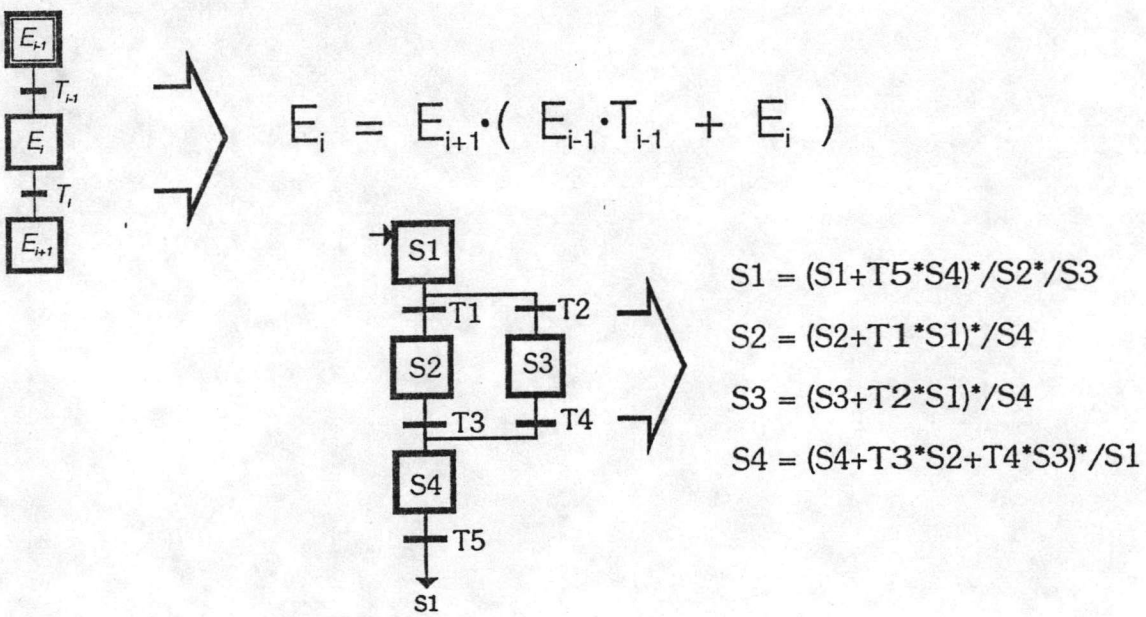
บทที่ 5

ตัวแปลภาษาฟังก์ชั่นชาร์ต

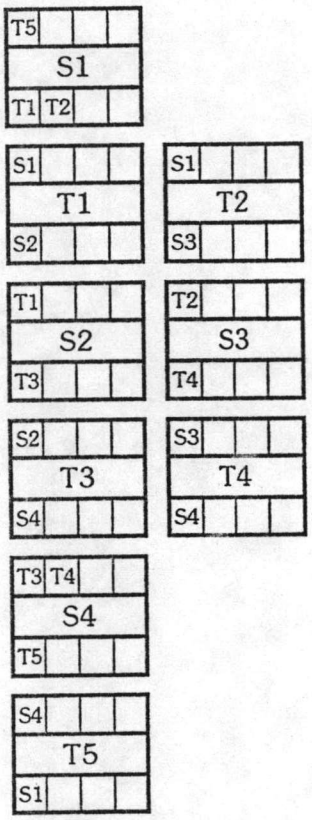
5.1 แนวความคิด

จากทฤษฎีการแปลในบทที่ 2 จึงทดลองนำมาแปลตัวอย่างชาร์ต เพื่อหาเทคนิคที่เหมาะสม ดังแสดงใน รูปที่ 5.1



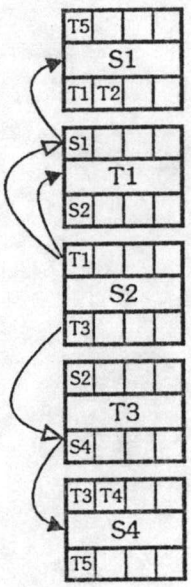
รูปที่ 5.1 : ตัวอย่างการแปล

จะสังเกตเห็นว่าในการแปล องค์ประกอบที่เป็นหลักก็คือสเต็ป กลุ่มของสมการที่ได้จะแสดงความสัมพันธ์ใดๆที่จะมีผลต่อการแอ็กทีฟของสเต็ป ความสัมพันธ์เหล่านั้นเป็นความสัมพันธ์ของสเต็ปและทรานสิชั่นที่อยู่รายรอบสเต็ปที่กำลังถูกพิจารณา จึงพอกำหนดโครงร่างในขั้นต้นได้ว่าจะต้องหยิบสเต็ปทุกๆสเต็ปขึ้นมาพิจารณาและแปลเป็นสมการบูลลีนของสเต็ปนั้น การแปลจะนำชาร์ตที่ได้จัดเก็บไว้มาแปล และจะพิจารณาเพียงแค่สเต็ปและทรานสิชั่นเท่านั้น เพราะการเชื่อมต่อของสเต็ปและทรานสิชั่นได้ถูกสร้างไว้แล้ว รูปที่ 5.2 แสดงตัวอย่างความสัมพันธ์ที่ได้สร้างไว้



รูปที่ 5.2 : ตัวอย่างความสัมพันธ์ที่ได้สร้างไว้

ในการแปลจะหยิบขึ้นมาพิจารณาครั้งละ 1 สเต็ป เพื่อสร้าง 1 สมการ จากตัวอย่างจึงทดลองพิจารณาสเต็ป S2 จะเห็นว่าในสมการความสัมพันธ์ที่ได้ ประกอบด้วยสเต็ป S1, S2, S4 และทรานสิชั่น T1 ซึ่งในการแปลจะต้องค้นหาสเต็ปเหล่านั้นมา สมมุติว่าเรากำลังยืนอยู่ที่สเต็ป S2 เมื่อมองขึ้นไปข้างบน (FromTransitionS) ก็จะเห็นเพียงทรานสิชั่น T1 มองลงไปข้างล่าง (ToTransitionS) ก็จะเห็นเพียง T3 การที่จะหา S2 ได้ จึงต้องไต่ขึ้นไปข้างบนไปถาม T1 และการหา S4 ก็จะต้องไต่ลงข้างล่างเพื่อถาม T3

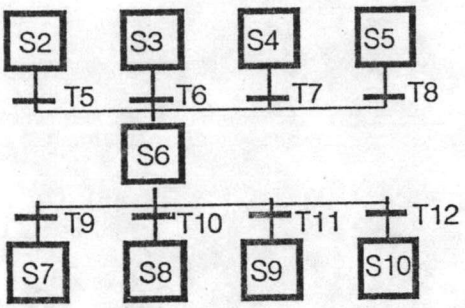


$$S2 = (S2 + T1*S1)^*/S4$$

รูปที่ 5.3 : ความสัมพันธ์ที่ต้องสร้างสำหรับ S2

ซึ่งกับทุกสเต็ปก็จะต้องสร้างความสัมพันธ์ในลักษณะนี้เช่นเดียวกัน กล่าวคือ เราทราบระยะห่างที่แน่นอนของสเต็ปและทรานสิชั่นที่จะต้องนำมาสร้างสมการบูลลีน แต่ยังไม่ทราบตำแหน่งและรายละเอียดภายในเซลล์ ณ จุดนี้จึงได้ทดลองสร้างโปรแกรมย่อย FindElement เพื่อใช้ในการค้นหาสเต็ปและทรานสิชั่นที่ต้องการโดยอาศัยโครงข่ายการเชื่อมโยงที่ได้สร้างไว้แล้ว ตัวแปรที่ FindElement ต้องการ ได้แก่ ทิศทางที่จะค้นหา, ระยะทางที่จะค้นหาไกลกี่ช่วง (จากสเต็ปถึงทรานสิชั่นที่อยู่ติดกันเรียกว่า 1 ช่วง), ตำแหน่งของจุดเริ่มต้น, เส้นทางที่จะใช้ (เพราะสเต็ปสามารถเชื่อมโยงกับทรานสิชั่นได้ทิศทางละ 4 ตัว, จึงต้องระบุทรานสิชั่นที่จะใช้เป็นทางผ่าน) ผลลัพธ์ที่ได้จากการค้นหาได้ออกแบบให้ FindElememt นำหมายเลขและตำแหน่งมาให้ หมายเลขนำไปใช้เขียนสมการได้ทันที ส่วนตำแหน่งสำหรับใช้ค้นหารายละเอียดอื่นๆ

