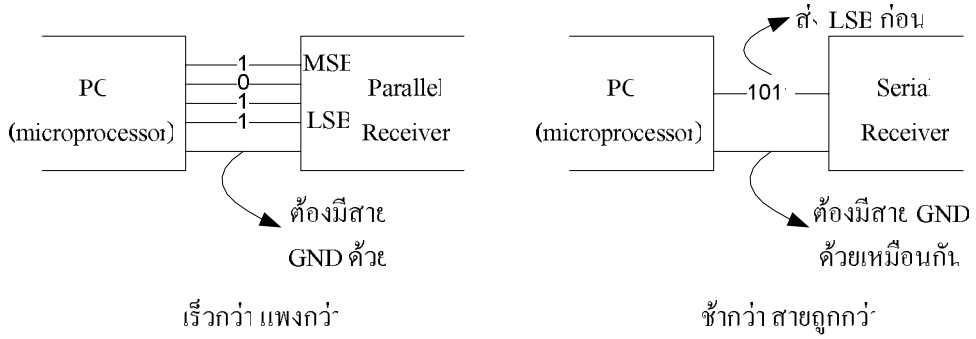
- ไม่แน่นอน ขึ้นอยู่กับความเร็วของ CPU
- การทำงานของ PC จะมี timer interrupt ประมาณ 18.28 ครั้งใน 1 วินาที สามารถรบกวนการรับ/ส่งข้อมูลได้
- ระยะสายไม่ควรเกิน 70 ซม. โดยประมาณ
- ถ้าคอมพิวเตอร์เป็นแบบ plugNplay ต้อง set ให้ parallel port ทำงานแบบธรรมชาติ

บทที่ 12 การส่งข้อมูลผ่าน Serial Port ของ PC

12.1 Serial Interfacing

การเชื่อมต่อระบบไมโครคอมพิวเตอร์กับอุปกรณ์ภายนอกส่วนใหญ่จะใช้การต่อประสานแบบขนาน (parallel tranmisson) กับแบบอนุกรม (serial transmission) สำหรับการต่อประสานแบบอนุกรมนิยมใช้กันมาก เช่น การเคลื่อนย้ายกันระหว่างไมโครคอมพิวเตอร์ หรือ อุปกรณ์เสริมต่างๆ เช่น mouse เป็นต้น

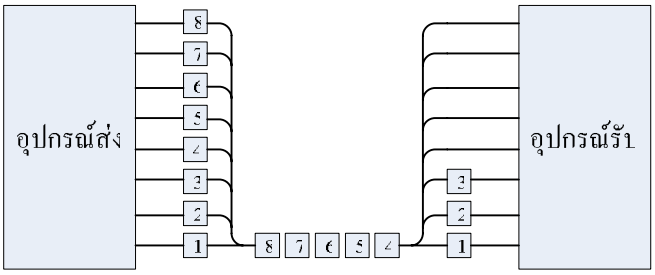
การต่อประสานแบบอนุกรมใช้ในการส่งและรับข้อมูลระยะไกล โดยส่งผ่านสายสัญญาณ 1 เส้น แทนที่จะเป็น 8 เส้นเหมือนการส่งผ่าน parallel port



รูปที่ 12.1 parallel vs serial interfacing

วิธีการถ่ายโอนข้อมูล

ในการถ่ายโอนข้อมูลแบบอนุกรมนั้น ข้อมูลจะได้รับการส่งออกมาครั้งละ 1 บิตระหว่างจุดรับและจุดส่ง จะเห็นว่ามีการส่งข้อมูลแบบอนุกรมนี้จะช้ากว่าการส่งข้อมูลแบบขนาน แต่ยังคงใช้อยู่ก็เพราะ ต้วกลางการสื่อสารต้องการช่องเดียวหรือมีสายเพียงคู่เดียวซึ่งจะประหยัดค่าใช้จ่าย ในการใช้ตัวกลางมากกว่าแบบขนานซึ่งถ้าเป็นระยะทางไกลจะดีเพราะเรามีระบบการสื่อสารทางโทรศัพท์ อยู่แล้ว จึงสามารถนำมาใช้ในการส่งข้อมูลแบบอนุกรมนี้ได้



