01076114
องค์ประกอบและสถาปัตยกรรมคอมพิวเตอร์
Computer Organization and Architecture

Function Call, Addressing Mode

Linker and Loader

Function Call

- Function หรือ Procedure เป็นวิธีการเขียนโปรแกรมที่สำคัญ เนื่องจากทำให้โปรแกรมมีโครงสร้างที่ทำให้เข้าใจได้ง่ายขึ้น
- ในการเรียก Function จะมีขั้นตอนดังนี้
  1. นำพารามิเตอร์ที่จะส่งไปยังฟังก์ชัน ไปเก็บในตำแหน่งที่ฟังก์ชันสามารถเข้าถึง
  2. ส่งการควบคุมสู่ฟังก์ชัน
  3. จองหน่วยความจำเพิ่มเติมสำหรับทำงานในฟังก์ชัน
  4. ทำงานตามโปรแกรมที่เขียนในฟังก์ชัน
  5. นำผลลัพธ์เก็บในตำแหน่งที่โปรแกรมที่เรียกมาสามารถเข้าถึงได้
  6. ส่งการควบคุมกลับไปที่โปรแกรมที่เรียกมา

Branch and Link

1. Parameter หากมีจำนวนไม่มาก ได้กำหนดเป็นแนวปฏิบัติ คือ ให้ส่งผ่าน register (เพื่อให้ทำงานได้เร็ว) กำหนดให้ใช้ r0-r3 ในการส่งผ่าน
2. ในการส่งการทำงานไปยัง function จะใช้คำสั่ง BL (branch and link) โดยเรียกใช้ดังนี้

BL ProcedureAddress

โดยคำสั่งจะทำงานคล้ายคำสั่ง B คือ จะสั่งให้กระโดดไปทำงานในตำแหน่งที่ต้องการ แต่ BL จะ copy รีจิสเตอร์ pc (Program Counter) หรือ r15 ซึ่งเป็นรีจิสเตอร์พิเศษที่ทำหน้าที่ชี้ตำแหน่งของคำสั่งที่จะทำงาน ไปไว้ที่รีจิสเตอร์ lr (Link Register) เพื่อใช้เป็น address ในการ return กลับจากฟังก์ชัน (เนื่องจาก ARM จะทำ pc=pc+4 ตอนที่เริ่มทำคำสั่ง ดังนั้น address ที่เก็บใน lr จะเป็น address ถัดไปจากคำสั่ง BL)

Branch and Link

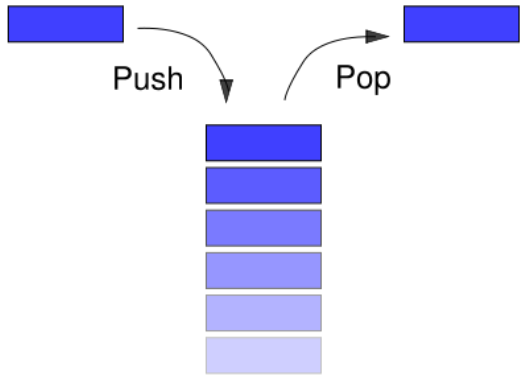
5. ในการกลับจากฟังก์ชันจะใช้คำสั่ง

   MOV pc, lr

6. ในการส่งค่ากลับ จะส่งทางรีจิสเตอร์ (เพื่อความรวดเร็วในการทำงานเช่นกัน) โดยจะใช้รีจิสเตอร์ r0 และ r1

Stack

- จากที่กำหนดให้การเรียก function จะใช้ register ได้เพียง r0-r3 (ไม่นับ lr และ pc ที่ใช้โดยคำสั่ง BL) เท่านั้นแต่ในความเป็นจริง การทำงานอาจจะต้องใช้ register มากกว่านั้น
- แต่จะนำ register อื่นๆ ไปใช้เลย ก็ทำไม่ได้ เนื่องจากไม่ทราบว่า ก่อนหน้านี้ (ในโปรแกรมที่เรียกมา) มีการใช้ register อะไรอยู่บ้าง
- ดังนั้นหากจะต้องมีการใช้ register ก็จะต้องรักษาข้อมูลใน register เหล่านั้นเอาไว้ ซึ่งโดยทั่วไป มักใช้ stack ในการเก็บข้อมูล



Example

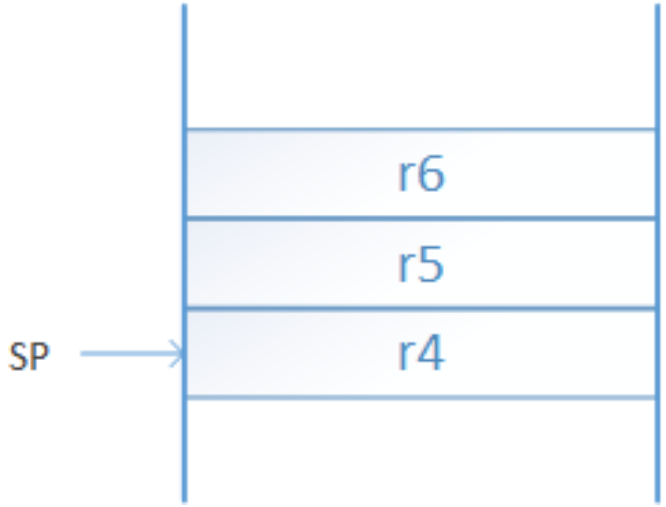
- จากโปรแกรมต่อไปนี้

```
int  leaf_example (int g, int h, int i, int j)
{
    int f ;
    f = (g + h) – (i + j);
    return f;
}
```

- มี parameter จำนวน 4 ตัว ดังนั้นกำหนดให้ r0=g, r1=h, r2=I, r3=j
- สำหรับ f จะใช้ r4
- ในการคำนวณ เราจะต้องใช้ register เพิ่มอีก 2 ตัว โดยจะใช้ r5 และ r6 ดังนั้น จะต้อง save ข้อมูล r4, r5, r6 ลง stack เสียก่อน

Example

```
leaf_example:
        SUB     sp, sp, #12     ; adjust stack to make room for 3 items
        STR     r6, [sp, #8]    ; save register r6 to stack
        STR     r5, [sp, #4]    ; save register r5 to stack
        STR     r4, [sp, #0]    ; save register r4 to stack
```



Example

จากนั้นเป็นส่วนประมวลผล

f = (g + h) – (i + j);

```
ADD     r5, r0, r1      ; register r5 contains g + h
ADD     r6, r2, r3      ; register r6 contains i + j
SUB     r4, r5, r6      ; f gets r5 – r6
```

จากนั้นก็เป็นการส่งคืนผลลัพธ์ผ่าน r0

```
MOV     r0, r4          ; return f (r0 = r4)
```

สุดท้ายจะเป็นการคืน stack และ return

```
LDR     r4, [sp, #0]    ; restore register r4
LDR     r5, [sp, #4]    ; restore register r5
LDR     r6, [sp, #8]    ; restore register r6
ADD     sp, sp, #12     ; adjust stack
MOV     pc, lr          ; jump back to calling routine
```

Example

```
int  leaf_example (int g, int h, int i, int j)
{
    int f ;
    f = (g + h) – (i + j);
    return f;
}
```

```
leaf_example:
        SUB     sp, sp, #12
        STR     r6, [sp, #8]
        STR     r5, [sp, #4]
        STR     r4, [sp, #0]
        ADD     r5, r0, r1
        ADD     r6, r2, r3
        SUB     r4, r5, r6
        MOV     r0, r4
        LDR     r4, [sp, #0]
        LDR     r5, [sp, #4]
        LDR     r6, [sp, #8]
        ADD     sp, sp, #12
        MOV     pc, lr
```