จากนั้นทดลองแปลสมการบูลลีนไปเป็นภาษานิมอนิค เพื่อจะได้สรุปหาโครงสร้างข้อมูลที่เหมาะสมที่จะใช้รองรับการแปล ทั้งนี้พิจารณาจากกรณีที่ใช้โครงสร้างการเชื่อมต่อครบทุกฟิลด์และทุกทิศทาง จากรูปที่ 5.4 พิจารณา S6 เพราะใช้โครงสร้างการเชื่อมต่อครบถ้วน จึงสามารถใช้เป็นตัวแทนของสเต็ปใดๆได้ จากรูปแบบภาษานิมอนิคที่ได้กับหมายเลขของสเต็ปและทรานสิชั่น สามารถวิเคราะห์ออกมาเป็นความสัมพันธ์ได้ดังตารางที่ 5.1



S6 = (S6 + T5S2 + T6S3 + T7S4 + T8S5)*/S7*/S8*/S9*/S10

LD S6	LD T7	ANI S7
LD T5	AND S4	ANI S8
AND S2	ORB	ANI S9
ORB	LD T8	ANI S10
LD T6	AND S5	OUT S6
AND S3	ORB	
ORB		

รูปที่ 5.4 : การแปลไปเป็นภาษานีมอนิค

	S6	T5	S2	T6	S3	T7	S4	T8	S5	S7	S8	S9	S10
MNEMONIC	LD	LD	AND	LD	AND	LD	AND	LD	AND	ANI	ANI	ANI	ANI
CODE	L	L	A	L	A	L	A	L	A	I	I	I	I

ตารางที่ 5.1 : ผลจากการวิเคราะห์

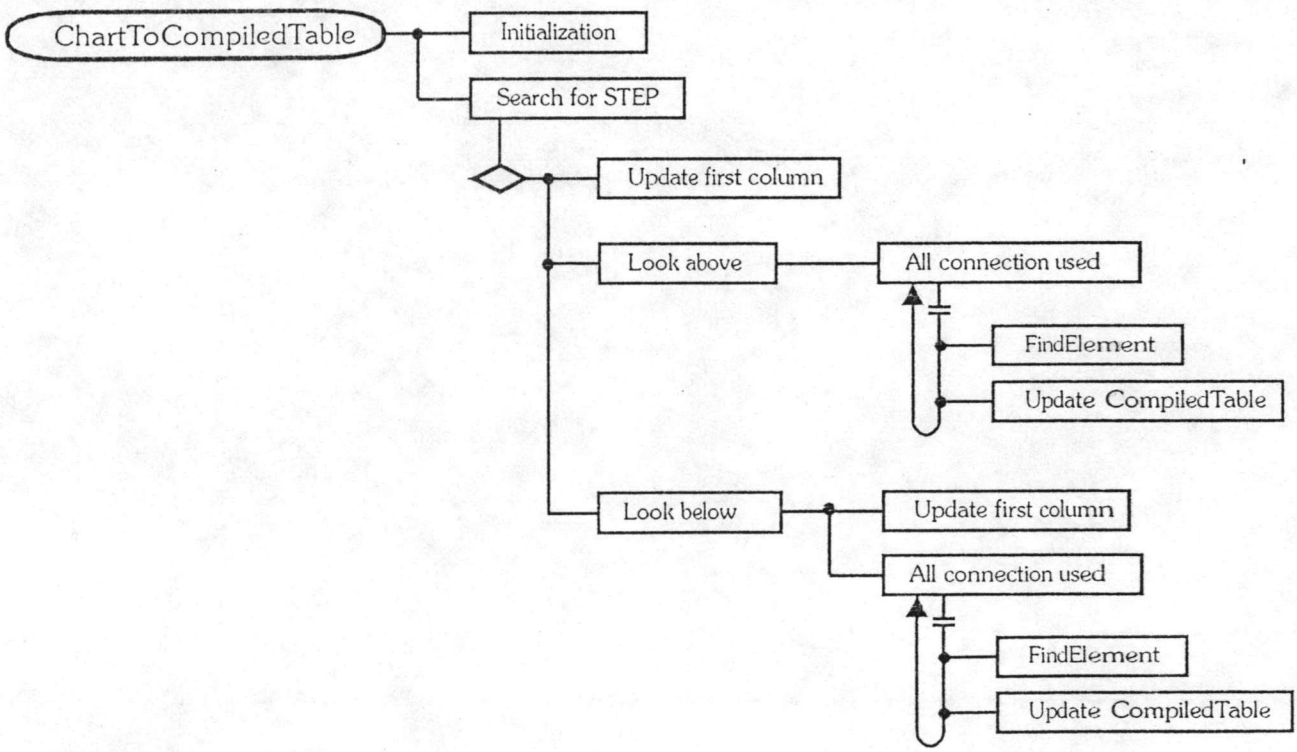
จะเห็นว่าผลการวิเคราะห์ที่ได้ สามารถนำไปแปลเป็นสมการบูลลีน และภาษานีมอนิคได้โดยง่าย จึงได้ออกแบบโครงสร้างข้อมูลที่จะรองรับการแปลในขั้นแรกตามผลการวิเคราะห์ในตารางที่ 5.1

```
MaxStep  =  162;
NameCoor  =  record
                Name :  string(3);
                Coor  :  Coordinate;
             end;
CompiledCell = record
                Data : NameCoor;
                Code : Char;
             end;
CompiledTable = array(1.Maxstep,1.13) of CompiledCell;
```

โครงสร้างที่ได้เป็นตารางเรียกว่า CompiledTable มีความกว้าง 13 คอลัมน์ จำนวนแถวเท่ากับค่าใน MaxStep เพราะ 1 แถว จะเก็บความสัมพันธ์สำหรับ 1 สเต็ป ส่วนค่า MaxStep ได้จากการประมาณจำนวนสเต็ปสูงสุดใน 1 ชาร์ต (40 แถว x 17 คอลัมน์) แต่ละเซลล์ของ CompiledTable ประกอบด้วยหมายเลข , ตำแหน่งและรหัส (Code)

ดังนั้นในการแปลชาร์ต ขั้นแรกจะแปลชาร์ตให้เป็น CompiledTable ก่อน จากนั้นจึงแปล CompiledTable ที่ได้ไปเป็นสมการบูลลีนหรือภาษานีมอนิคตามต้องการ

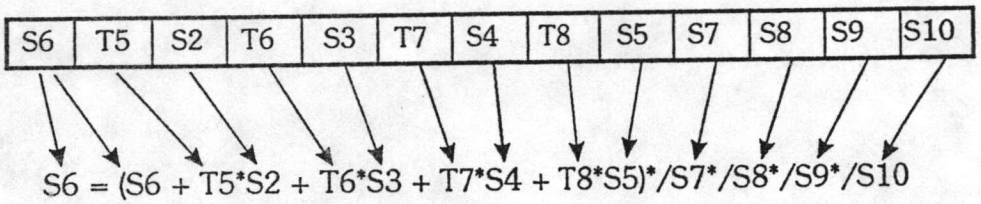
การแปลชาร์ตเป็น CompiledTable จะต้องค้นหาสเต็ปทุกๆสเต็ป และหยิบขึ้นมาวิเคราะห์ทีละสเต็ป การวิเคราะห์จะอาศัยโปรแกรมย่อย FindElement ที่ได้สร้างไว้แล้ว ช่วยค้นหาสเต็ปหรือทรานสิชั่นที่ต้องการ



รูปที่ 5.5 : DSD ของการแปลเป็น CompiledTable

5.2 การแปลเป็นสมการบูลลีน

พิจารณา CompiledTable เทียบกับสมการบูลลีน ดังรูปที่ 5.6 จะเห็นว่าสามารถสร้างสมการบูลลีนได้โดยตรง โดยอ่านหมายเลขขึ้นมาและเขียนเครื่องหมายต่างๆรวมเข้าไป เมื่อจัดการจนหมด 1 แถว ก็จะได้สมการบูลลีน 1 สมการ ขั้นตอนถัดไปจึงเก็บสมการบูลลีนเหล่านี้ลงแฟ้มข้อมูลในรูปแบบของ "Text file" และตั้งชื่อแฟ้มข้อมูลตามชื่อของชาร์ต ส่วนนามสกุลใช้ ".BLN"



