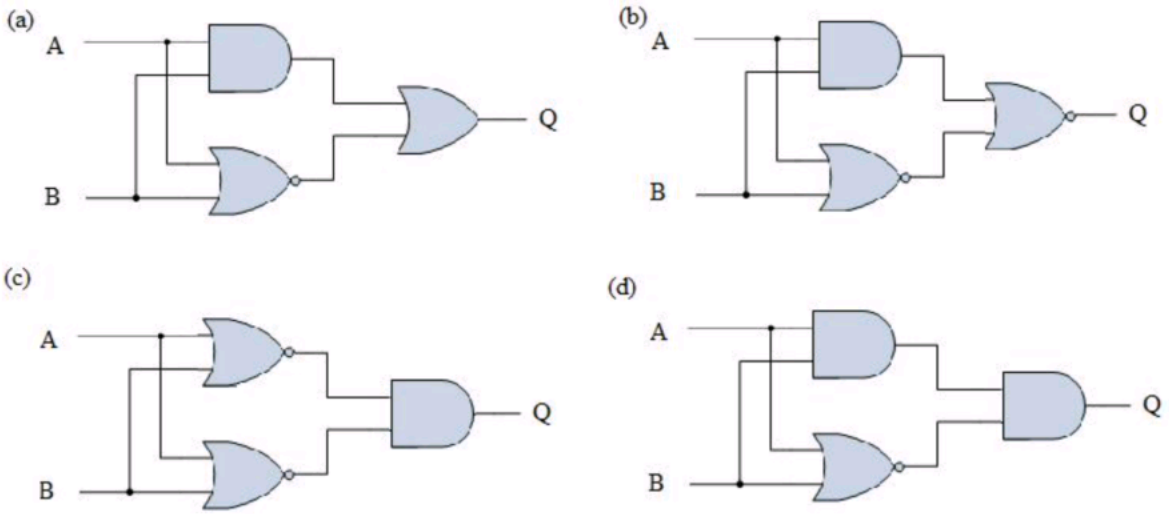
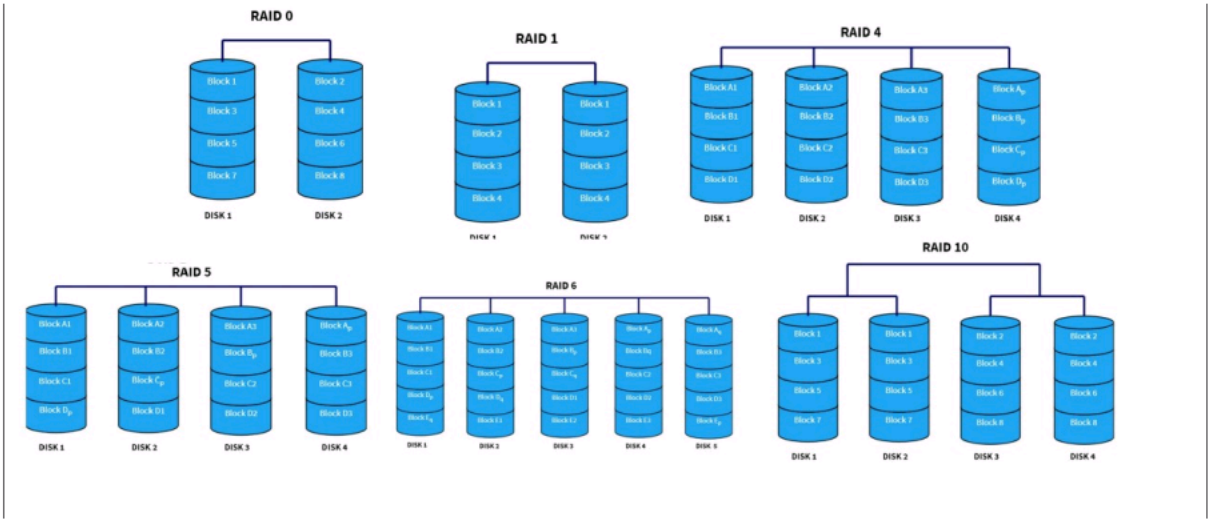
https://www.borntodev.com/2020/07/10/logic-gates-101/