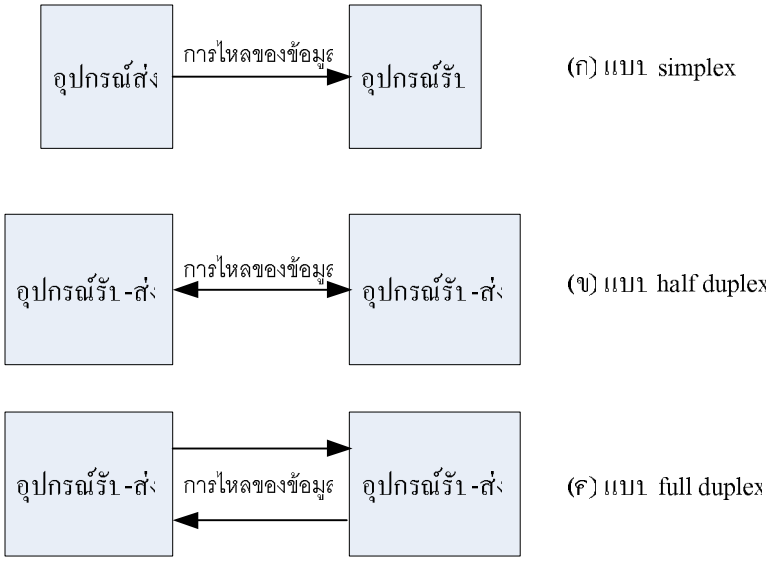
รูปที่ 12.2 การส่งข้อมูลแบบอนุกรม

การส่งข้อมูลแบบอนุกรม ข้อมูลจะถูกเปลี่ยนให้เป็นแบบอนุกรมเสียก่อนแล้วค่อยทยอยส่งครั้งละ 1 บิต ไปยังที่จะรับ ณ จุดรับจะต้องมีกลไกในการเปลี่ยนข้อมูลที่ส่งมาครั้งละบิตให้เป็นสัญญาณแบบขนาน ซึ่งลงตัวพอดี นั่นคือ บิตที่ 1 ลงที่บัสข้อมูลเส้นที่1พอดีการที่จะทำให้การแปลงสัญญาณจากแบบอนุกรมครั้งละบิตให้ลงพอดีนั้น จำเป็นต้องมีกลไกที่เหมาะสมเพื่อป้องกันการผิดพลาดจากการรับกลไกที่ว่าแบ่งออกเป็น 2 แบบ คือ แบบซิงโครนัส (synchronous) และแบบอะซิงโครนัส (asynchronous)

รูปแบบของการติดต่อสื่อสารแบบอนุกรม

การติดต่อแบบอนุกรมอาจจะแบ่งตามรูปลักษณะการส่งข้อมูลได้ 3 แบบคือ

1. แบบซิมเพลก (simplex) เป็นการส่งข้อมูลได้ทางเดียวเท่านั้น บางครั้งเรียกว่า การส่งทิศทางเดียว
2. แบบฮาล์ฟดูเพลกซ์ (half duplex) ทั้ง 2 สถานีสามารถรับและส่งได้แต่ ณ เวลาต่างกัน
3. แบบฟูลดูเพลกซ์ (full duplex) ทั้ง 2 สถานีสามารถรับและส่งได้ในเวลาเดียวกัน



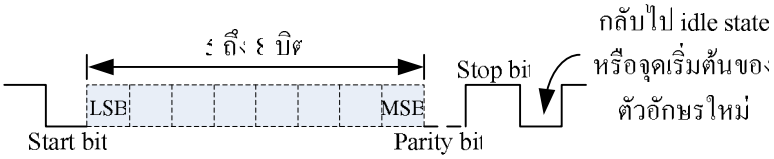
รูปที่ 12.3 รูปแบบของการติดต่อสื่อสารแบบอนุกรม