รูปที่ 5.6 : เปรียบเทียบ CompiledTable กับสมการบูลลีน

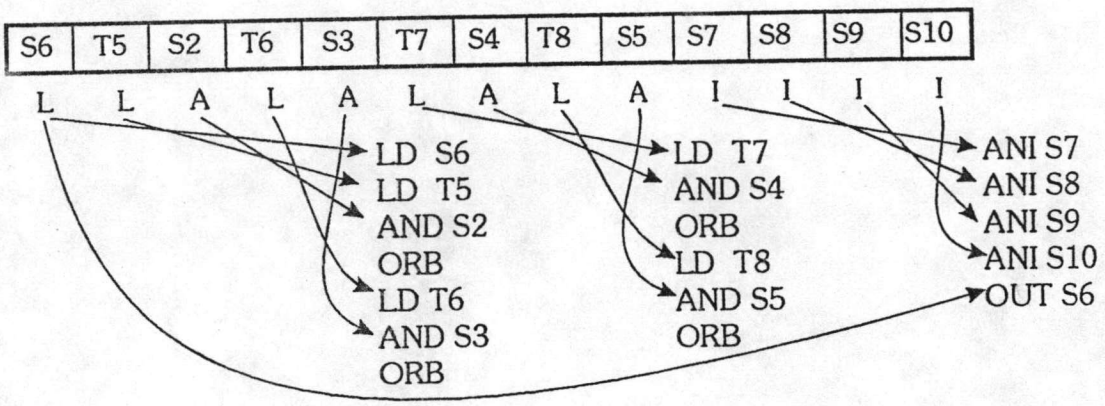
```
Procedure CompileToBooleanEquation(BR:byte; Var Eq:String);
var A : string[3];
    i : byte;
Begin
     A:=Compiled[BR,1].Data.Name;
     Eq:=A+'=('+A;
     For i:=1 to 4 do
         begin
           If (Compiled[BR,i*2].Data.Name<>'') then
              begin
                   A:=Compiled[BR,i*2].Data.Name;
                   Eq:=Eq+'+'+A;
              end;
           If (Compiled[BR,i*2+1].Data.Name<>'') then
              begin
                   A:=Compiled[BR,i*2+1].Data.Name;
                   Eq:=Eq+'*'+A;
              end;
         end;
     Eq:=Eq+')';
     For i:=1 to 4 do
         If (Compiled[BR,9+i].Data.Name<>'') then
            begin
                 A:=Compiled[BR,9+i].Data.Name;
                 Eq:=Eq+'*/'+A;
            end;
End;

Procedure CompiledTableToBooleanTextFile;
var BLFile    : Text;
    WriteFile : string[12];
    Equation  : String;
    Row       : byte;
Begin
     Row:=1;
     If fMerge then WriteFile:=MergeFile+'.BLN'
               else WriteFile:=CurrentFile+'.BLN';
     Assign(BLFile,WriteFile);
     Rewrite(BLFile);
     While (Compiled[Row,1].Data.Name<>'') do
           begin
             CompileToBooleanEquation(Row,Equation);
             Writeln(BLFile,Equation);
             Inc(Row);
           end;
     Close(BLFile);
```

รูปที่ 5.7 : โปรแกรมการแปลเป็นสมการบูลลีน

5.3 การแปลเป็นภาษานีมอนิค

ขั้นตอนนี้จะแปล CompiledTable ให้เป็นภาษานีมอนิคโดยจัดเก็บเป็นแฟ้มข้อมูลประเภทข้อความ (Text file) พิจารณาการ เปรียบเทียบ CompiledTable กับคำสั่งภาษานีมอนิคที่จะต้องแปลให้ได้ ดังรูปที่ 5.8 จะเห็นว่าจะสามารถแปลได้โดยตรง โดย 1 บรรทัดของ CompiledTable จะสามารถแปลได้คำสั่ง 1 ชุด สำหรับ 1 สเต็ป แต่ละคอลัมน์จะแปลได้เป็นคำสั่ง 1 บรรทัด คำสั่ง 1 บรรทัด ประกอบด้วย 2 ส่วน ส่วนแรกเป็นคำสั่งซึ่งพิจารณาได้จากฟิลด์ Code ส่วนที่สองเป็นหมายเลขสเต็ป



รูปที่ 5.8 : เปรียบเทียบ CompiledTable กับคำสั่งภาษานีมอนิคที่ได้

ที่กล่าวมาในย่อหน้าที่แล้วเป็นแนวความคิดหลักในการแปลในส่วนนี้ แต่ผลลัพธ์ยังไม่สามารถนำไปใช้งานได้ เนื่องจากมีรายละเอียดในอีกหลายส่วนที่จะต้องจัดการเพิ่มเติมดังจะกล่าวถึงต่อไป

5.3.1 การออกแบบ

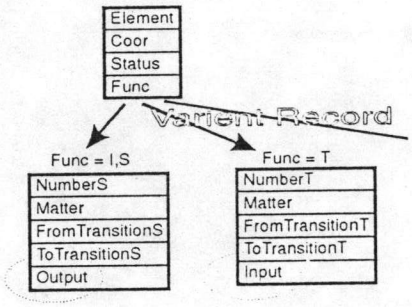
จากคำสั่งภาษานีมอนิคในรูปที่ 5.8 ตามความเป็นจริง ในส่วนที่สองจะต้องใช้หมายเลขหน้าสัมผัสของรีเลย์แทน (ซึ่งผู้ใช้ได้ป้อนไว้แล้ว) แต่ในการออกแบบจะยังมิใช้หมายเลขหน้าสัมผัสตามที่ระบุไว้ในทันที แต่จะกำหนดหมายเลขของรีเลย์ช่วย (Auxiliary Relay) ให้ใช้แทนไปก่อน ทั้งนี้เพื่อช่วยให้การจัดการในรายละเอียดต่างๆทำได้ง่ายขึ้น ดังจะกล่าวในหัวข้อถัดไป จึงได้ออกแบบวางโครงร่างของโปรแกรมภาษานีมอนิคที่จะแปลให้แบ่งเป็น 3 ส่วน ได้แก่

ส่วนที่ 1 : แสดงการนำอินพุตมาใช้งาน และการกำหนดหมายเลขของรีเลย์ช่วยให้กับอินพุตต่างๆ

ส่วนที่ 2 : เป็นโปรแกรมที่ได้จากการแปลชาร์ตโดยตรงตามเทคนิควิธีที่ได้กล่าวข้างต้น แต่จะนำหมายเลขรีเลย์ช่วยมาใช้

ส่วนที่ 3 : ประมวลเอาต์พุตจากรีเลย์ช่วยซึ่งเป็นผลจากการทำงานในส่วนที่ 2 ให้ได้เป็นหมายเลขรีเลย์เอาต์พุตตามที่ผู้ใช้ได้กำหนดไว้

จึงขอนำโครงสร้างข้อมูลที่ได้ออกแบบเตรียมไว้ สำหรับงานในส่วนนี้ ซึ่งได้แก่โครงสร้างในส่วนเอาต์พุตของสเต็ปและอินพุตของทรานสิชั่น ดังแสดงในรูปข้างล่าง



STEP	OUTPUT 1	OUTPUT 2	OUTPUT 3
	Output contact	Auxiliary relay	Supporting data
	Normal relay Timer Counter	Assigned by system	Normal,Set,Reset Constant Constant

TRANSITION	INPUT 1	INPUT 2	INPUT 3
	Input contact	Auxiliary relay	Supporting data
	Normal relay Timer Counter	Assigned by system	Normal or Invert

รูปที่ 5.9 : โครงสร้างข้อมูลของชาร์ตในส่วนอินพุตและเอาต์พุต

เอาต์พุตและอินพุตในฟิลด์ที่สอง คือส่วนที่ได้ออกแบบเตรียมไว้สำหรับการกำหนดหมายเลขของรีเลย์ช่วยเพิ่มเติมเข้าไป และได้เขียนโปรแกรมย่อย AssignAuxiliaryRelay เพื่อช่วยกำหนดหมายเลขของรีเลย์ช่วยให้กับสเต็ปและทรานสิชั่นทุกๆตัว การอ้างอิงถึงสเต็ปและทรานสิชั่นใดๆในขณะนี้ จึงอ้างอิงได้ด้วยรีเลย์ช่วยที่กำหนดให้ ซึ่งในการแปลในส่วนที่ 2 จะใช้หมายเลขของรีเลย์ช่วยในฟิลด์ที่ 2 นี้ ส่วนการแปลในส่วนที่ 1 และ 3 ก็จะใช้โครงสร้างข้อมูลในส่วนนี้ทั้ง 3 ฟิลด์