The logic gate combination that represents the Boolean expression $(A.B).\overline{(A+B)} = Q$



RAID Levels (0, 1, 4, 5)

- **RAID 0:** ไม่มีการทำซ้ำข้อมูล ข้อดีคือความเร็วในการอ่าน/เขียนสูง แต่ไม่มีการป้องกันข้อมูล
- **RAID 1:** ทำสำเนาข้อมูลทั้งหมด ข้อดีคือการป้องกันข้อมูลที่ดี แต่มีการสูญเสียพื้นที่
- **RAID 4:** การเขียนข้อมูลจะมีการเขียนข้อมูล parity แยก ข้อดีคือสามารถกู้ข้อมูลได้
- **RAID 5:** การกระจายข้อมูลและ parity ทำให้เกิดการสำรองข้อมูลและเพิ่มความเร็ว



1. Partitioning

Partitioning เป็นเทคนิคการจัดการหน่วยความจำที่แบ่งพื้นที่หน่วยความจำออกเป็นส่วนย่อย ๆ (Partitions) เพื่อจัดสรรให้กับโปรแกรมหรือกระบวนการต่าง ๆ ที่รันบนระบบ

ข้อดีของ Partitioning:

- **การจัดสรรที่เรียบง่าย:** การจัดการหน่วยความจำในรูปแบบ Partitioning ทำได้ง่าย โดยแบ่งพื้นที่หน่วยความจำออกเป็นส่วน ๆ และจัดสรรให้กับโปรแกรมหรือกระบวนการที่ต้องการ
- **การแยกพื้นที่:** Partitioning ช่วยแยกพื้นที่ของโปรแกรมต่าง ๆ ทำให้ป้องกันการรบกวนกันระหว่างโปรแกรมหรือกระบวนการ
- **เหมาะสำหรับโปรแกรมขนาดเล็ก:** Partitioning เหมาะสำหรับระบบที่โปรแกรมมีขนาดเล็กและไม่ต้องการหน่วยความจำขนาดใหญ่

ข้อจำกัดของ Partitioning:

- **การใช้หน่วยความจำไม่คุ้มค่า:** Partitioning อาจทำให้เกิดปัญหาการใช้หน่วยความจำไม่คุ้มค่า (Internal Fragmentation) เนื่องจากพื้นที่ใน Partition อาจถูกใช้ไม่เต็มประสิทธิภาพ
- **การจัดสรรแบบคงที่ (Fixed Partitioning):** หากเป็นการจัดสรรแบบคงที่ (Fixed Partitioning) การปรับขนาด Partition ในภายหลังจะทำได้ยาก ทำให้ไม่ยืดหยุ่นต่อความต้องการของโปรแกรมหรือกระบวนการที่เปลี่ยนแปลงไป
- **ขนาดโปรแกรมจำกัด:** ขนาดของ Partition อาจจำกัดขนาดของโปรแกรมที่สามารถโหลดและรันได้

Figure 4: Fixed Partitioning

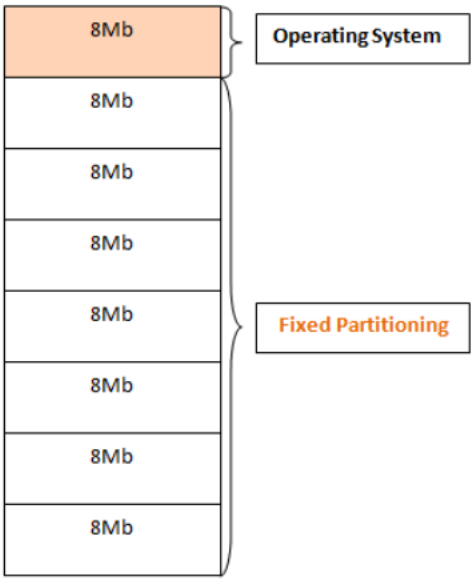
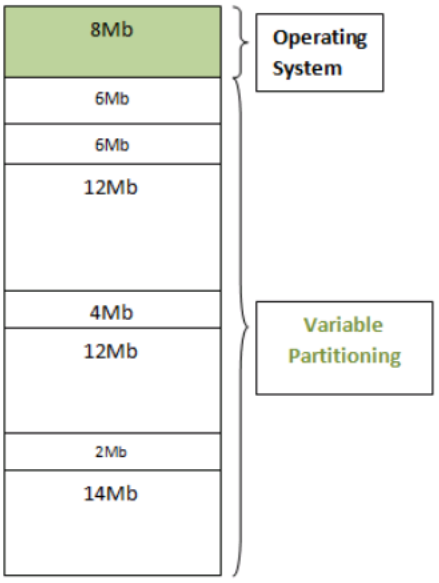