Storage Management on a Call/Return

1. Function มีหน้าที่สร้างพื้นที่สำหรับเก็บตัวแปรของ Function ใน Stack
2. ก่อนและหลังทำคำสั่ง BL ทั้งผู้เรียกและผู้ถูกเรียก จะต้องไม่ใช้ register r0-r3 และ r12 เนื่องจากต้องใช้ในการส่งผ่าน argument
3. Function มีหน้าที่ต้องเก็บรักษา register อื่นๆ ที่ไม่ใช่ r0-r3 และ r12
4. เมื่อ Function ใช้งาน Stack จะต้อง Update รีจิสเตอร์ sp เสมอ
5. ในการส่งคืนผลลัพธ์จะต้องส่งคืนทาง register r0 และ r1
6. ก่อนจะส่งกลับการทำงานจะต้องคืนค่าใน register ที่มีการใช้งาน, free stack

Preserved	Not preserved
Variable register: r4-r11	Argument register: r0-r3
Stack pointer register: sp	Intra-procedured-call register: r12
Link register : lr	Stack below the stack pointer
Stack above the stack pointer	

Nested Function

- ในการเขียนโปรแกรมนั้น ไม่สามารถกำหนดได้ว่า จะต้องเรียกใช้ function เพียงชั้นเดียว นอกจากฟังก์ชันที่อยู่ปลายสุด (leaf function) คือ ไม่มีการเรียกฟังก์ชันอื่นๆ ต่ออีก
- ในกรณีที่มีการเรียกฟังก์ชันต่อกันมากกว่า 1 ชั้น การเก็บ pc เอาไว้ที่ lr จะไม่พอเพียง เช่น ในกรณี

```
Proc  A

        call  Proc B
              …
              call Proc C
                 …
              return
        return
return
```

Example

```
int   fact  (int n)
{
      if (n < 1)  return (1);
            else return (n * fact(n-1));
}
```

- กำหนดให้ n = r0
- เนื่องจากเป็น recursive function ดังนั้นทุกรอบจะต้อง save ทั้ง lr และ n

```
fact:
        SUB     sp, sp, #8          ; adjust stack for 2 times
        STR     lr, [sp, #4]        ; save the return address
        STR     r0, [sp, #0]        ; save the argument n
```

Example

- ตอนแรกจะต้องตรวจสอบก่อนว่าเป็น n<1 หรือไม่

```
        CMP     r0, #1          ; compare n to 1
        BGE     L1              ; if n >=1 goto L1 (continue recursive)
```

- ถ้า <1 จะส่งค่า 1 กลับ

```
        MOV     r0, #1          ; return 1
        ADD     sp, sp, #8      ; pop 2 items off stack
        MOV     pc, lr
```

- วนเรียก Function ต่อไป

```
L1:     SUB     r0, r0, #1      ; n >= 1: argument gets (n-1)
        BL      fact            ; call fact with (n-1)
```

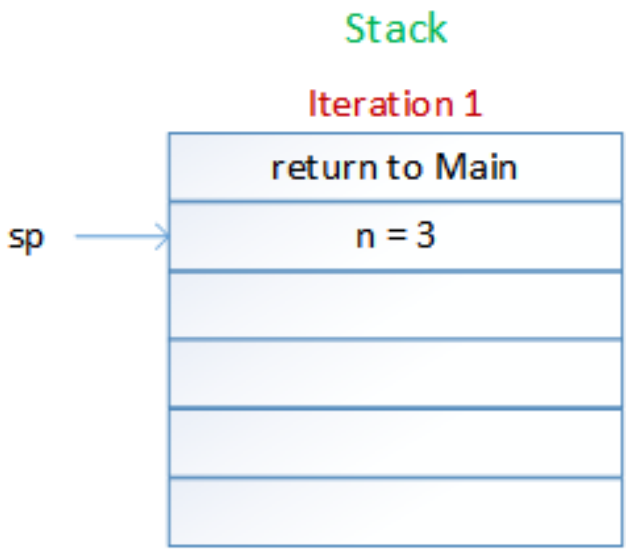
```
if (n < 1)  return (1);
        else return (n * fact(n-1));
```

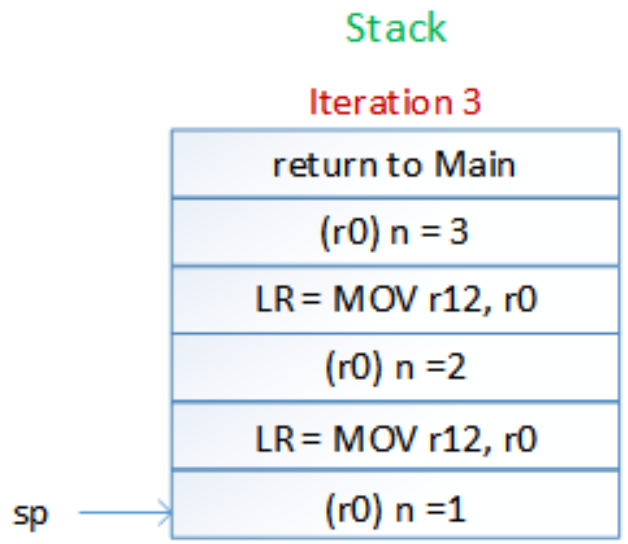
Example

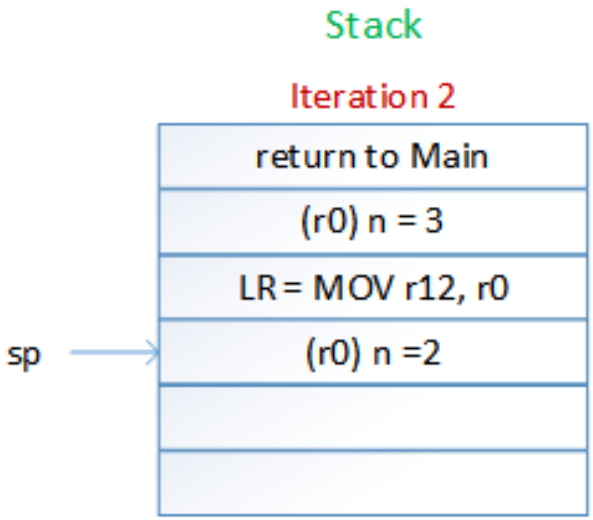
- คืนค่าและ return

```
MOV     r12, r0          ; save the return value
LDR     r0, [sp, #0]     ; return from BL: restore argument n
LDR     lr, [sp, #4]     ; restore return address
ADD     sp, sp, #8       ; adjust stack pointer
MUL     r0, r0, r12      ; return n * fact (n-1)
MOV     pc, lr
```