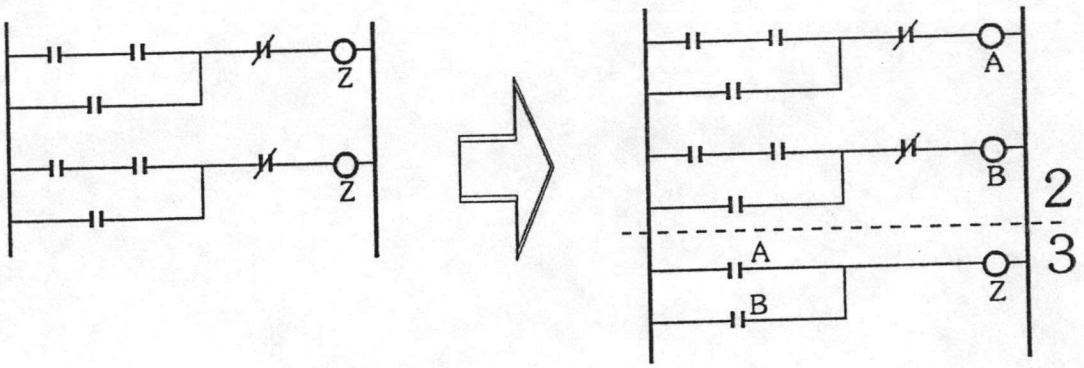
จากการออกแบบได้ทดลองกับ PC ของออมร่อน (OMRON) จนเป็นที่น่าพอใจ จึงได้ออกแบบปรับโครงสร้างให้สามารถแปลไปเป็นภาษานีมอนิคของ PC อื่นๆได้อีก ซึ่งจะกล่าวถึงในหัวข้อถัดไป ผลลัพธ์จากการแปลทั้ง 3 ส่วน จะถูกลำเลียงนำไปจัดเก็บลงแฟ้มข้อมูล โดยใช้ชื่อแฟ้มเดิม แต่นามสกุลจะเปลี่ยนไปตามยี่ห้อของ PC ตามที่ผู้ใช้กำหนด

5.3.2 <u>รายละเอียดการจัดการ</u>

เอาต์พุตซ้ำกัน

ปัญหาการเขียนโปรแกรมนีมอนิคที่ใช้เอาต์พุตซ้ำกัน เป็นปัญหาทางเทคนิคของ PC ซึ่งในการเขียนโปรแกรมนีมอนิคผู้ใช้จะต้องระมัดระวังการเขียนเอง และเพื่อให้เจตนารมณ์ของผู้ใช้ที่ต้องการใช้สเต็ปหลายๆสเต็ปไปขับเอาต์พุตตัวเดียวกันได้ไม่ผิดเพี้ยนไป จึงได้ออกแบบโปรแกรมให้สามารถจัดการกับปัญหานี้ได้ โดยการนำรีเลย์ช่วยมาใช้งานในการ

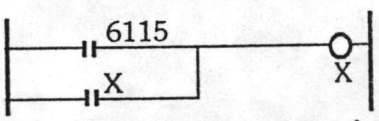
แปลในส่วนที่สอง ก็จะทำให้ไม่มีการใช้เอาต์พุตซ้ำกันในโปรแกรมนีมอนิคที่ได้ และการแปลในส่วนที่สามก็จะนำหมายเลขของรีเลย์ช่วยของสเต็ปเหล่านั้นมาออร์ (OR) กัน และขับเอาต์พุตตัวที่ต้องการก็จะสามารถแก้ปัญหานี้ได้



รูปที่ 5.10 : การแก้ปัญหาการใช้เอาต์พุตซ้ำกัน

สเต็ปเริ่มต้น

เนื่องจากสเต็ปเริ่มต้นเป็นสเต็ปแรกที่จะทำงานเมื่อเปิดเครื่อง จึงต้องหาแนวทางที่จะทำให้โปรแกรมนีมอนิคที่ได้ทำงานในส่วนนี้ได้ถูกต้อง ใน PC ส่วนใหญ่จะมีรีเลย์ช่วยพิเศษ (Special auxiliary relay) อยู่ 1 ตัวที่จะให้พัลส์ (Pulse) 1 ลูกทันทีที่สั่งให้ทำงาน (run) ใน PC ของออมร่อนเครื่องนี้[1] (C-500) ได้แก่รีเลย์หมายเลข "6115" จึงออกแบบให้นำรีเลย์ช่วยพิเศษตัวนี้ มาขับสเต็ปเริ่มต้นทุกๆสเต็ป (โดยขับรีเลย์ช่วยประจำสเต็ปนั้น) แต่เนื่องจากรีเลย์ช่วยตัวนี้จะให้พัลส์เพียง 1 ลูก จึงต้องนำสเต็ปเริ่มต้นเหล่านั้นไปคงค่า (Hold) ตัวเองไว้ด้วย



6115 = Auxiliary relay that give first pulse
X = Auxiliary relay of initial step

รูปที่ 5.11 : การกระตุ้นสเต็ปเริ่มต้น

---

1. OMRON. OMRON user's manual. Programmable Controller model SYSMAC C-500. Japan : OMRON Tateisi electronics co. p. A-3.

เงื่อนไขเป็นจริงตลอด

เนื่องจาก PC มีรีเลย์ช่วยพิเศษที่จะให้เอาต์พุตที่เป็นจริง ตลอดเวลาการทำงาน ได้แก่ รีเลย์ช่วยพิเศษหมายเลข "6113" จึงนำมาใช้โดยออกแบบให้โปรแกรมย่อย AssignAuxiliaryRelay กำหนดค่า 6113 ให้กับทรานสิชั่นทุกตัวที่มีเงื่อนไขเป็นจริงตลอด และถือว่าทรานสิชั่นเหล่านี้รับอินพุตจากรีเลย์หมายเลข 6113

เงื่อนไขเป็นนิเสธ

เมื่อตรวจสอบพบว่ามีการใช้นิเสธในทรานสิชั่นใด (ตรวจสอบจากข้อมูลของชาร์ตของทรานสิชั่นนั้นในฟิลด์ Input(3) ว่าเท่ากับตัวอักษร "!" หรือไม่) การแปลโปรแกรมนีมอนิคในส่วนที่ 1 ในการนำอินพุตนั้นมาใช้งาน ก็จะใช้คำสั่ง "LD NOT" แทนคำสั่ง "LD" ดังนั้นในการแปลโปรแกรมในส่วนที่ 2 ก็จะแปลไปตามปกติได้

เอาต์พุตเป็นตัวตั้งเวลา

ดังได้กล่าวไปแล้วว่า เมื่อผู้ใช้กำหนดให้ใช้เอาต์พุตเป็นตัวตั้งเวลา ระบบก็จะสอบถามค่าคงที่ในการตั้งเวลา ในการแปลในส่วนที่ 1 เมื่อตรวจสอบพบก็จะใช้คำสั่ง "LD TIM" แทนคำสั่ง "LD" และ "LD NOT TIM" แทนคำสั่ง "LD NOT" ซึ่งเป็นส่วนที่กำหนดการนำรีเลย์ช่วยมาใช้แทน ทำให้ในส่วนที่ 2 สามารถแปลได้ตามปกติ แต่ในส่วนที่ 3 ซึ่งเป็นการขับ เอาต์พุตนั้น จะต้องปรับปรุงให้ใช้คำสั่งขับตัวตั้งเวลาแทนคำสั่ง "OUT" ดังนี้

TIM X	X : หมายเลขของตัวตั้งเวลา
#K	K : ค่าคงที่การตั้งเวลา

เอาต์พุตเป็นตัวนับ

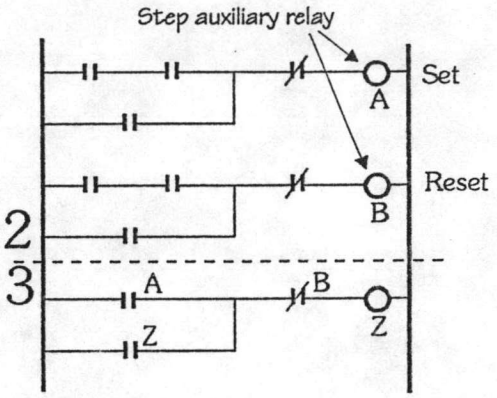
จะต้องจัดการในลักษณะคล้ายกับตัวตั้งเวลา กล่าวคือ ในการแปลส่วนที่ 1 จะใช้คำสั่ง "LD CNT" และ "LD NOT CNT" แทน แต่ในส่วนที่ 3 จะซับซ้อนกว่า เนื่องจากการใช้ตัวนับจะมีอีกสเต็ปมาทำหน้าที่รีเซ็ทตัวนับตัวนี้ ในการแปลจะต้องค้นหาสเต็ปนั้นและนำมาใช้แปลร่วมกัน คำสั่งที่ใช้ขับตัวนับมีรูปแบบดังนี้