Figure 5: Variable Partitioning



2. Paging

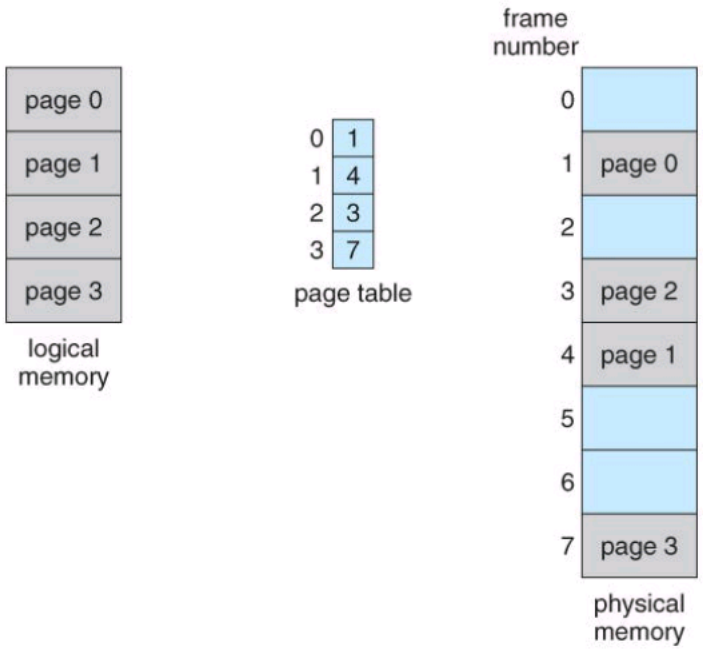
Paging เป็นเทคนิคการจัดการหน่วยความจำที่แบ่งโปรแกรมออกเป็นส่วนย่อย ๆ ที่เรียกว่า "Pages" และแบ่งหน่วยความจำจริงออกเป็นส่วนที่เรียกว่า "Frames" ซึ่ง Pages เหล่านี้จะถูกโหลดลงไปใน Frames ตามที่ต้องการ

ข้อดีของ Paging:

- **การใช้หน่วยความจำอย่างมีประสิทธิภาพ:** Paging ช่วยให้การจัดสรรหน่วยความจำเป็นไปอย่างมีประสิทธิภาพมากขึ้น เนื่องจากไม่จำเป็นต้องจัดสรรพื้นที่ที่ต่อเนื่องกัน ทำให้ลดปัญหาการสูญเสียพื้นที่หน่วยความจำ (External Fragmentation)
- **ความยืดหยุ่น:** Pages และ Frames มีขนาดเท่ากัน ทำให้การจัดสรรหน่วยความจำมีความยืดหยุ่นและสามารถจัดการได้ง่าย
- **การจัดการที่เป็นระบบ:** Paging ช่วยให้การจัดการหน่วยความจำเป็นระบบมากขึ้น เนื่องจากแต่ละ Page มีขนาดเท่ากันและถูกจัดเก็บใน Frame ที่สอดคล้องกัน

ข้อจำกัดของ Paging:

- **Overhead ในการจัดการ:** Paging อาจมี overhead ในการจัดการตาราง (Page Table) ที่เก็บข้อมูลการแมประหว่าง Pages และ Frames โดยเฉพาะอย่างยิ่งเมื่อโปรแกรมมีขนาดใหญ่
- **Internal Fragmentation:** แม้ว่าจะลดปัญหา External Fragmentation แต่ Paging ยังมีปัญหา Internal Fragmentation ซึ่งเกิดขึ้นเมื่อมีการใช้พื้นที่ใน Frame ไม่เต็ม