Example



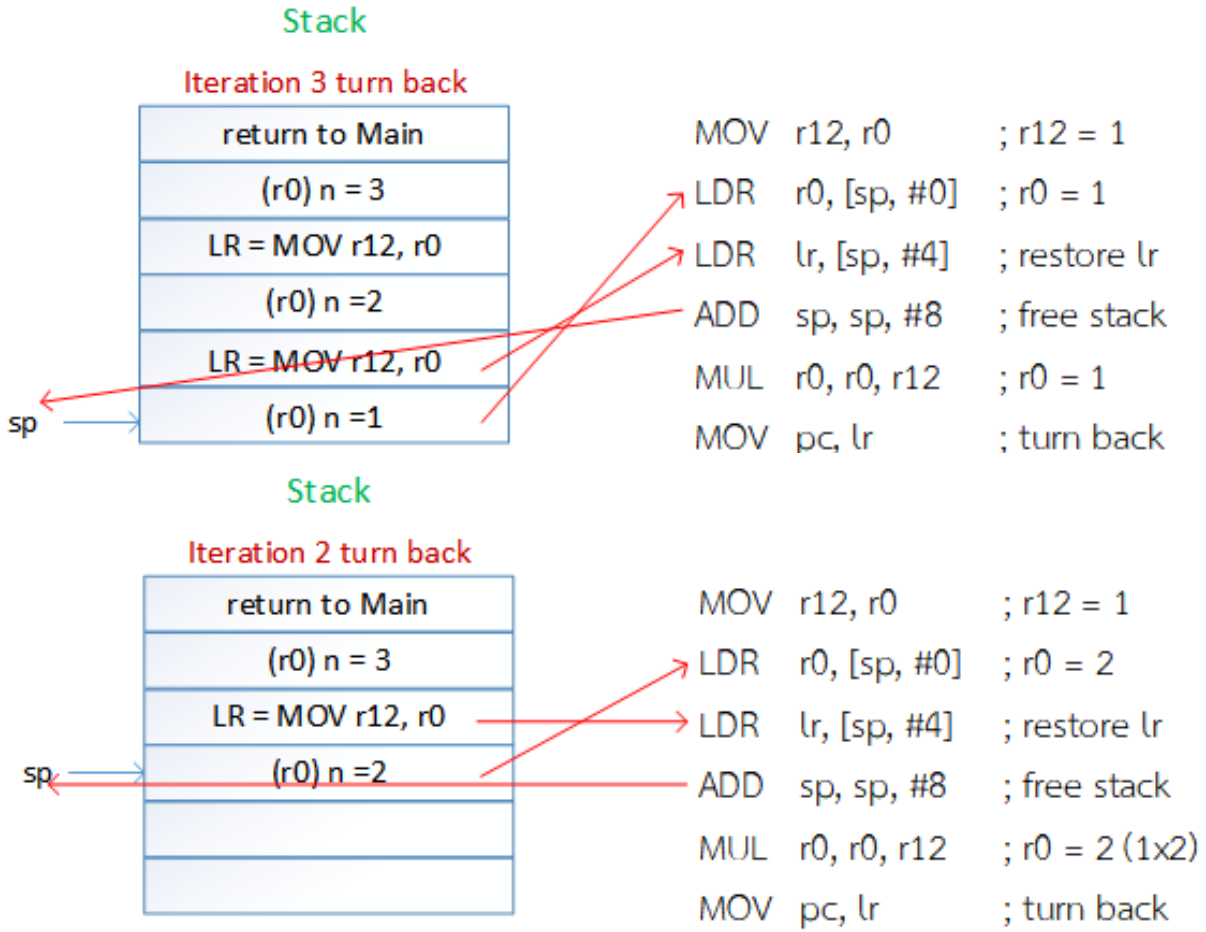





```
fact:   SUB   sp, sp, #8    ; adjust stack for 2 times
        STR   lr, [sp, #4]  ; save the return address
        STR   r0, [sp, #0]  ; save the argument n
CMP     r0, #1              ; compare n to 1
        BGE   L1            ; if n >=1 goto L1
        ...
L1:     SUB   r0, r0, #1    ; n >= 1: argument (n-1)
        BL    fact          ; call fact with (n-1)
```

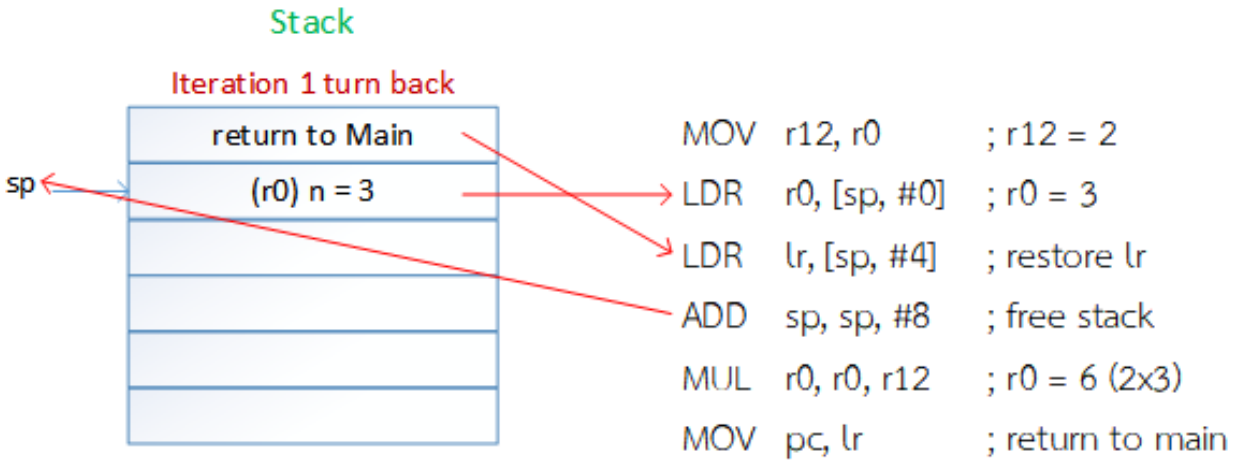
Example

- เมื่อ r0 เป็น 0 แล้ว จะ return 1 ตามเงื่อนไขแรก
- หลังจากนั้น จะเข้าสู่ else



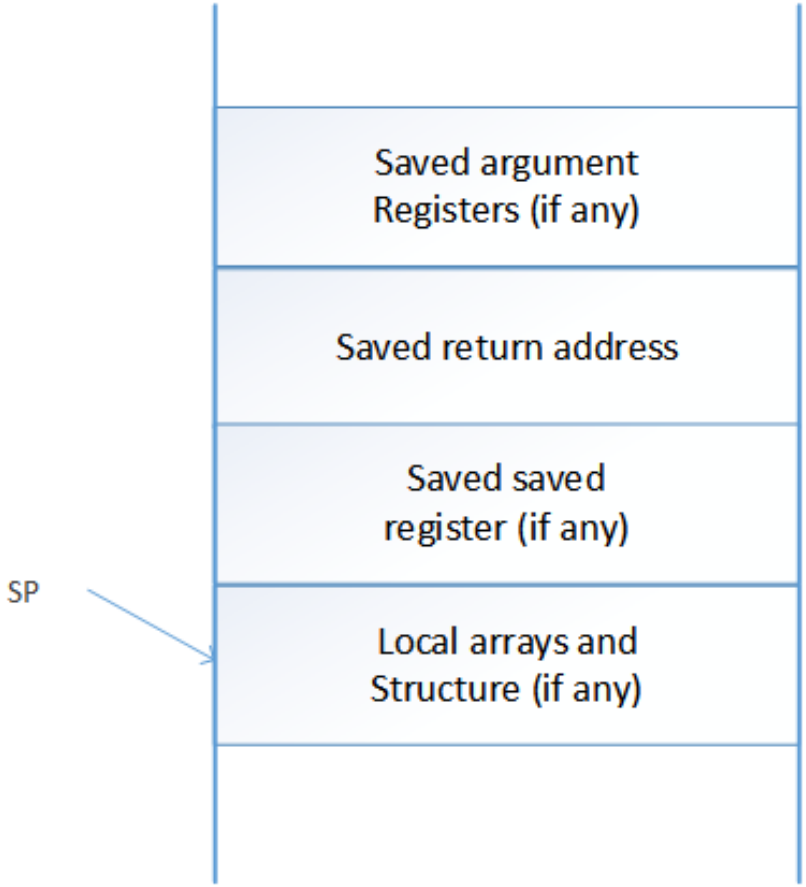
Example

- ครั้งสุดท้ายจะได้ผลลัพธ์เป็น 6 และกลับสู่ main



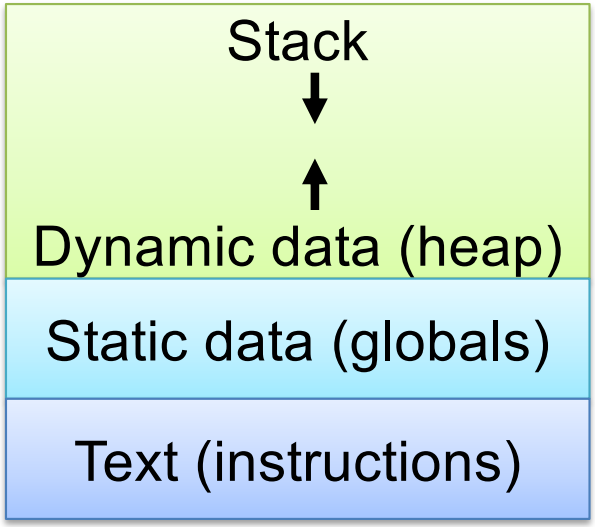
Stack allocation convention

- แนวทางการใช้งาน Stack ที่นิยม
  - ส่วนบนสุด จะเป็น argument register ที่ pass เข้ามา
  - ถัดมาจะเป็น return address
  - ถัดมาจะเป็นการ save” register ที่จะใช้งานในฟังก์ชัน
  - สุดท้ายจะเป็นตัวแปร local ที่ไม่สามารถใช้งานโดยใช้ register ได้ (เช่น array)