LD X	X : หน้าสัมผัสที่ใช้นับ (clock pulse)
LD Y	Y : หน้าสัมผัสที่ใช้รีเซ็ท
CNT Z	Z : หมายเลขของตัวนับ
#K	K : ค่าคงที่การนับ

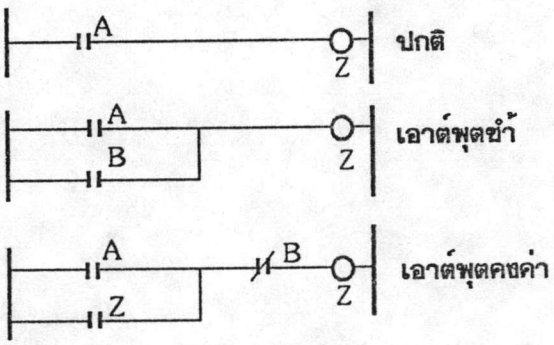
โดยหน้าสัมผัสที่ใช้เป็น clock pulse ก็คือ หมายเลขของรีเลย์ช่วยของสเต็ปที่เรียกใช้ตัวนับ หน้าสัมผัสที่ใช้รีเซ็ท ก็คือ หมายเลขของรีเลย์ช่วยของสเต็ปที่รีเซ็ทตัวนับตัวนั้น

เอาต์พุตประเภทคงค่า

ในการเรียกใช้นั้น ผู้ใช้กำหนดสเต็ปหนึ่งให้เซ็ทค่า (คงค่าเอาต์พุตให้แอ็กทีฟไว้) และใช้อีกสเต็ปหนึ่งสำหรับรีเซ็ทเอาต์พุตตัวนั้น การจัดการในการแปลจะคล้ายกับกรณีเอาต์พุตซ้ำกัน โดยจัดการในการแปลส่วนที่ 3 แต่จะต้องนำเอาต์พุตมาคงค่าตัวเองไว้ เพราะหน้าสัมผัสของรีเลย์ช่วยของสเต็ปที่ใช้เซ็ทจะแอ็กทีฟเฉพาะในช่วงที่สเต็ปแอ็กทีฟเท่านั้น



รูปที่ 5.12 : การออกแบบจัดการเอาต์พุตประเภทคงค่า



รูปที่ 5.13 : เปรียบเทียบการจัดการเอาต์พุตประเภทต่างๆ

รูปที่ 5.13 แสดงการเปรียบเทียบการจัดการเอาต์พุตประเภทต่างๆในการแปลส่วนที่ 3 หน้าสัมผัส A, B คือ หน้าสัมผัสของรีเลย์ช่วยของสเต็ป และ Z คือ หน้าสัมผัสของเอาต์พุต

การแปลเป็นภาษานีมอนิคของ PC ยี่ห้ออื่นๆ

เพื่อให้สามารถแปลซาร์ตไปเป็นภาษานีมอนิคของ PC ได้หลายๆยี่ห้อ จึงได้ออกแบบนำคำสั่งภาษานีมอนิคที่ใช้ในการแปล แยกออกไปเก็บไว้ในแฟ้มข้อมูลต่างหาก และออกแบบให้ผู้ใช้สามารถแก้ไขเพิ่มเติมคำสั่งภาษานีมอนิคของ PC ยี่ห้ออื่นได้อีก โดยกำหนดรูปแบบที่แน่นอนไว้ และตั้งชื่อแฟ้มข้อมูลดังกล่าวว่า "BRANDS.MNE" ซึ่งมีโครงสร้างดังนี้ (ตัวอย่างคำสั่งของออมร่อน)

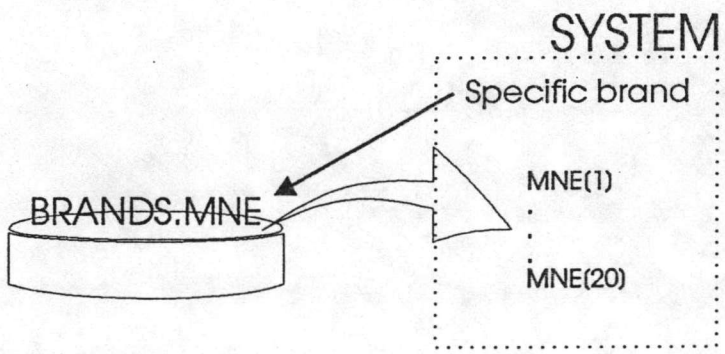
บรรทัดที่	ข้อความภายในแฟ้ม	หมายเหตุ
1	#OMRON	ยี่ห้อ, ขึ้นต้นด้วย "#" ตามด้วยตัวอักษรไม่เกิน 8 ตัว
2	OMR	นามสกุลของแฟ้มข้อมูลภาษานีมอนิคที่ต้องการ
3	6115	รีเลย์ช่วยพิเศษที่ให้พัลส์ 1 ลูกทันทีที่สั่งให้ PC ทำงาน
4	6113	รีเลย์ช่วยพิเศษที่แอ็กทีฟตลอดเวลาการทำงานของ PC
5	LD	คำสั่ง LD
6	LD NOT	คำสั่ง LOAD NOT
7	AND	
8	AND NOT	
9	OR	
10	OR NOT	
11	OR LD	คำสั่ง OR BLOCK
12	OUT	
13	TIM	คำสั่งในการเรียกใช้ตัวตั้งเวลา เช่น LD TIM 001
14	CNT	คำสั่งในการเรียกตัวนับ เช่น LD CNT 120
15	TIM	คำสั่งที่ใช้ขับตัวตั้งเวลา เช่น TIM 001
16	#	# 0010
17	CNT	คำสั่งที่ใช้ขับตัวนับ เช่น CNT 120
18	#	# 0003
19	END	คำสั่งสิ้นสุดโปรแกรม
20		เว้น 1 บรรทัด

ในการเพิ่มเติมคำสั่งของ PC อื่นๆ ทำได้โดยการนำแฟ้มข้อมูล "BRANDS.MNE" มาแก้ไข ซึ่งสามารถใช้ Text editor ใดๆก็ได้ ทั้งนี้ผู้ใช้จะต้องเขียนเพิ่มเติมต่อท้ายข้อมูลเดิม โดยอ้างอิงตามรูปแบบที่ได้กำหนดไว้ ในการออกแบบได้สร้างแฟ้มข้อมูลที่บรรจุคำสั่งของ PC ไว้ 4 ยี่ห้อ ได้แก่ OMRON, MITSUBISHI, GE, IZUMI ในรูปที่ 5.14 แสดงข้อมูลภายในแฟ้ม BRANDS.MNE ที่สร้างไว้

```
#OMRON              #MITZU
OMR                 MZB
6115                M8002
6113                M8000
LD                  LD
LD   NOT            LDI
AND                 AND
AND  NOT            ANI
OR                  OR
OR   NOT            ORI
OR   LD             ORB
OUT                 OUT
TIM                 T
CNT                 C
TIM                 OUT T
  #                     K
CNT                 OUT C
  #                     K
END                 END
```

รูปที่ 5.14 : BRANDS.MNE

ในการสั่งให้แปลชาร์ตเป็นภาษานีมอนิคนั้น ขั้นแรกผู้ใช้จะต้องกำหนดก่อนว่าจะแปลไปเป็นภาษานีมอนิคของ PC ใด โดยการเลือกคำสั่ง "BRAND" ระบบก็จะให้ป้อนชื่อยี่ห้อ ซึ่งจะต้องตรงกับชื่อที่ระบุไว้ในแฟ้ม BRANDS.MNE จากนั้นระบบจะอ่านคำสั่งภาษานีมอนิคของยี่ห้อที่กำหนดมาเก็บไว้ในตัวแปร MNE(1) ถึง MNE(20) เพื่อใช้อ้างอิงในการแปล