3. Demand Paging

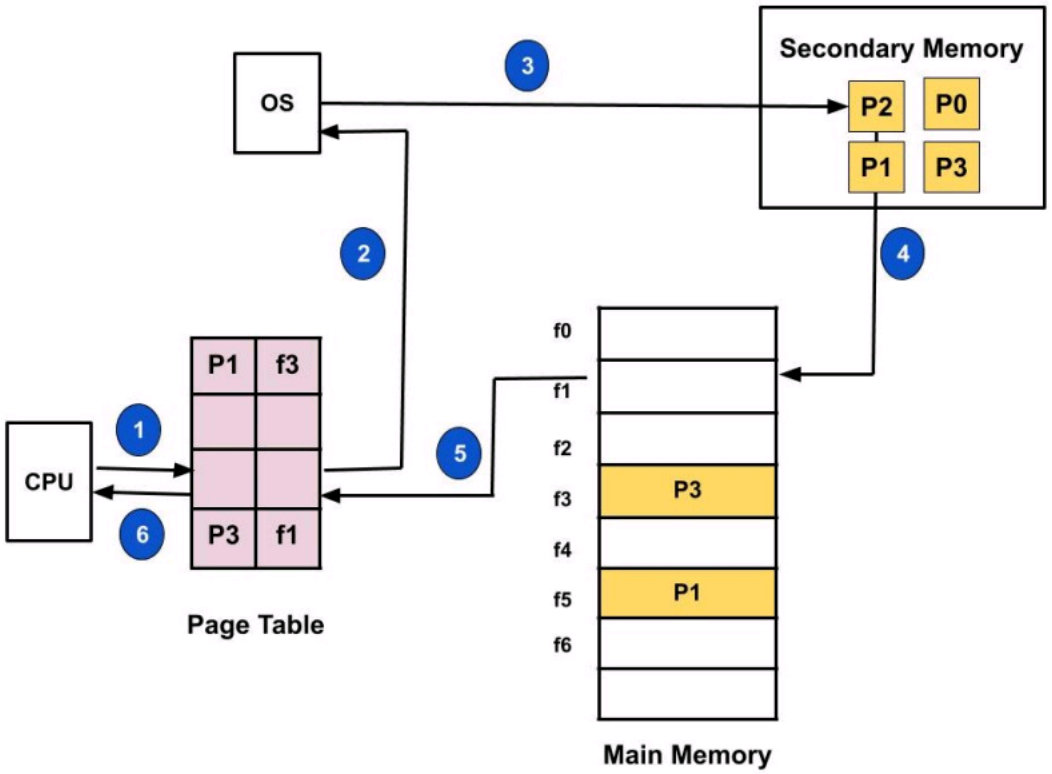
Demand Paging เป็นการผสมผสานระหว่างเทคนิค Paging กับแนวคิดของการ Swapping โดย Pages จะถูกโหลดเข้าสู่หน่วยความจำจริงเมื่อโปรแกรมต้องการใช้เท่านั้น ทำให้สามารถจัดการหน่วยความจำได้อย่างมีประสิทธิภาพมากขึ้น

ข้อดีของ Demand Paging:

- **ประหยัดหน่วยความจำ:** Demand Paging ช่วยประหยัดพื้นที่หน่วยความจำจริง เนื่องจากโปรแกรมจะไม่ถูกโหลดลงทั้งหมด แต่จะโหลดเฉพาะ Pages ที่ต้องการใช้งานจริง ๆ
- **การใช้หน่วยความจำอย่างมีประสิทธิภาพ:** Demand Paging ช่วยลดการใช้พื้นที่หน่วยความจำที่ไม่จำเป็นและทำให้สามารถจัดการกับโปรแกรมขนาดใหญ่ได้แม้ว่าหน่วยความจำจริงจะมีขนาดเล็ก
- **การจัดการที่ยืดหยุ่น:** Demand Paging ช่วยให้ระบบสามารถจัดการหน่วยความจำได้อย่างยืดหยุ่น และลดปัญหาการใช้หน่วยความจำไม่คุ้มค่า (Internal Fragmentation)