Memory Allocation

- การใช้งานหน่วยความจำ นอกเหนือจากที่ใช้งานผ่าน stack แล้ว
- ยังมีการใช้งานหน่วยความจำที่เป็น dynamic allocation อีกอย่างหนึ่ง คือ heap
- Heap เป็นการใช้งานหน่วยความจำอีกรูปแบบหนึ่ง อาจเกิดจากการขอใช้หน่วยความจำโดยตรง เช่น คำสั่ง malloc() หรืออาจเกิดจากโปรแกรมที่เป็น oop สร้าง object ก็ได้
- เนื่องจาก heap จะโตขึ้นด้านบน ในขณะที่ stack จะโตลงมาด้านล่าง ดังนั้นจะต้องเผื่อพื้นที่ของโปรแกรมให้เพียงพอด้วย



Example

- การทำ recursion ที่มีการใช้ stack แต่ก็สามารถจะทำแบบไม่ใช้ stack ได้ในบางกรณี

```
int  sum  (int n, int acc)
{
     if (n > 0)
          return sum(n-1, acc+n);
     else return acc;
}
```

```
sum:    CMP     r0, #0
        BLE     sum_exit
        ADD     r1, r1, r0
        SUB     r0, r0, #1
        B sum
sum_exit:
        MOV     r0, r1
        MOV     pc, lr
```

Suffix S

- ในคำสั่งของ ARM โดยทั่วไปจะไม่ Update ค่าลงใน Flag เช่น คำสั่ง

```
SUB     r0, r0, r0
```

- หากต้องการให้นำผลของการทำงานไป Update ใน Flag ด้วย จะต้องใส่ Suffix S ต่อท้ายคำสั่ง เช่น

```
SUBS    r0, r0, r0
```

- คำสั่งข้างต้นจะทำให้ Zero Flag เป็น 1 (Set)

Cond	F	I	Opcode	S	Rn	Rd	Operand2
4 bits	2 bits	1 bits	4 bits	1 bits	4 bits	4 bits	12 bits

String

- string ในภาษา c จะปิดท้ายด้วยอักขระ \0
- ใน ARM มีคำสั่ง LDRB, STRB สำหรับประมวลผลระดับ byte
- x=r0, y=r1

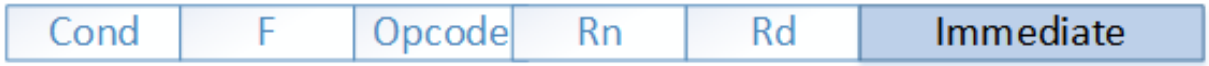
```
int  strcpy (char x[ ], char y[ ])
{
      int i;
      i = 0;
            while ((x[i] = y[i]) != '\0')
                  i += 1;
}
```

```
strcpy:
        SUB     sp, sp , #4
        STR     r4, [sp, #0]
        MOV     r4, #0
L1:     ADD     r2, r4, r1      ; y[i]
        LDRBS   r3, [r2,#0]     ; r3=y[i]
        ADD     r12, r4, r0     ; x[i]
        STRB    r3, [r12, #0]
        BEQ     L2
        ADD     r4, r4, #1      ; i=i+1
        B       L1
L2:     LDR     r4, [sp, #0]
        ADD     sp, sp, #4
        MOV     pc.lr
```

Addressing Mode

- คือรูปแบบคำสั่ง (เน้นที่ DP และ DT) ของ ARM ซึ่งหากเราจำหรือเข้าใจ addressing mode แล้ว จะทำให้ทราบว่าคำสั่งแบบไหนที่ใช้ได้บ้าง

1. Immediate : ADD r2, r0, #5

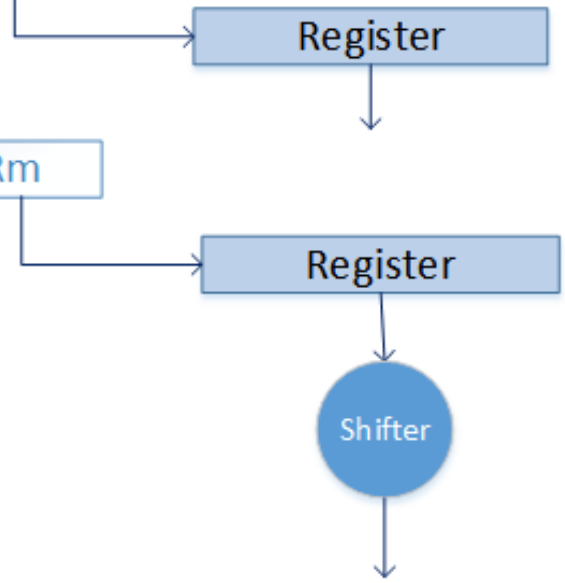


2. Register : ADD r2, r0, r1

3. Scaled register : ADD r2, r0, r1, LSL #2

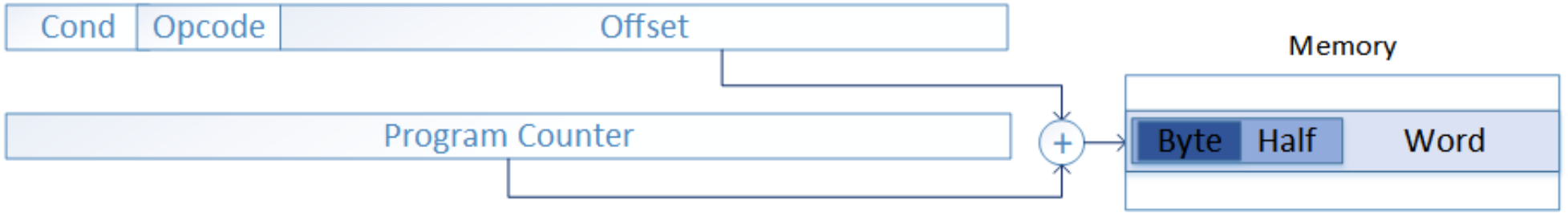
เอา r1 shift จำนวน 2 บิต (x4)