ความเร็วในการส่งถ่ายโอนข้อมูลแบบอนุกรม

ความเร็วของการถ่ายโอนข้อมูลแบบอนุกรมมีหน่วยวัดเป็น บิตต่อวินาที (bit per second: bps) แต่เรายังมีหน่วยที่นิยมใช้กันมากคือ อัตราโบด์ (baud rate) ซึ่งหมายถึง การเปลี่ยนแปลงของสัญญาณใน 1 วินาที หลายคนยังเข้าใจสับสนระหว่างหน่วยบีพีเอส กับอัตราโบด์ กล่าวคือ การเปลี่ยนแปลงของสัญญาณของสัญญาณ 1 ครั้งอาจจะแสดงถึง การส่งข้อมูลแบบอนุกรมมากกว่า 1 บิต อัตราการส่งข้อมูลเป็นจำนวนบิตจึงเท่ากับ อัตราโบด์คูณกับจำนวนบิตใน 1 โบด์

การสื่อสารแบบอะซิงโครนัส

การสื่อสารแบบนี้ประกอบด้วยบิตเริ่มต้น (start bit) และบิตสิ้นสุด (stop bit)



รูปที่ 12.4 stop และ stop bits ของการสื่อสารแบบอะซิงโครนัส

ขณะที่สถานะของการส่งเป็นแบบว่าง (idle) คือยังไม่มีสัญญาณที่ส่งออกมาแต่จะมีสัญญาณ หรือมีแรงดันตลอดเวลาเพื่อความแน่ใจว่าฝ่ายรับยังติดต่ออยู่กับฝ่ายส่งฝ่ายส่งจะเริ่มส่งข้อมูลบอกจุดเริ่มต้นสัญญาณของอะซิงโครนัสจะเป็น "0" หนึ่งช่วงสัญญาณนาฬิกา บิตนี้เรียกว่าบิตสตาร์ต ข้อมูล 1 ตัวอักษรที่ตามหลังบิตสตาร์ตจะมีขนาดตั้งแต่ 5 บิต จนถึง 8 บิต โดยอักขระนี้ส่วนมากจะนิยมใช้รหัสแอสกี

แรกเริ่มที่เดียวของการส่งข้อมูล จะส่งข้อมูลจะส่งรหัสโบดอด (Baudot code) ซึ่งใช้ 5 บิตในการแทนอักขระ 1 ตัว ส่วนที่ตามหลังข้อมูลก็จะเป็นบิตพาริตี้ ซึ่งจะอาจจะใช้หรือไม่ใช้ก็ได้บิตพาริตี้จะทำหน้าที่เป็นตัวตรวจสอบ ความถูกต้องของสัญญาณที่ได้รับ บิตพาริตี้อาจจะเป็นแบบคู่ (even) หรือแบบคี่ (odd) ก็ได้ หมายความว่า ถ้าหากเป็นพาริตี้คู่ จำนวนบิตที่เป็น "1" ในช่วงบิตข้อมูลกับบิตพาริตี้รวมกันแล้วต้องเป็นเลขคู่ผู้ส่งข้อมูล จะทำหน้าที่ตรวจสอบข้อมูลแล้วใส่บิตพาริตี้เอง

ฝ่ายรับเมื่อรับสัญญาณแล้วก็จะต้องตรวจสอบดูว่าเป็นจริงดังสถานการณ์ที่ตั้งไว้หรือไม่ หากผิดพลาดก็หมายความว่าสัญญาณที่รับนั้นผิดพลาดไปจากสถานีที่ส่งออกมาทั้งนี้ทั้งนั้นจะต้องผิดเป็นจำนวนคี่เท่านั้นคือ ผิดไป 1 บิต 3 บิต หรือ 5 บิตพร้อมกัน จึงจะตรวจสอบได้ว่าผิดเป็นจำนวนคู่ผลรวมของจำนวนบิตที่เป็น "1" ก็ยังเป็นคู่อยู่ดี

ทั้งนี้ทั้งนั้นไม่ได้หมายความว่าพาริตี้จะตรวจสอบการผิดพลาดเป็นจำนวนคู่ได้ความจริงแล้วสามารถตรวจสอบความผิดพลาด ได้เหมือนพาริตี้คู่แต่แทนที่จะตรวจสอบดูว่าสัญญาณ ที่รับเข้ามามีจำนวนคู่ ก็ตรวจสอบดูว่ามีจำนวนคี่หรือเปล่า อย่างไรก็ตาม โอกาสที่จะผิดพลาดเป็น 2 4 6 หรือ 8 บิตพร้อมกันมีน้อยมาก