ข้อจำกัดของ Demand Paging:

- **Page Fault:** Demand Paging อาจเกิดปัญหา Page Fault เมื่อโปรแกรมต้องการใช้ Page ที่ยังไม่ได้โหลดลงในหน่วยความจำ ทำให้ต้องมีการเข้าถึงหน่วยความจำเสมือน (Secondary Storage) ซึ่งทำให้ระบบทำงานช้าลง
- **Overhead ของระบบ:** Demand Paging อาจทำให้ระบบมี Overhead ในการจัดการหน่วยความจำเพิ่มขึ้น เนื่องจากต้องมีการตรวจสอบและโหลด Pages อย่างต่อเนื่อง
- **ความซับซ้อน:** การจัดการ Demand Paging มีความซับซ้อนมากกว่าการใช้ Paging แบบธรรมดา เนื่องจากต้องมีการตรวจสอบและจัดการการโหลด Pages เพิ่มขึ้น



สรุป

- **Partitioning:** เหมาะสำหรับระบบที่โปรแกรมมีขนาดเล็กและไม่ซับซ้อน แต่มีข้อจำกัดในเรื่องของการจัดสรรพื้นที่และการใช้หน่วยความจำที่ไม่คุ้มค่า
- **Paging:** ช่วยให้การจัดการหน่วยความจำเป็นไปอย่างมีประสิทธิภาพและยืดหยุ่นมากขึ้น แต่ยังมีปัญหา Internal Fragmentation และ Overhead ในการจัดการ Page Table
- **Demand Paging:** เพิ่มความยืดหยุ่นและประหยัดหน่วยความจำ แต่มีความซับซ้อนและอาจเกิดปัญหา Page Fault ทำให้ระบบทำงานช้าลง

1. ข้อมูลพื้นฐาน

- **ขนาดของ Cache Memory:** 512 bytes
- **ขนาดของ Block (Cache Line Size):** 8 bytes
- **จำนวนของ Ways:** 4 (4-way set-associative)
- **ขนาดของ Address:** 12 bits

2. การคำนวณขนาดของ Block, Set, และ Offset

ขนาดของ Offset (หรือ Word)

Offset คือขนาดของข้อมูลภายในแต่ละ block ซึ่งจะบ่งบอกตำแหน่งที่แน่นอนใน block ขนาด 8 bytes:

$$\text{ขนาดของ Offset} = \log_2(\text{ขนาดของ Block}) = \log_2(8) = 3 \text{ bits}$$

จำนวนของ Sets

Cache มีขนาด 512 bytes และแต่ละ block ขนาด 8 bytes ทำให้จำนวน blocks เป็น:

$$\text{จำนวนของ Blocks} = \frac{\text{ขนาดของ Cache Memory}}{\text{ขนาดของ Block}} = \frac{512 \text{ bytes}}{8 \text{ bytes/block}} = 64 \text{ blocks}$$

เนื่องจาก cache เป็น 4-way set-associative, ดังนั้นจำนวน sets คือ:

$$\text{จำนวนของ Sets} = \frac{\text{จำนวนของ Blocks}}{\text{จำนวนของ Ways}} = \frac{64}{4} = 16 \text{ sets}$$

ขนาดของ index (หรือ Set) จะเป็น:

$$\text{ขนาดของ Index} = \log_2(\text{จำนวนของ Sets}) = \log_2(16) = 4 \text{ bits}$$

ขนาดของ Tag

ขนาดของ tag สามารถคำนวณจากขนาดของ address, ขนาดของ index และ offset:

$$\text{ขนาดของ Tag} = \text{ขนาดของ Address} - (\text{ขนาดของ Index} + \text{ขนาดของ Offset})$$

$$\text{ขนาดของ Tag} = 12 - (4 + 3) = 5 \text{ bits}$$

สรุป

- **ขนาดของ Tag:** 5 bits
- **ขนาดของ Index (Set):** 4 bits
- **ขนาดของ Offset (Word):** 3 bits

นี่คือจำนวน bit ที่ใช้ในแต่ละส่วนของ address สำหรับ cache memory แบบ 4-way set-associative ที่ระบุ.