เหมาะกับ array

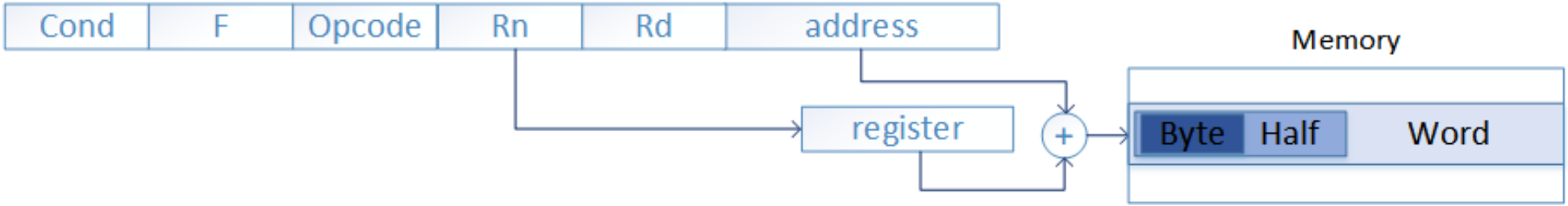


Addressing Mode

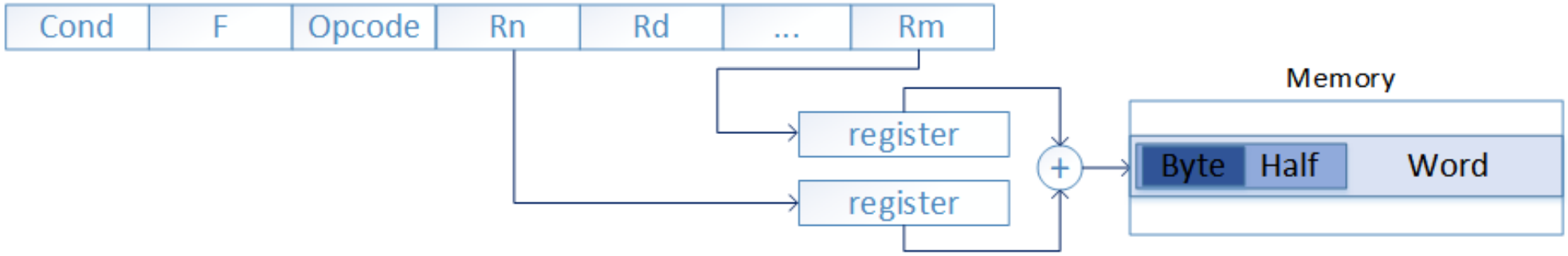
4. PC-relative : BEQ 1000



5. Immediate offset : LDR r2, [r0, #8]

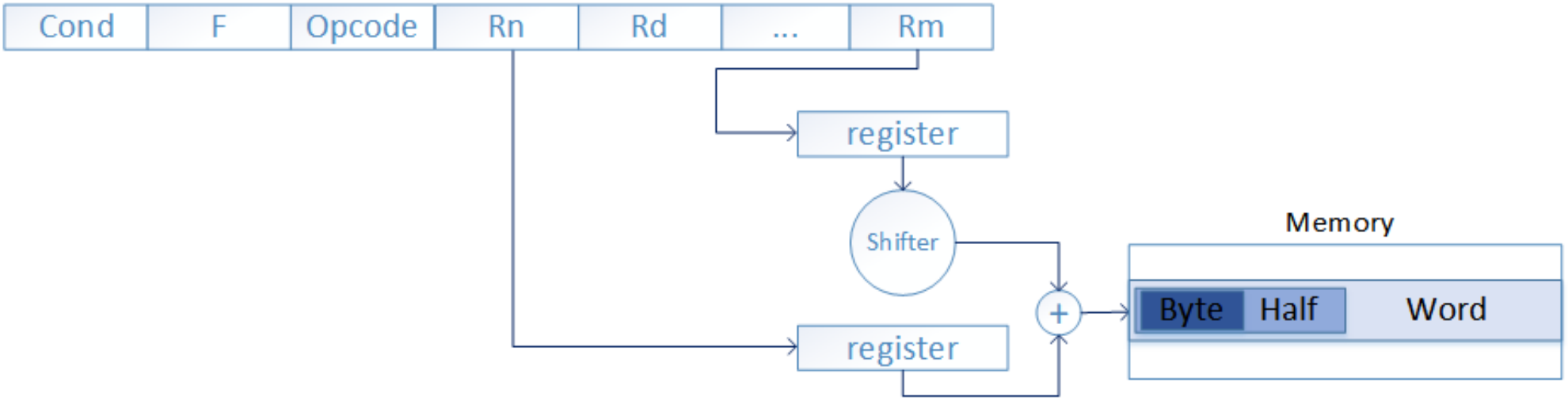


6. Register offset : LDR r2, [r0, r1] ; ใช้รีจิสเตอร์เป็น offset

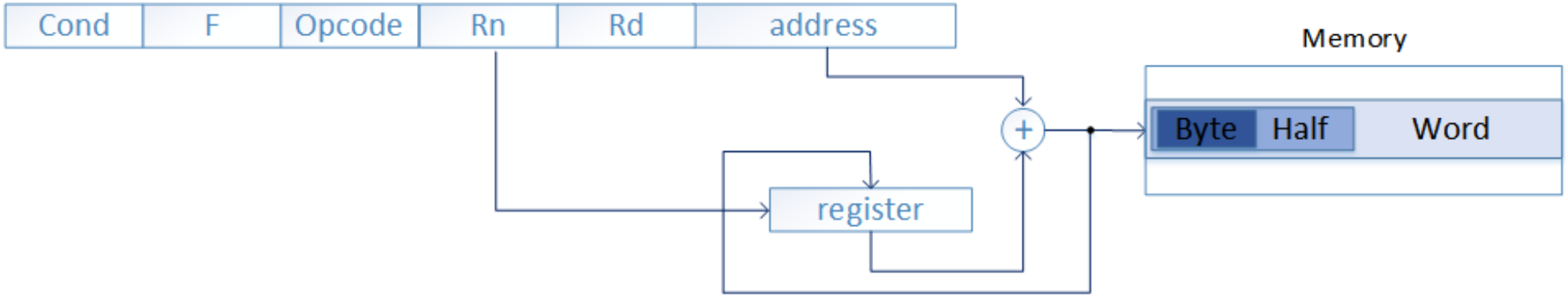


Addressing Mode

7. Scaled register offset : LDR r2, [r0, r1, LSL #2]

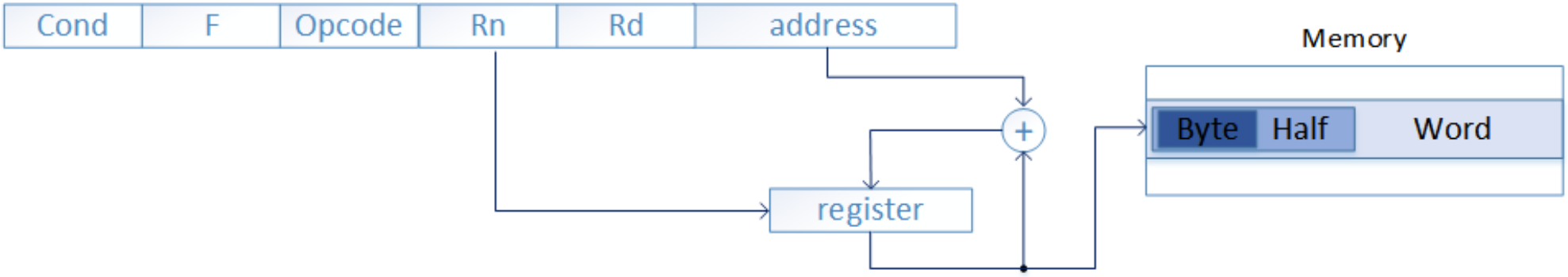