รูปที่ 5.15 : การใช้งานแฟ้ม BRANDS.MNE

การจัดการในส่วนนี้เขียนฟังก์ชั่น RetrieveBrandCode ซึ่งจะให้ค่าจริงเมื่อสามารถโหลด (Load) คำสั่งที่ต้องการขึ้นมาได้ และจะให้ค่าเท็จเมื่อไม่สามารถโหลดคำสั่งได้สำเร็จ สาเหตุที่เขียนการจัดการเป็นฟังก์ชั่นก็เพื่อให้สามารถสั่งโหลด และตรวจสอบได้รวบรัด

```
Function RetrieveBrandCode:boolean;
Var
   MNEfile  : Text;
   Head     : string[20];

   LoadFile : string[12];
   Wx,Wy    : integer;
   i        : byte;
Begin
{    Wx:=400; Wy:=50;}
     RetrieveBrandCode:=true;
     Head:='#'+BRAND;
     Assign(MNEfile,'BRANDS.MNE');
     {$I-} Reset(MNEfile); {$I+}
     If IOResult=2
        then begin
             Message('BRANDS File Not Found! (BRANDS.MNE)',a);
             RetrieveBrandCode:=false;
        end
        else begin
             Repeat Readln(MNEfile,MNE[1]); Until (MNE[1]=Head) or (Eof(MNEfile));
{show code   GWriteln(Wx,Wy,MNE[1]);}
             If not (Eof(MNEfile))
                then For i:=2 to 20 do
                     begin
                          Readln(MNEfile,MNE[i]);
{for debug                GWriteln(Wx,Wy,MNE[i]);}
                     end
                else begin
                          For i:=2 to 20 do MNE[i]:='';
                          RetrieveBrandCode:=false;
                end;
             Close(MNEfile);
        end;
End;
```

รูปที่ 5.16 : ฟังก์ชั่น RetrieveBrandCode

5.4 การแปลเป็นรหัสภายในของออมร่อน

เพื่อให้สามารถส่งโปรแกรมภาษานีมอนิคที่แปลแล้วไปให้เครื่อง PC ได้ จะต้องแปลโปรแกรมภาษานีมอนิคนั้นไปเป็นรหัสภายในของ PC ก่อน ซึ่งในงานวิจัยนี้ได้เลือกใช้ PC ของออมร่อนสำหรับทดลองก็เพราะ สามารถหา PC ของออมร่อนมาใช้งานได้และทางบริษัทออมร่อนได้มอบตารางสำหรับแปลภาษานีมอนิคไปเป็นรหัสภายในของภาษาเครื่องของออมร่อนให้

5.4.1 รหัสภายในของออมร่อน

ในขั้นต้นจึงศึกษารหัสภายในหรือภาษาเครื่องของออมร่อนให้เป็นที่เข้าใจเสียก่อน ตารางที่ 5.2 เป็นรหัสภายในของออมร่อนที่นำมาใช้งาน (ตัดตอนมาจากภาคผนวก ข)

No. of byte / Instruction	1	2	3	4	5	6	7	8
LD I,II	LSRA	ORA	EXT.					
	44	BA	OPERAND *1)		= 00(bit)00(CH-No)			
LD NOT I,II	LSRA	ORA	EXT					
	44	BA	OPERAND *2)		= 00(bit)01(CH-No)			
AND I,II	ANDA	EXT						
	B4	OPERAND *1)						
AND NOT I,II	ANDA	EXT						
	B4	OPERAND *2)						
OR I,II	ORA	EXT						
	BA	OPERAND *1)						
OR NOT I,II	ORA	EXT						
	BA	OPERAND *2)						
AND LD	LSLA	BCS	02	ANDA	%$7F			
	48	25	02	84	7F			
OR LD	LSLA	BCC	02	ORA	%$80			
	48	24	02	8A	80			
OUT I,II	LDB	EXT		STA	EXT			
	F6	OPERAND *1)		B7	OPERAND *1)			
OUT NOT I	COMA	LDB	EXT	STA	EXT		COMA	
	43	F6	OPERAND *1)	B7	OPERAND *1)		43	

ตารางที่ 5.2 : รหัสภายในของออมร่อนที่ใช้งาน

จะพิจารณาเห็นว่ารหัสสำหรับคำสั่งต่างๆจะมีขนาดความยาว (จำนวนไบต์) ไม่เท่ากันและแต่ละรหัสจะประกอบด้วยส่วนประกอบหลักตามคำสั่งนั้นๆ เช่น คำสั่งที่ไม่มีโอเปอร์แรนด์ (Operand) ก็จะแปลไปเป็นกลุ่มรหัสกลุ่มหนึ่ง ส่วนคำสั่งที่มีโอเปอร์แรนด์ก็จะประกอบด้วยรหัส 2 กลุ่ม กลุ่มแรกแปลจากชื่อคำสั่ง กลุ่มที่ 2 คำนวณจากค่าโอเปอร์แรนด์

จากตารางในช่องซ้ายสุดจะเป็นคำสั่งและประเภทของโอเปอร์แรนด์ที่จะใช้ได้ ตามข้อกำหนดระบุไว้ว่ามีโอเปอร์แรนด์อยู่ 5 ประเภท รหัสภายในของแต่ละคำสั่งแสดงไว้ทางขวา โดยแสดงให้เห็นว่าแต่ละไบต์ (byte) ประกอบด้วยข้อมูลอะไรบ้าง ตัวเลขที่แสดงเป็นเลขฐาน 16 และคำว่า OPERAND หมายความว่าจะต้องนำค่าโอเปอร์แรนด์ไปคำนวณตามข้อกำหนด ซึ่งแยกตามประเภทของโอเปอร์แรนด์ ให้ได้เป็นตัวเลขรหัสขนาด 2 ไบต์ นำมาบรรจุลงตรงตำแหน่งนั้น

5.4.2 <u>การคำนวณค่าโอเปอร์แรนด์</u>

ขอให้พิจารณาตารางการคำนวณค่าโอเปอร์แรนด์จากภาคผนวก ข ในตารางแสดงการคำนวณแยกตามประเภทของโอเปอร์แรนด์ พิจารณาโอเปอร์แรนด์ Type I จะมีขนาด 16 บิต บิตที่เขียนเลข 0 และ 1 จะต้องใช้ตามนั้น บิตที่มีเครื่องหมายสามเหลี่ยมและตัวเลขภายใน เป็นเลขฐาน 2 ซึ่งคำนวณมาจากเลข 2 ตัวท้ายของโอเปอร์แรนด์ที่ผู้ใช้ป้อนนำมาจัดตามตำแหน่งที่ระบุ โดยบิต MSB ให้ใช้นิเสธ และบิตที่มีอักษร x เป็นเลขฐาน 2 ซึ่งคำนวณมาจากเลข 2 ตัวหน้าของโอเปอร์แรนด์ที่ผู้ใช้ป้อน และบิตที่ 10 จะเป็น 1 ถ้าโอเปอร์แรนด์นี้ใช้กับคำสั่ง AND และบิตที่ 9 จะเป็น 1 ถ้ามีคำสั่ง NOT ปนอยู่ด้วย ตัวอย่างเช่น LD NOT 0001 จะคำนวณได้เป็น 0000 1010 0000 0001