ภาพรวม

sql

```
+------------------+------------------+------------------+
|       Tag        |       Index      |      Offset      |
|    (5 bits)      |      (4 bits)    |     (3 bits)     |
+------------------+------------------+------------------+

Cache Memory:
+-------------------------------------------------+
|                 Set 0 (4 blocks)                |
+-------------------------------------------------+
|  Block 0 | Block 1 | Block 2 | Block 3          |
|  (8 bytes) (8 bytes) (8 bytes) (8 bytes)        |
+-------------------------------------------------+
|                 Set 1 (4 blocks)                |
|  ...                                            |
+-------------------------------------------------+
|                 Set 15 (4 blocks)               |
+-------------------------------------------------+
```

รายละเอียดของแต่ละ Block

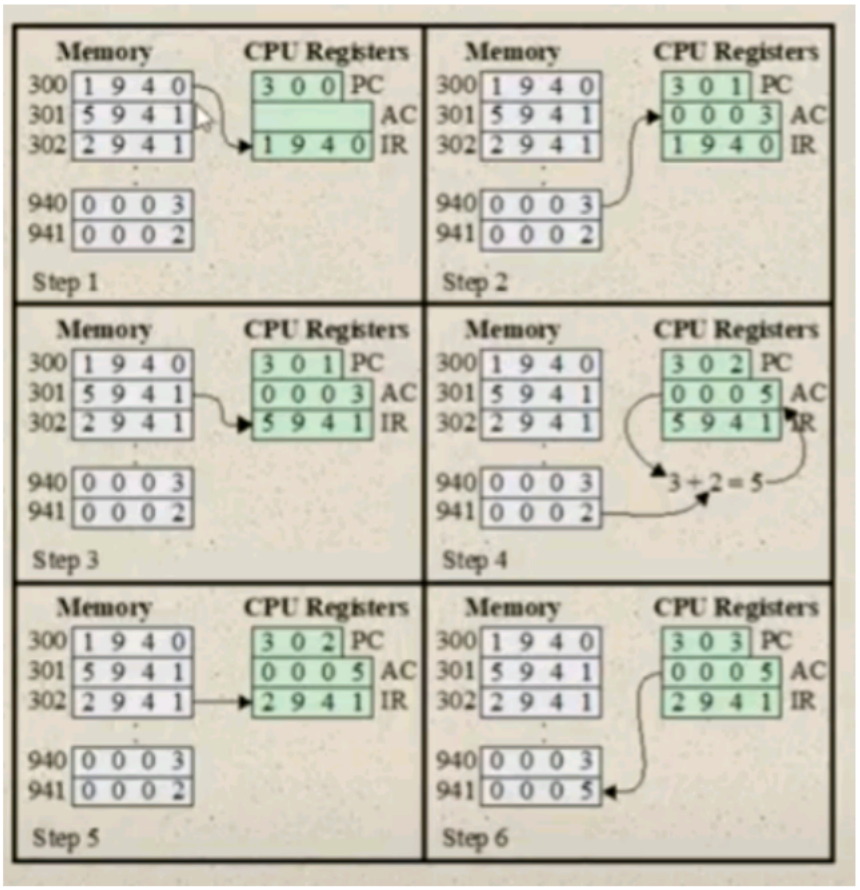
mathematica คัดลอกโค้ด

```
+-----------------------------------------------------------+
|                      Block (8 bytes)                      |
+-----------------------------------------------------------+
| Byte 0 | Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Byte 7 |
+-----------------------------------------------------------+
```

Memory Block ที่มีโอกาศอยู่ใน Set 3 คือ Block 12,13,14,15 (Set ละ 4 Block)

Addressing Mode

- **Immediate Addressing:** ใช้ข้อมูลในคำสั่งโดยตรง
- **Direct Addressing:** ใช้ address ที่อยู่ในคำสั่ง
- **Indirect Addressing:** ใช้ address ที่ถูกชี้โดย address ที่อยู่ในคำสั่ง
- **Register Addressing:** ใช้ข้อมูลที่อยู่ใน register
- **Register Indirect Addressing:** ใช้ address ที่ถูกชี้โดย register