8. Immediate offset pre-indexed : LDR r2, [r0, #4]! ; r0 = r0 + 4

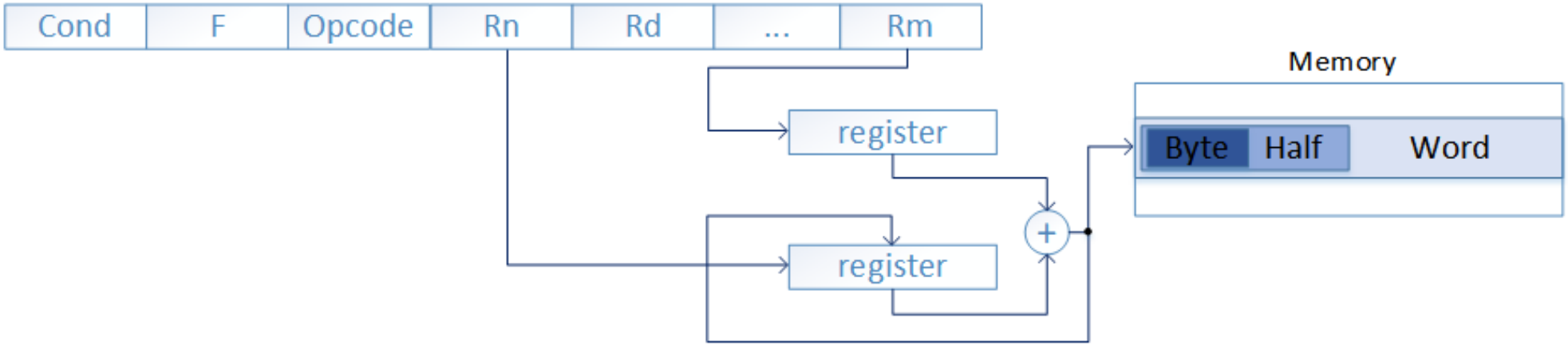


Addressing Mode

9. Immediate offset post-indexed : LDR r2, [r0], #4 ; r0 = r0 + 4 post

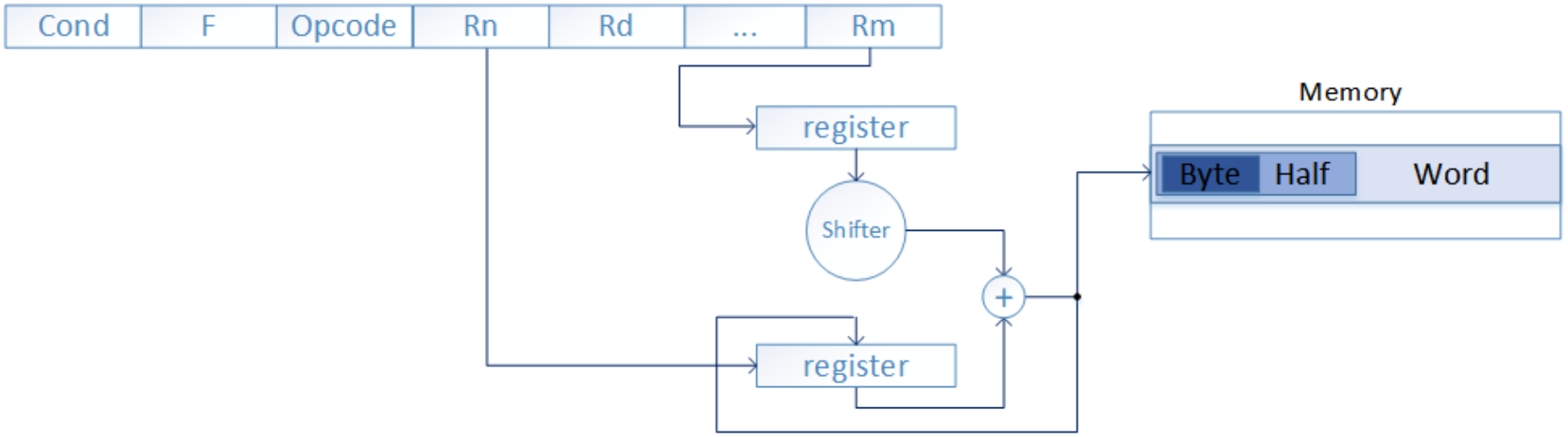


10. Register offset pre-indexed : LDR r2, [r0, r1]!

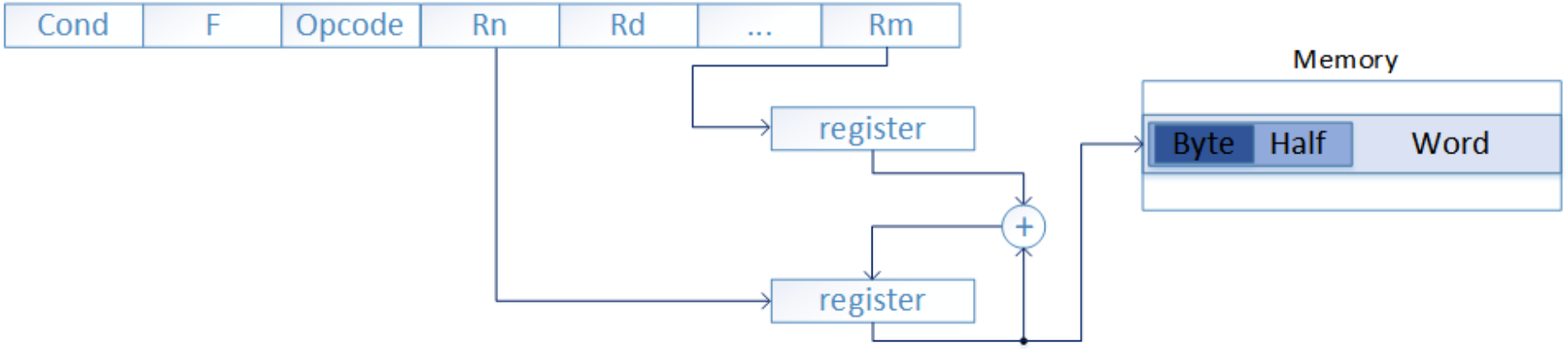


Addressing Mode

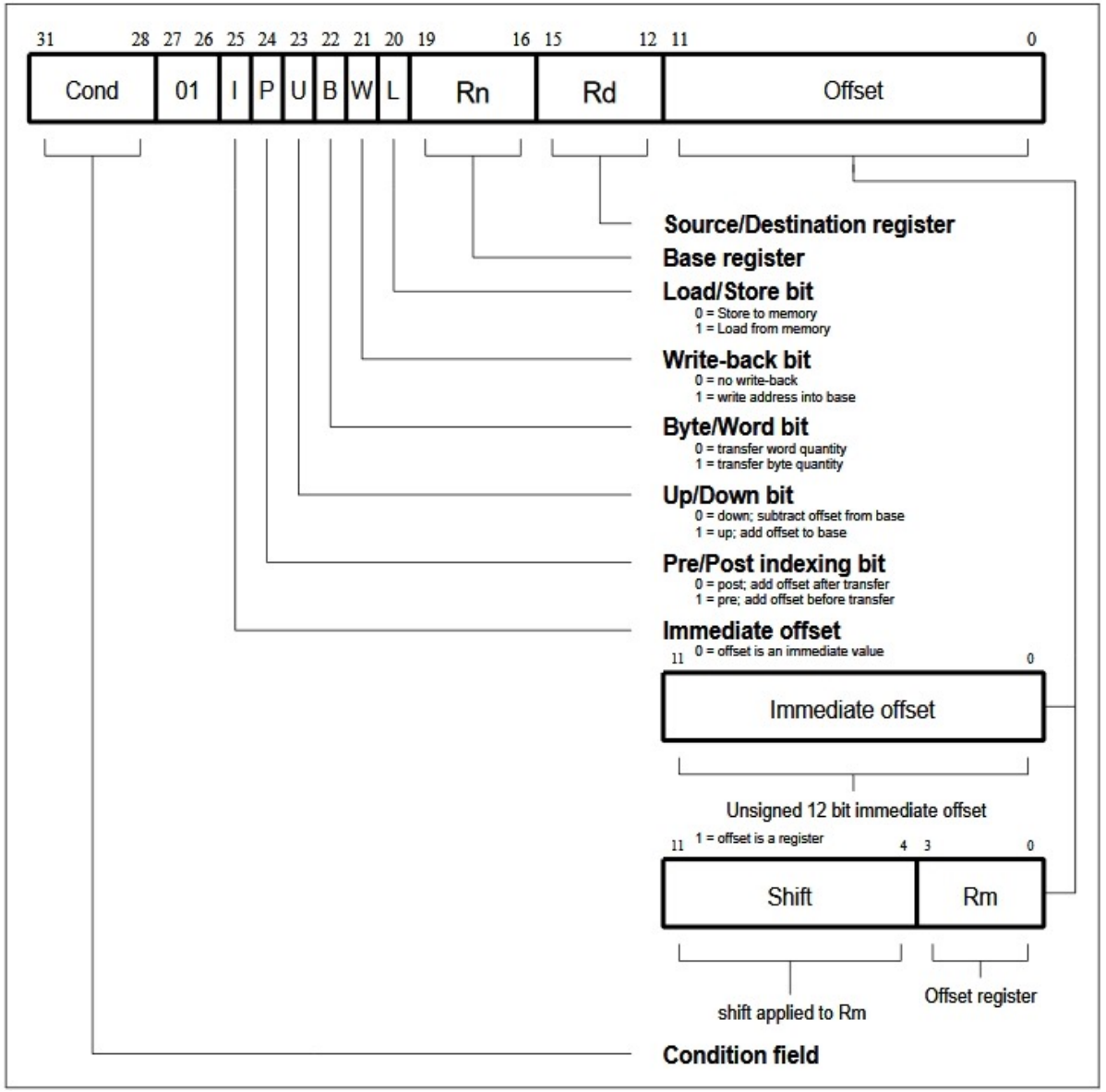
11. Scaled register offset pre-indexed : LDR r2, [r0, r1, LSL #2]