ย้อนกลับมาดูสัญญาณอะซิงโครนัสใหม่ หลังจากบิตพาริตี้แล้วจะต้องมีบิตสต็อปซึ่งเป็น "1" ความกว้าง ของบิตสต็อปอาจจะเป็น 1 1.5 หรือ 2 พัลส์ของสัญญาณนาฬิกา ซึ่งแล้วแต่ผู้รับและผู้ส่งจะตกลงใช้กันเอง

การเริ่มใช้พอร์ตอนุกรมจึงจำเป็นต้องตั้งค่าต่างๆสำหรับการสื่อสาร ซึ่งมีดังต่อไปนี้ คือ

1. ความเร็วของการส่ง
2. ความยาวของรหัส 1 อักขระ

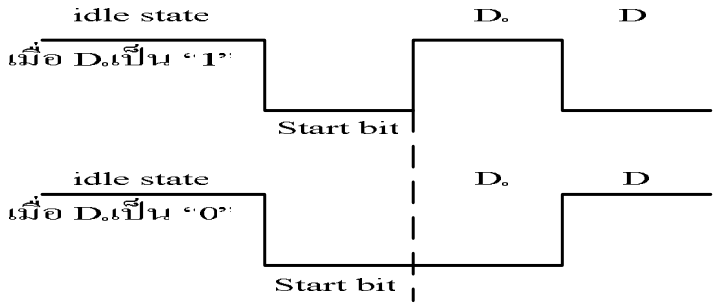
3. บิตตรวจสอบ
4. จำนวนบิตสต็อป

การสื่อสารแบบซิงโครนัส

ข้อแตกต่างระหว่างวงจรการส่งข้อมูลอนุกรมแบบซึงโครนัสและแบบอะซิงโครนัสก็คือ ความต่อเนื่องของข้อมูลที่ส่งในแบบซิงโครนัส ข้อมูลที่ส่งออกมาเป็นแบบต่อเนื่อง ไม่มีบิตสตาร์ตหรือบิตสต็อป หรือแม้กระทั่งบิตพาริตี้รูปแบบที่ใช้ในการส่งข้อมูลแบบซิงโครนัสจึงแตกต่างไปจากการส่งข้อมูล แบบอะซิงโครนัส เช่น รูปแบบของบริษัทไอบีเอ็ม ใช้รูปแบบไบซิงก์ (binary synchronous transmission)

การซิงโครไนซ์จะทำในระดับอักขระซึ่งหมายความว่าอักขระแต่ละตัวมีขอบเขตที่แน่นอนแต่ละอักขระ ไม่มีบิตสตาร์ต หรือบิตสต็อปเหมือนอะซิงโครนัส การซิงโครนัสจะกระทำที่จุดเริ่มต้นของการส่งข้อมูล สถานีส่งจะส่งสัญญาณที่เรียกว่า ตัวอักษรนำ (leading pad character)ไปยังสถานีรับก่อนที่จะเริ่มส่งข้อมูล ตัวอักษรนำจะประกอบด้วย "0" และ "1" สลับกัน เพื่อให้สถานีรับจัดสัญญาณนาฬิกาให้ตรงกันก่อนส่ง ข้อมูลก็จะมีการส่งอักขระที่เรียกว่า syn ตามหลังตัวอักษรนำออกมาสถานีส่งจำเป็นจะบอกความยาวของข้อมูลมาในกลุ่มนี้ และต้องบอกเครื่องหมายที่เป็นตัวบอกจุดเริ่มต้นของข้อมูลด้วย