อธิบายข้อดีและข้อจำกัดของการใช้การอ้างอิงที่อยู่ของตัวถูกดำเนินการทั้งสองรูปแบบ (5 คะแนน)

Three-address instructions CPU with 3 Register	
Instruction	Operation
ADD A, B, C	A <- B + C
SUB A, B, C	A <- B - C
MPY A, B, C	A <- B * C
DIV A, B, C	A <- B / C

Two-address instructions CPU with 2 Register	
Instruction	Operation
ADD A, B	A <- A + B
SUB A, B	A <- A – B
MPY A, B	A <- A * B
DIV A, B	A <- A / B
MOV A, B	A <- B

1. Load Immediate (โหลดค่าคงที่เข้าไปใน register)

```
LDI AC, 100          ; โหลดค่าคงที่ 100 ไปยัง register AC
```

2. Load Direct (โหลดค่าจากที่อยู่ตรงไปยัง register)

```
LDD AC, 100          ; โหลดค่าจากตำแหน่งหน่วยความจำ 100 ไปยัง regist
er AC
```

3. Store Direct (เก็บค่าจาก register ไปยังที่อยู่ตรง)

```
STD AC, 500          ; เก็บค่าจาก register AC ไปที่ตำแหน่งหน่วยความจำ 500
```

4. Add Register Indirect (บวกค่าจากที่อยู่ที่ระบุโดย register ไปยัง register)

```
LDD AC, 400          ; โหลดค่าจากตำแหน่งหน่วยความจำ 400 ไปยัง register AC
ADD Base, AC         ; บวกค่าจาก register AC กับค่าจาก register Base
STD AC, 500          ; เก็บค่าผลลัพธ์ที่ register AC ไปที่ตำแหน่งหน่วยความจำ 500
```

5. Subtract Immediate (ลบค่าคงที่ออกจาก register)

```
LDI AC, 50           ; โหลดค่าคงที่ 50 ไปยัง register AC
SUBI AC, 10          ; ลบค่าคงที่ 10 จาก register AC
```

6. Move Register (ย้ายค่าจาก register หนึ่งไปยังอีก register หนึ่ง)

```
MOV Base, AC         ; ย้ายค่าจาก register AC ไปยัง register Base
```

7. Branch if Zero (กระโดดไปยังที่อยู่ที่ระบุหากค่าใน register เป็นศูนย์)

```
BEQ AC, 200          ; กระโดดไปที่ตำแหน่ง 200 ถ้าค่าใน register AC เป็นศูนย์
```

8. **Load Displacement** (โหลดค่าจากตำแหน่งที่อยู่ที่ระบุด้วยการเพิ่ม offset)

```
LDD AC, 100          ; โหลดค่าจากตำแหน่งหน่วยความจำ 100 ไปยัง register AC
ADD AC, 400          ; บวกค่า 400 เข้าไปใน register AC
STD AC, 500          ; เก็บผลลัพธ์ที่ register AC ไปที่ตำแหน่งหน่วยความจำ 500
```

คำสั่งเหล่านี้แสดงถึงการใช้ Addressing Modes ต่างๆ และการบวกค่าที่ตำแหน่ง 100 และ 400 รวมถึงการเก็บผลลัพธ์ไปยังตำแหน่ง 500 อย่างครบถ้วน

Operation	Opcode (Binary)
AC <- $(0xXXXh)_{mem}$	0001
AC -> $(0xXXXh)_{mem}$	0010
AC <- $(0xXXXh)_{I/O}$	0101
AC -> $(0xXXXh)_{I/O}$	0110
AC <- AC + $(0xXXXh)_{mem}$	1001

ขนาด 12-bit ในฐาน 2:

1. **จำนวนค่าที่สามารถเก็บได้:**
   - จำนวนทั้งหมดที่สามารถเก็บได้ใน 12-bit คือ $2^{12} = 4096$ ค่าที่เป็นไปได้
2. **ช่วงของข้อมูล:**
   - ค่าต่ำสุด: 0000 0000 0000 (ในฐาน 2) ซึ่งเป็น 0 (ในฐาน 10)
   - ค่าสูงสุด: 1111 1111 1111 (ในฐาน 2) ซึ่งเป็น 4095 (ในฐาน 10)