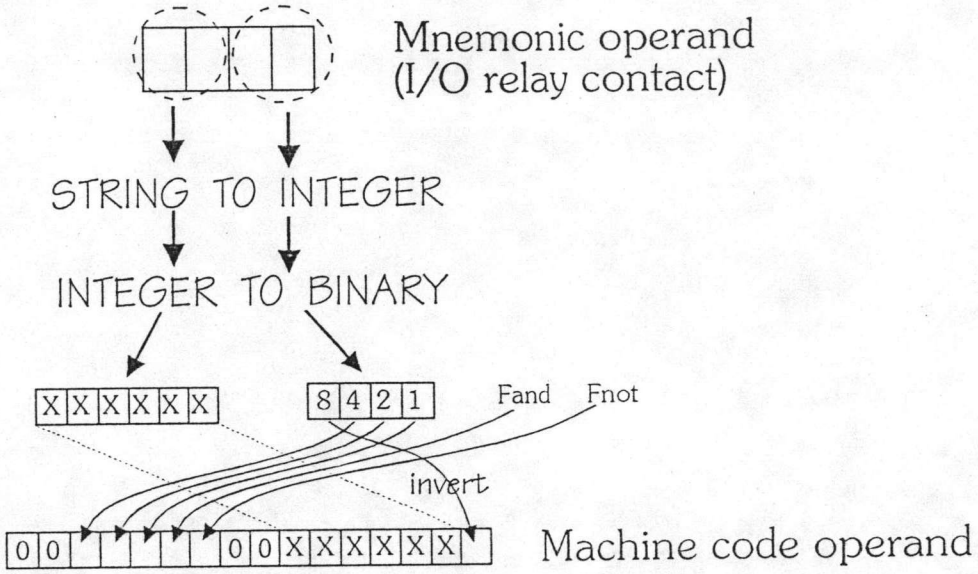
โอเปอร์แรนด์ Type II เป็นการนำตัวตั้งเวลาและตัวนับไปใช้งาน การคำนวณมีลักษณะคล้าย Type I โอเปอร์แรนด์ Type III มิได้ใช้ ส่วนโอเปอร์แรนด์Type IV ใช้สำหรับคำนวณค่าคงที่ของตัวตั้งเวลาและตัวนับ จากตารางที่ 2 ในรหัสภายในของตัวตั้งเวลาจะมีช่องหนึ่งเขียนว่า "BINARY" สามารถคำนวณได้จากหมายเลขของตัวตั้งเวลาที่ผู้ใช้กำหนด โดยเปลี่ยนให้เป็นเลขฐาน 2 จะขอยกตัวอย่างการแปลโปรแกรมนีมอนิค ไปเป็นรหัสภายในดังนี้

LD 0001	44 BA 0A 01
AND 0002	B4 14 01
AND NOT 3200	B4 04 41
OR LD	44 2A 02 8A 80
OUT 0201	F6 08 05 B7 08 05
END	7E C0 00

ในการคำนวณค่าโอเปอร์แรนด์ได้เขียนฟังก์ชั่น Operand เพื่อทำหน้าที่นี้ โดยจะรับตัวแปรและให้ค่าดังนี้

```
Function Operand (S : String; Var Fand, Fnot : boolean) : string;
```

โปรแกรมหลักจะวิเคราะห์ประโยคที่แปลก่อนว่ามีคำสั่ง AND หรือ NOT หรือไม่ และจะปรับค่าตัวแปร Fand และ Fnot ให้ถูกต้อง พร้อมกับส่งโอเปอร์แรนด์ที่ต้องการคำนวณมาให้ ฟังก์ชั่นโอเปอร์แรนด์จะคำนวณและให้ผลลัพธ์เป็นตัวอักษรรหัสขนาด 4 ตัวอักษร การทำงานของฟังก์ชั่นนี้แสดงดังแผนภาพในรูปที่ 5.17



รูปที่ 5.17 : การทำงานของฟังก์ชั่น Operand

5.4.3 <u>การแปล</u>

เนื่องจากรหัสภายในนี้มีความสัมพันธ์กับภาษานีมอนิคโดยตรงแบบหนึ่งต่อหนึ่ง จึงออกแบบให้การแปลเป็นรหัสภายในกระทำไปพร้อมๆกับการแปลเป็นภาษานีมอนิค โดยทุกๆบรรทัดของภาษานีมอนิคที่ถูกสร้างขึ้นก็จะทำให้เกิดการแปลเป็นรหัสภายในพร้อมๆกัน รหัสภายในที่แปลได้จะถูกลำเลียงไปเขียนลงแฟ้มข้อมูลเก็บไว้ โดยใช้ชื่อตามชาร์ต แต่เปลี่ยนนามสกุลเป็น "ITN" ซึ่งอาจกล่าวได้ว่าสามารถแปล CompiledTable ไปเป็นรหัสภายในได้โดยตรง

5.5 การแปลกลุ่มของชาร์ต

เนื่องจากชาร์ตที่ออกแบบมีขนาดจำกัด คือ 40 แถว x 17 คอลัมน์ ในบางครั้งโปรแกรมที่ผู้ใช้ต้องการเขียนมีขนาดใหญ่กว่า จึงได้ออกแบบให้ผู้ใช้สามารถแยกเขียนโปรแกรมเดียวกันเป็นหลายๆชาร์ตได้โดยไม่จำกัดจำนวนชาร์ต ดังนั้นผู้ใช้จึงสามารถเขียนชาร์ตที่มีขนาดใหญ่เท่าใดก็ได้ไม่จำกัด แต่ทั้งนี้จะต้องไม่ใหญ่เกินกว่าที่หน่วยความจำของ PC จะรับได้ ในการเขียนกลุ่มชาร์ตนี้มีข้อแม้อยู่ว่าจะต้องไม่ใช้เอาต์พุตรีเลย์ซ้ำกันระหว่างชาร์ต และการแปลกลุ่มชาร์ตผู้ใช้จะต้องทำตามขั้นตอนดังนี้

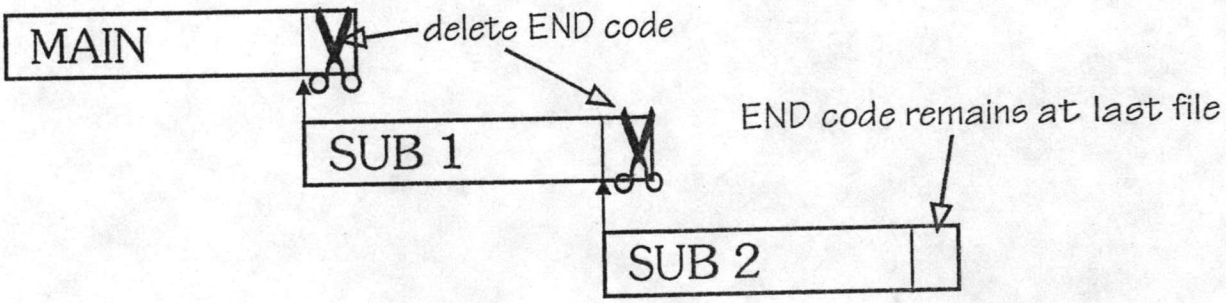
1. แปลชาร์ตแรก (ในกรณีที่เขียนชาร์ตเป็นลำดับ) หรือชาร์ตหลัก (ในกรณีที่ออกแบบ แบบ Top-down)

2. เลือกใช้คำสั่ง Merge ป้อนชื่อชาร์ตถัดไปและกด Enter

ในกรณีกลุ่มซาร์ตมีมากกว่า 2 ซาร์ต ก็ให้ทำซ้ำ 2 ซ้ำจนกว่าจะครบทุกซาร์ต ผลจากการแปลจะได้แฟ้มข้อมูลรหัสภายในร่วมของกลุ่มซาร์ต โดยใช้ชื่อตามซาร์ตแรก ซึ่งพร้อมที่จะส่งไป PC ได้ทันที วิธีเขียนการเชื่อมโยงลำดับการทำงานภายในกลุ่มของซาร์ต จะขอกล่าวถึงในบทที่ 9

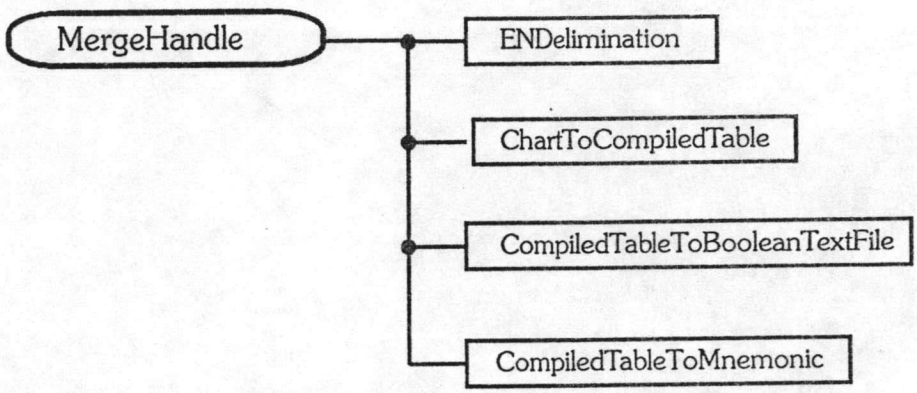
หลักการที่สำคัญในการแปลกลุ่มซาร์ต ก็คือ ในการกำหนดหมายเลขรีเลย์ช่วยให้กับสเต็ปนั้น (AssignAuxiliaryRelay) ในแต่ละซาร์ตจะต้องไม่ซ้ำกัน ก็จะทำให้สามารถนำโปรแกรมนีมอนิคหรือรหัสภายในของแต่ละซาร์ตมาเรียงต่อกันและนำไปใช้งานได้เลย จึงได้ออกแบบโดยใช้หลักการนี้กับรหัสภายใน แต่กับโปรแกรมนีมอนิคได้แยกเก็บไว้ตามเดิม เพราะมิได้เป็นส่วนที่ใช้ควบคุมการทำงานของ PC โดยตรงและเพื่อให้ผู้ใช้สามารถทำเอกสารแยกของแต่ละซาร์ตได้