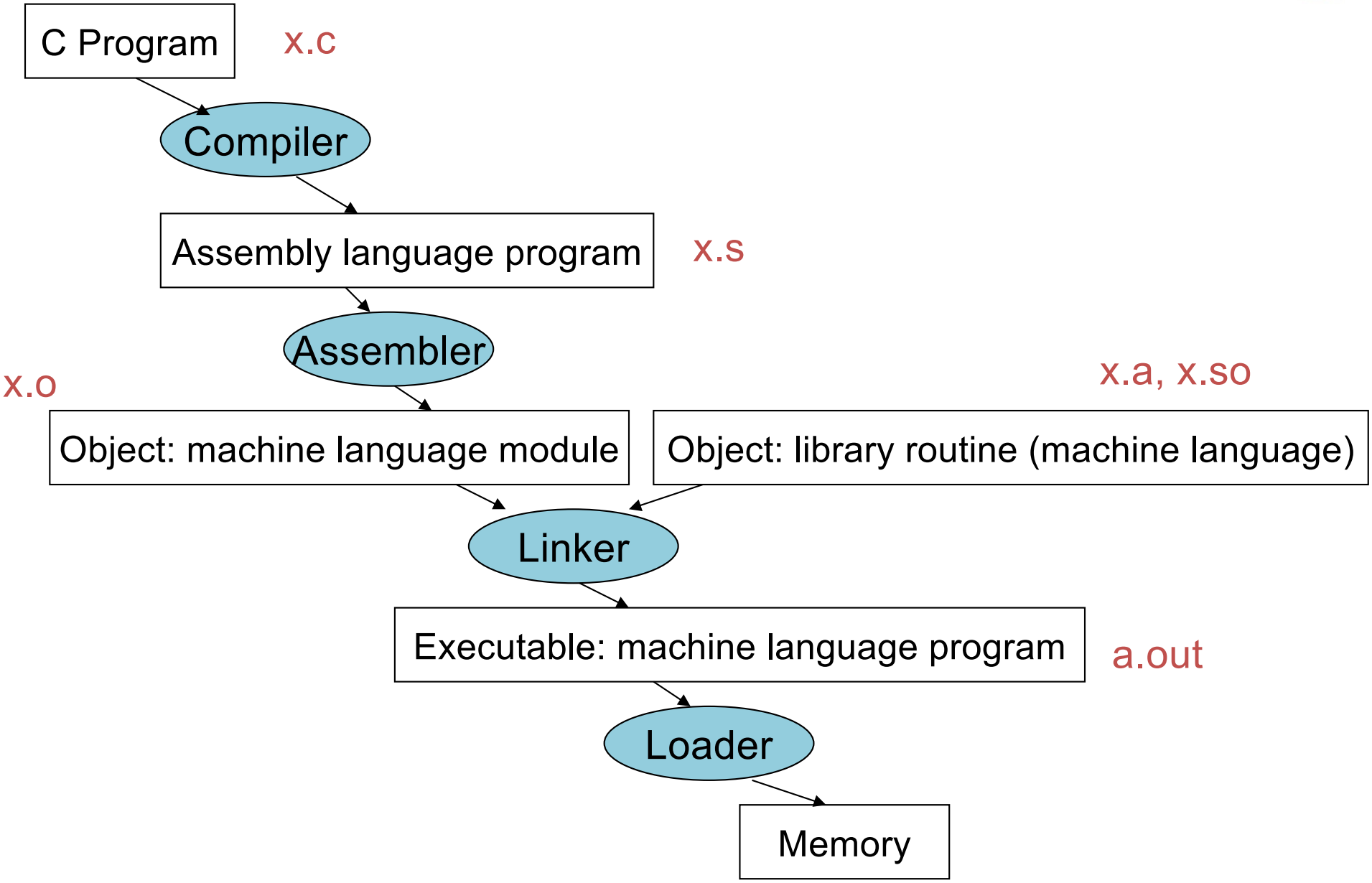
12. Register offset post-indexed : LDR r2, [r0], r1



LDR Instruction



Starting a Program



Role of Assembler

- แปลงภาษา assembly (pseudo instruction ซึ่งยังไม่ใช่ภาษาเครื่อง แต่ใกล้เคียงกับภาษาเครื่อง) ให้เป็นภาษาเครื่อง
- ทำหน้าที่แปลงตัวเลข เพื่อให้เขียนโปรแกรมง่าย เช่น ใช้ฐาน 10 โดย assembler จะแปลงเป็นฐาน 2 ให้
- ผลลัพธ์ของ assembler มีดังนี้
  - Object File ซึ่งประกอบด้วย ข้อมูล และ คำสั่ง
  - Symbol Table เนื่องจากการเขียนโปรแกรม อาจจะมีไฟล์มากกว่า 1 ไฟล์ก็ได้ ดังนั้นในขั้นตอนการแปลนี้ จะยังไม่ทราบว่า Label แต่ละตัวจะอยู่ที่ address ใด ดังนั้นจึงใช้วิธีอ้างอิงกับต้นไฟล์ (นับต้นไฟล์เป็น 0) ดังนั้นในคำสั่งเช่น LDR จะยังไม่รู้ว่าจะโหลดจากแอดเดรสใด จึงต้องทำเป็นตารางเพื่อบอกว่ามีการอ้างตำแหน่งในที่ใดบ้าง

Role of Assembler

- โครงสร้างของ Object File ประกอบด้วย
  - File header : ขนาดและตำแหน่งของแต่ละส่วน
  - Text segment : รหัสภาษาเครื่อง
  - Static data segment : ส่วนของข้อมูล ข้อความ
  - Relocation Information : เก็บตำแหน่งของคำสั่ง หรือ ข้อมูลที่เป็น Absolute address ที่ต้องรอตำแหน่งจริงที่จะเกิดขึ้นตอนที่โหลดลงในหน่วยความจำ
  - Symbol table : เก็บตำแหน่งของคำสั่ง หรือ ข้อมูลที่มีการอ้างอิงภายนอก
  - Debugging information :