ระยะเวลาและอัตราการส่งข้อมูล



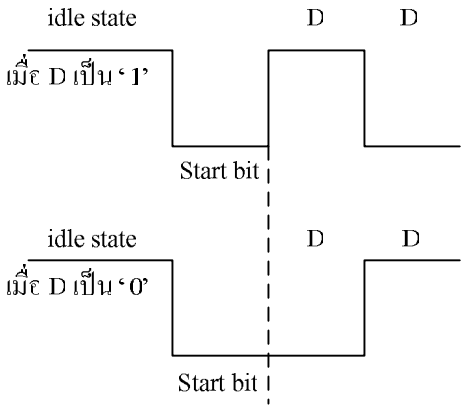
รูปที่ 12.5 การส่งข้อมูลที่มีขนาด 8บิตจากระบบไมโครโพรเซสเซอร์ส่งออกที่ช่องสื่อสาร แบบอนุกรม

ในการส่งข้อมูลแบบอนุกรมมีสิ่งที่สำคัญที่จะต้องพิจารณาคือ ความเร็วของข้อมูลในการส่งซึ่งเราเรียกว่า อัตราบิต (bit rate) ตามที่กล่าวมา และกรณีที่ให้อัตราการเปลี่ยนแปลงของสัญญาณ 1 ครั้งต่อข้อมูล 1 บิต จะได้อัตราบิตเท่ากับอัตราโบด์

บิตสตาร์ตและบิตสต็อป

การรับส่งข้อมูลแบบอะซิงโครนัสจะต้องมีการบอกจุดเริ่มต้นและจุดสิ้นสุดของเฟรม (frame) ข้อมูลเสมอ โดยปกติจะให้สภาวะไอเดิลเหมือนเช่นบิตสต็อป ดังนั้นส่วนของบิตสตาร์ตจะตรงข้ามกับไอ

เดิล โดยทั่วไปของการส่งข้อมูลจะใช้ 1 บิตที่เป็นลอจิก "0" เป็นตัวบอกบิตสตาร์ต ส่วนบิตสต็อปจะยาวกว่าที่กำหนดก็ได้ก่อนที่จะเริ่มต้นเฟรมใหม่



รูปที่ 12.6 การบอกบิตสตาร์ต

มาตรฐาน	ระยะทางที่ส่งได้	ความเร็วที่ส่งได้	แรงดันสำหรับ logic “1”	แรงดันสำหรับ logic “0”
RS-232	50 ฟุต หรือ 17 เมตร	20 Kbps	+5V ถึง +15V	-5V ถึง -15V
RS-422	4,000 ฟุต หรือ 1.3 กิโลเมตร	10 Mbps	+2V ถึง +6V	-2V ถึง -6V
RS-423	40 ฟุต หรือ 13 เมตร	100 Kbps	+4V ถึง +6V	-4V ถึง -6V

ตารางที่ 12.1 มาตรฐานของการสื่อสารแบบอนุกรม

ระยะทางไกลขึ้น ความเร็วที่ส่งต้องช้าลงเพื่อไม่ให้เกิดความผิดพลาดในข้อมูลที่รับส่งกัน เช่นถ้าจะส่งข้อมูลผ่าน RS-422 เป็นระยะทางถึงระยะทางสูงสุดที่ส่งได้ (1.3 กิโลเมตร) จะทำให้ความเร็วที่ส่งได้ไม่ถึงความเร็วสูงสุดที่ส่งได้คือ สามารถส่งด้วยความเร็วไม่ถึง 10 Mbps

รูปที่ 12.7 ตัวอย่างการต่อ remote terminal กับ computer โดยผ่าน modem

DTE = Data Terminal Equipment (คือ terminal หรือ computer)

DCE = Data Communication Equipment (คือ modem)

รูปที่ 12.7 การต่อระหว่าง DTE และ DCE

12.2 Handshaking

ทำให้การติดต่อสื่อสารทำได้อย่างถูกต้องและมีประสิทธิภาพ หลีกเลี่ยงปัญหาข้อมูลสูญหายได้ (เนื่องจากอุปกรณ์รับและอุปกรณ์ส่งไม่รู้กัน)

RTS = Request To Send

CTS = Clear To Send

TxD = Transmitted Data

SG = Signal Ground

DSD = Data Set Ready

DTR = Data Terminal Ready

DCD = Received line signal detector

RI = Ring Indicator

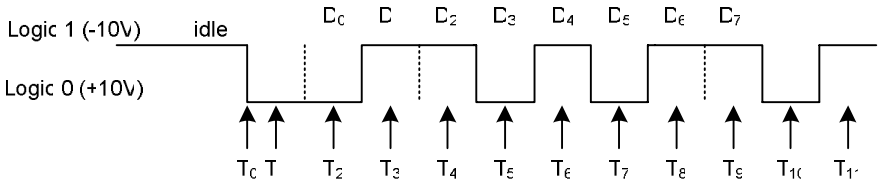
เครื่องหมาย – หมายถึง active low (-5V ถึง -15 V ทางด้านส่ง)