ในการแปล ระบบจะแปลซาร์ตแรกก่อน จะได้แฟ้มรหัสภายในเกิดขึ้นมา เมื่อผู้ใช้ระบุว่า Merge ก็จะนำแฟ้มรหัสภายในของซาร์ตแรกขึ้นมาและตัดส่วนที่เป็นรหัสสิ้นสุดการทำงานทิ้งไป จากนั้นจะแปลรหัสภายในของซาร์ตที่สองต่อท้ายข้อมูลในแฟ้มรหัสภายในของซาร์ตแรก จนจบด้วยรหัสสิ้นสุดการทำงาน (เพราะกลุ่มซาร์ตนี้อาจมีเพียง 2 ซาร์ตก็ได้) ในการ Merge ซาร์ตที่ 3 ก็จัดการในลักษณะเดียวกัน



รูปที่ 5.18 : การรวมรหัสภายในของกลุ่มซาร์ต

ในการจัดการงานในส่วนนี้ ขั้นแรกจะต้องไปแก้โปรแกรม AssignAuxiliaryRelay ให้สามารถกำหนดหมายเลขรีเลย์ช่วยของซาร์ตที่นำมาแปลร่วมให้ต่อเนื่องกันได้ ซึ่งทำได้โดยสร้างแฟลก (Flag) ขึ้นมา 1 ตัว ซึ่งจะแอ็กทีฟเมื่อมีการแปลกลุ่มซาร์ต และให้โปรแกรมย่อย AssignAuxiliaryRelay ตรวจสอบแฟลกนี้ก่อนที่จะทำงานหลัก การจัดการในส่วนต่อไปเป็นการพัฒนาโปรแกรมย่อย MergeHandle ขึ้นมา ซึ่งมีการทำงานดังแสดงในรูปที่ 5.19

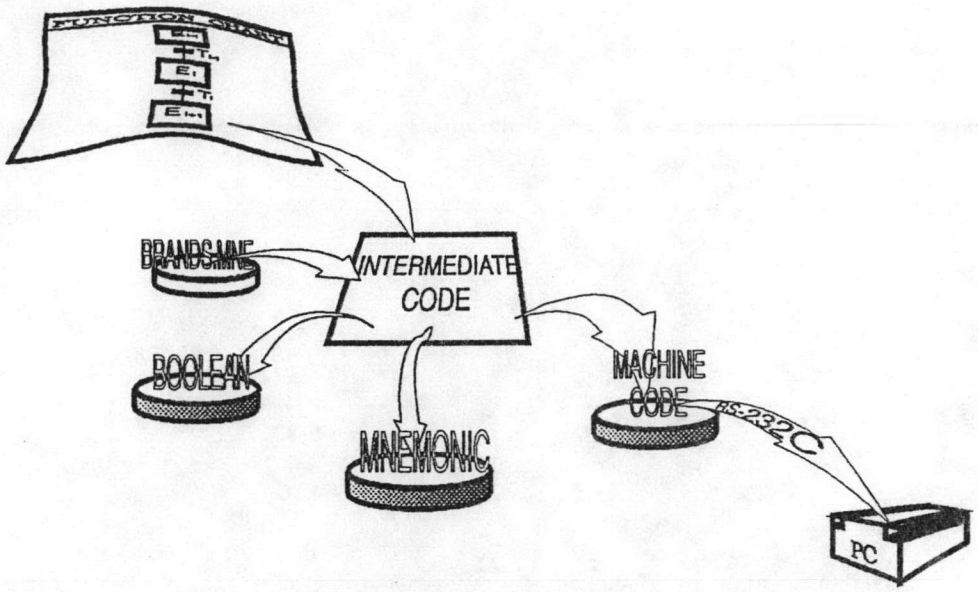


รูปที่ 5.19 : โปรแกรมย่อย MergeHandle

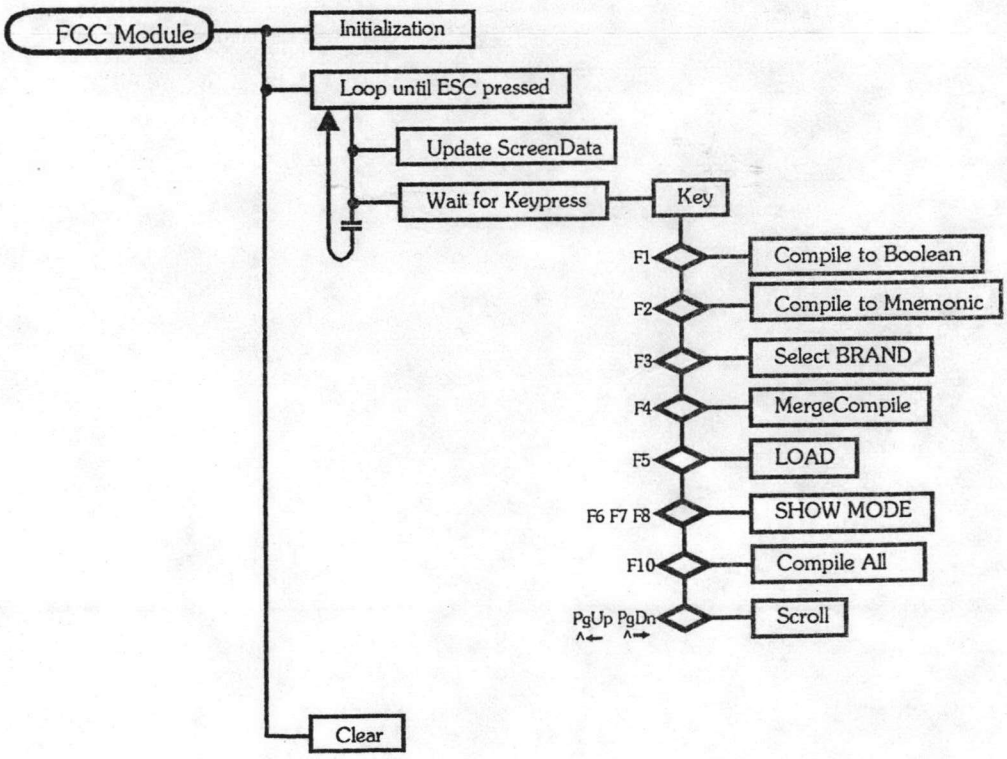
โปรแกรมย่อย ENDelimination จะตัดรหัสของการสิ้นสุดโปรแกรมออกจากแฟ้มรหัสภายในของชาร์ตแรก โปรแกรมย่อย ChartToCompiledTable จะแปลชาร์ตไปเป็น CompiledTable โปรแกรมย่อย CompiledTableToBooleanTextFile จะแปล CompiledTable ไปเป็นสมการบูลลีน และโปรแกรมย่อย CompiledTableToMnemonic จะแปล CompiledTable ไปเป็นภาษานีมอนิคและรหัสภายใน

5.6 สรุปการแปล

จากการออกแบบการแปลทั้งหมด จะเห็นได้ว่าโครงสร้าง CompiledTable เป็นส่วนสำคัญที่ช่วยลดเวลาและขั้นตอนการแปลในส่วนต่างๆลงอย่างมาก เพราะโครงสร้าง CompiledTable นี้ สามารถนำไปแปลเป็นสมการบูลลีนหรือภาษานีมอนิคของ PC ยี่ห้อต่างๆ หรือแปลไปเป็นรหัสภายในของออมร่อนได้ทันที ซึ่งสามารถสรุปเส้นทางของข้อมูลได้เป็นแผนภาพดังรูปที่ 5.20 โปรแกรมที่จัดการงานในโมดูลการแปลนี้(Function Chart Compiler, FCC) ได้แก่ โปรแกรม "FCOMPILE" ซึ่งมีขั้นตอนการทำงานดังแสดงด้วยแผนภาพในรูปที่ 5.21



รูปที่ 5.20 : สรุปการแปล



รูปที่ 5.21 : DSD ของโมดูล FCC