ตัวอย่าง:

- **Binary 0 ถึง 4095:**
  - Decimal 0 = Binary 0000 0000 0000
  - Decimal 1 = Binary 0000 0000 0001
  - Decimal 255 = Binary 0000 1111 1111
  - Decimal 4095 = Binary 1111 1111 1111

1. Immediate Addressing Mode

คำสั่ง:

assembly
Copy code
```
LDI AC, #25       ; โหลดค่าคงที่ 25 ลงใน register AC
```

อธิบายการทำงาน:

- **Immediate Addressing**: คำสั่งนี้ใช้การระบุค่าคงที่ (Immediate Value) ตรงๆ ในคำสั่งเอง
- **LDI AC, #25**: คำสั่งนี้จะโหลดค่าคงที่ 25 เข้าไปใน register AC โดยตรง โดยไม่ต้องเข้าถึงหน่วยความจำ

2. Direct Addressing Mode

คำสั่ง:

assembly
Copy code
```
LDD AC, 100       ; โหลดค่าจากตำแหน่งหน่วยความจำ 100 ลงใน register AC
```

อธิบายการทำงาน:

- **Direct Addressing**: คำสั่งนี้ใช้ที่อยู่ของหน่วยความจำโดยตรงในการเข้าถึงข้อมูล
- **LDD AC, 100**: คำสั่งนี้จะโหลดค่าจากที่อยู่หน่วยความจำ 100 ลงใน register AC

3. Indirect Addressing Mode

คำสั่ง:

assembly
Copy code
```
LDD AC, (Base)    ; โหลดค่าจากที่อยู่ที่ระบุใน register Base ลงใน register AC
```

อธิบายการทำงาน:

- **Indirect Addressing**: คำสั่งนี้ใช้ register ที่เก็บที่อยู่ของข้อมูลจริง ซึ่งข้อมูลอยู่ที่ตำแหน่งที่ระบุโดย register
- **LDD AC, (Base)**: คำสั่งนี้จะโหลดค่าจากตำแหน่งหน่วยความจำที่ระบุใน register Base ลงใน register AC

4. Register Addressing Mode

คำสั่ง:

assembly
Copy code
```
MOV AC, Base      ; คัดลอกค่าจาก register Base ไปยัง register AC
```

อธิบายการทำงาน:

- **Register Addressing**: คำสั่งนี้ทำการเข้าถึงข้อมูลจาก register หนึ่งไปยังอีก register หนึ่ง
- **MOV AC, Base**: คำสั่งนี้จะคัดลอกค่าจาก register Base ไปยัง register AC

5. Register Indirect Addressing Mode

คำสั่ง:

assembly
Copy code
```
ADD AC, (Base)    ; บวกค่าจากตำแหน่งที่ระบุใน register Base ไปยัง register AC
```

อธิบายการทำงาน:

- **Register Indirect Addressing**: คำสั่งนี้ใช้ register ที่เก็บที่อยู่ของข้อมูลจริง และเข้าถึงข้อมูลจากตำแหน่งที่ระบุ
- **ADD AC, (Base)**: คำสั่งนี้จะบวกค่าจากตำแหน่งหน่วยความจำที่ระบุใน register Base ไปยัง register AC

สรุป:

- **Immediate Addressing**: ใช้ค่าคงที่ที่ระบุในคำสั่งเอง
- **Direct Addressing**: ใช้ที่อยู่ของหน่วยความจำโดยตรง
- **Indirect Addressing**: ใช้ที่อยู่ที่เก็บใน register เพื่อเข้าถึงหน่วยความจำ
- **Register Addressing**: ใช้ register ในการเก็บข้อมูล
- **Register Indirect Addressing**: ใช้ register ที่เก็บที่อยู่ในการเข้าถึงข้อมูล

การใช้ addressing modes ต่าง ๆ ช่วยให้สามารถจัดการกับข้อมูลในหลายรูปแบบและทำให้การเขียนโปรแกรม assembly มีความยืดหยุ่นมากขึ้น.