\+ หมายถึง active high (+5V ถึง +15 V ทางด้านส่ง)

12.3 Asynchronous Techniques

Asynchronous serial communication หมายถึง การส่งข้อมูลทีละ 1 character (1 byte)

Synchronous serial communication หมายถึง การส่งข้อมูลทีละ 1 block (หลายๆ 1 byte)



รูปที่ 12.8 การส่งข้อมูลขนาด 8 บิตโดยใช้เทคนิคอะซิงโครนัส

รูปที่ 12.8 แสดงถึงการส่งข้อมูลขนาด 8 บิตโดยใช้เทคนิคอะซิงโครนัส โดยที่

- T_0 – start
- T_1 – start bit double check
- T_2 - T_9 – ข้อมูลขนาด 8 บิต
- T_{10} – parity bit (ในรูปของ odd parity: จำนวนบิตของข้อมูลที่เป็น 1 รวมทั้ง parity bit รวมกันแล้วต้องเป็นเลขคี่)
- T_{11} – stop bit

Clock เป็นปัจจัยหลักของ serial communication เงื่อนไขข้อเดียวของ clock ใน asynchronous technique คือ clock ของตัวรับตัวส่งจะต้องใกล้เคียงกันมาก โดยทั่วไปจะใช้ clock ที่มีความถี่เป็น integer multiple ของ transmitter bit rate (16 เท่า - ใช้กันมากที่สุด)

IC ที่ใช้ใน serial communication เรียกว่า universal asynchronous receiver transmitter (UART) เช่น เบอร์ 8250UART

12.4 Decoding Serial Bit Streams and Error Detection

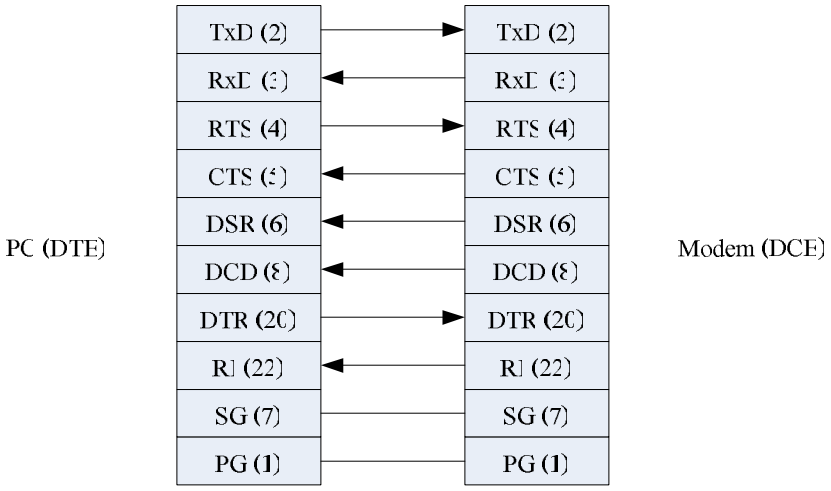
UART มีความสามารถในการ detect error ได้ 3 แบบคือ

1. Parity error: MSB เก็บค่าที่ทำให้จำนวนเลข “1” นั้นมีจำนวนคู่ (even) หรือคี่ (odd)
2. Overrun error: ไปเซ็ต overrun error flag ใน statua register เมื่อไมโครโพรเซสเซอร์ไปแนค่าข้อมูลที่ได้รับจาก data bus buffer ไม่ทันก่อนที่ค่าใหม่จะเข้ามา ซึ่งการทำงานที่ถูกต้องนั้นไมโครโพรเซสเซอร์ต้องไปอ่านค่าใน status register ก่อนอ่านค่าข้อมูลที่ได้รับ
3. Framing error: ถูกเซ็ตเมื่อ receiver ได้รับ 0 แทนที่จะเป็น 1 สำหรับ stop bit(s)

ในทางปฏิบัติเราอาจจะไม่สนใจ error เหล่านี้ก็ได้ แต่ใช้ซอฟต์แวร์ในการตรวจ error จากข้อมูลที่รับมาแทน วิธีการที่นิยมคือ

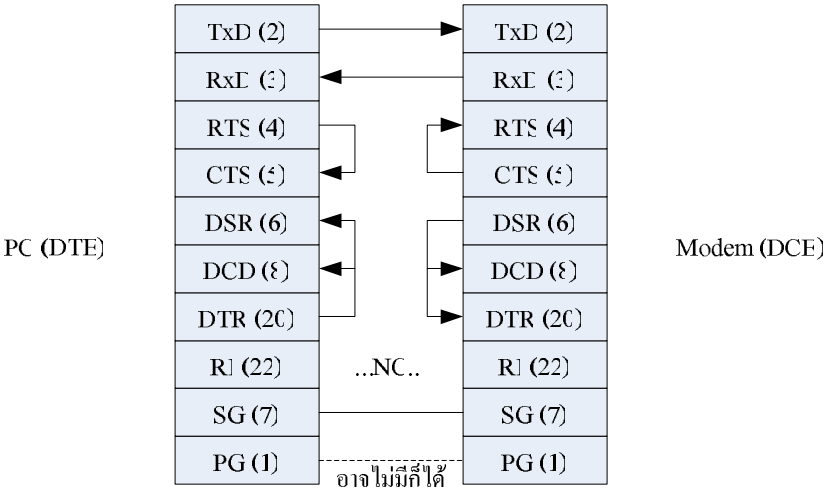
- Checksum
- Longtitudinal Redundancy Check (LRC)
- Cyclic Redundancy Check (CRC)

การต่อระหว่าง PC (DTE) เข้ากับ modem (DCE) โดยมีการใช้สัญญาณ handshaking



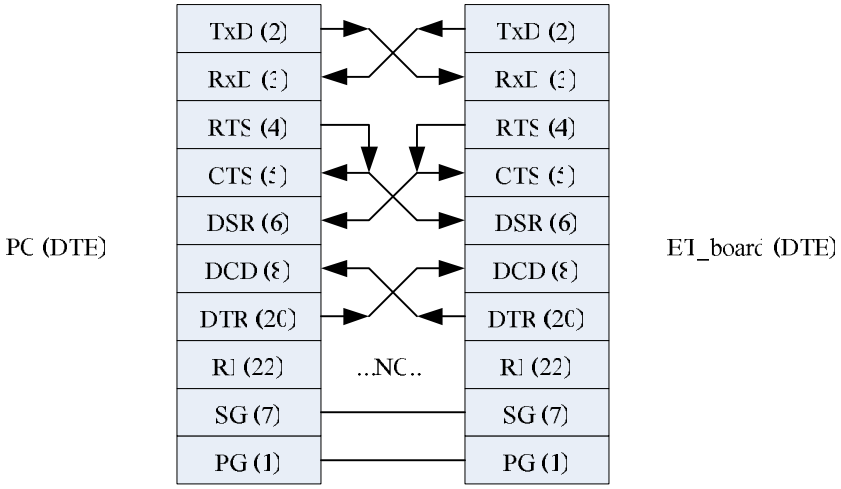
(ที่มา: Tompkins, *1988)*

การต่อระหว่าง PC (DTE) เข้ากับ modem (DCE) โดยไม่มีการใช้สัญญาณ handshaking



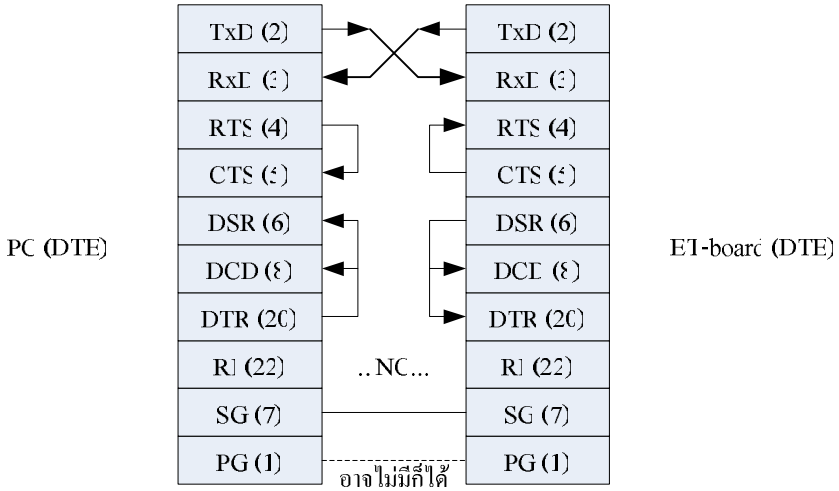
(ที่มา: Tompkins, *1988)*

การต่อระหว่าง DTE เข้ากับ DTE โดยตรงโดยไม่มีการใช้สัญญาณ handshaking



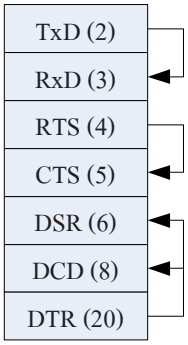
(ที่มา: Tompkins, *1988)*

การต่อระหว่าง DTE เข้ากับ DTE โดยตรงโดยมีการใช้สัญญาณ handshaking



(ที่มา: Tompkins, *1988)*

การตรวจสอบ UART มักจะต่อสาย connector เพื่อให้เกิด loop back ดังนี้



(ที่มา: Tompkins, *1988)*

เอกสารอ้างอิง

- **The Z80 Microprocessor: Architecture, Interfacing, Programming and Design**, Ramesh Gaonkar, Merrill Publishing Company, 1992.
- **Interfacing Sensors to the IBM PC**, Willis J. Tompkins และ John G. Webster, Prentice Hall, 1988.
- **Digital Systems**, Ronald J. Tocci, 5th Edition, Prentice-Hall, 1991.
- **ไมโครโพรเซสเซอร์ (Z80) I**, มงคล ทองสงคราม, พิมพ์ครั้งที่ 6, ห้างหุ้นส่วนจำกัด วี. เจ.พริ้นติ้ง, 2545
- **ไมโครโพรเซสเซอร์ (Z80) II**, มงคล ทองสงคราม, พิมพ์ครั้งที่ 4, ห้างหุ้นส่วนจำกัด วี. เจ.พริ้นติ้ง, 2542
- **บทความทางด้านไมโครโพรเซสเซอร์ (Z80)**, อาชีวะศึกษาจังหวัดนครราชสีมา http://203.172.183.132/chatree/ett_articles.htm
- **NECTEC's Web Based Learning**, http://www.nectec.or.th/courseware/computer/index.html
- **Electronics Tips**, มหาวิทยาลัยราชภัฏจันทรเกษม, http://elec.chandra.ac.th/tipntrick/z80/default.htm
- เอกสารประกอบการสอนรายวิชา Microprocessor Interfacing and Control Design, รศ.ยืน ภู่วรวรรณ, มหาวิทยาลัยเกษตรศาสตร์, http://www.cpe.ku.ac.th/~yuen/204471/
- เอกสารประกอบการสอนรายวิชา สถาปัตยกรรมไมโครโพรเซสเซอร์, รศ.ยืน ภู่วรวรรณ, มหาวิทยาลัยเกษตรศาสตร์, http://www.cpe.ku.ac.th/~yuen/204323/
- http://hyperphysics.phy-astr.gsu.edu/hbase/electronic/opampvar5.html
- เอกสารประกอบการสอนรายวิชา Microprocessor Interfacing Techniques, J.R. Drummond, University of Toronto http://faraday.physics.utoronto.ca/PHY406F/ln_intro.htm
- HyperPhysics, Department of Physics and Astronomy, Geogial State University, http://hyperphysics.phy-astr.gsu.edu/hbase/hframe.html
- Official Support-Page: Hardware - Software - Utilities - FAQ – Docs for Z80-Family http://www.z80.info/