Role of Linker

- โปรแกรมที่มีขนาดใหญ่ มักจะมีการแยกเป็นหลายไฟล์ เพื่อสะดวกในการเขียน และการแปลคำสั่ง (Complies) มิฉะนั้น แค่แก้บรรทัดเดียวก็อาจจะต้องแปลใหม่ทั้งหมด
- ทำหน้าที่รวม object file ให้เป็น executable file (รวมทั้ง Library)
- ทำหน้าที่กำหนด address ของคำสั่งและข้อมูล ที่มีการอ้างอิงข้ามไฟล์
- กำหนดการจัดเรียงทั้งคำสั่งและข้อมูล เมื่อโหลดโปรแกรมลงในหน่วยความจำ
- Linker จะใช้ Relocation information และ Symbol table ในการทำงาน

Example

Object File Header	Name	Procedure A	
	Text size	100h	
	Data size	20h	
Text segment	Address	Instruction	
	0	LDR r0, 0(r3)	
	4	BL 0	
	…	…	
Data segment	Address	Instruction	
	0	(X)	
	…	…	
Relocation information	Address	Instruction Type	Dependency
	0	LDR	X
	4	BL	B
Symbol Table	Label	Address	
	X	-	
	B	-	

Example

Object File Header	Name	Procedure A	
	Text size	200h	
	Data size	30h	
Text segment	Address	Instruction	
	0	STR r1, 0(r3)	
	4	BL 0	
	…	…	
Data segment	Address	Instruction	
	0	(Y)	
	…	…	
Relocation information	Address	Instruction Type	Dependency
	0	STR	Y
	4	BL	A
Symbol Table	Label	Address	
	Y	-	
	A	-	

Example

- ใน Proc A จะต้องหา address ของตัวแปร X และ Proc B
- ใน Proc B จะต้องหา address ของตัวแปร Y และ Proc A
- กำหนดให้ text segment เริ่มที่ 40 0000h และ data segment เริ่มที่ 1000 0000h
- ในส่วน text นั้น Proc A ยาว 100h จึงเริ่มที่ 0040 0000h และ Proc B เริ่มที่ 0040 0100h
- ส่วน data นั้น Proc A ยาว 20h จึงเริ่มที่ 1000 0000h และ Proc B เริ่มที่ 1000 0020h

Example

Executable File Header		
	Text size	300h
	Data size	50h
Text segment	Address	Instruction
	0040 0000h	LDR r0, 8000h(r3)
	0040 0004h	BL 00 00ECh
	…	…
	0040 0100h	STR r1, 8020h(r3)
	0040 0104h	BL FF FDDDh
	…	…
Data segment	Address	
	1000 8000h	(X)
	…	…
	1000 8020h	(Y)
	…	…

Example

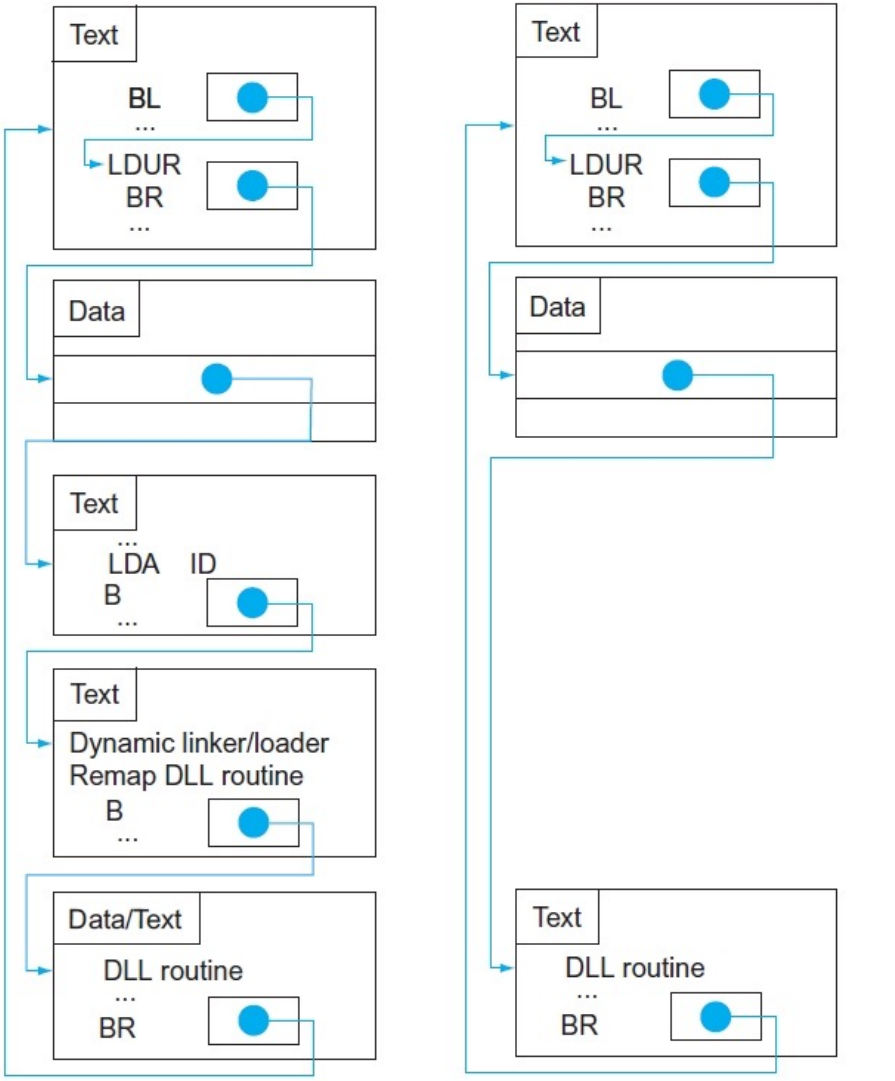
- คำสั่ง BL แรก ใช้ PC-relative addressing เนื่องจากตัวคำสั่งอยู่ที่ 0040 0004h ซึ่งต้องกระโดดไปที่ 0040 0100h ดังนั้น 0040 0100 – 0040 0004 + 8 = 0000 00ECh (+8 เนื่องจาก PC จะเลื่อนไปคำสั่งถัดไปก่อนจะ execute : อธิบายต่อไป)
- คำสั่ง BL ที่สอง ใช้ PC-relative addressing เช่นกัน เนื่องจากตัวคำสั่งอยู่ที่ 0040 0104h ซึ่งต้องกระโดดไปที่ 0040 0000h ดังนั้น 0040 0000 – 0040 0104 + 8 = -112h ซึ่งแปลงเป็น 2’s complement ได้เป็น FFFF FDDDh
- สำหรับคำสั่ง LDR และ STR จะอ้างอิงกับ base register โดยที่นี้สมมติให้ r4 = 1000 0000h ดังนั้นการจะได้ address = 1000 8000 (address of X) ก็จะต้องใส่ค่า 8000h และ 8020h ในคำสั่งที่ 2

Role of Loader

1. อ่าน Header ของ Executable File เพื่อดูขนาดของ text segment และ data segment
2. จองเนื้อที่ใน memory ให้เพียงพอสำหรับ text และ data
3. จองเนื้อที่สำหรับ stack และ heap
4. Copy คำสั่งและข้อมูลลงใน text และ data ที่จองไว้
5. Copy พารามิเตอร์จากระบบ (ถ้ามี) เข้ามาที่ main program (ผ่าน stack)
6. กำหนดค่าเริ่มต้นของ register กำหนด stack pointer
7. กระโดดเข้าสู่ startup routine ซึ่งทำหน้าที่ copy พารามิเตอร์ เข้าสู่ argument register จากนั้นจึง jump เข้าสู่ main program

Dynamic Link Libraries

- เป็นส่วนของโปรแกรม ที่ไม่ได้โหลดไปพร้อมกับโปรแกรมหลัก โดยจะโหลดเมื่อมีการเรียกใช้
  - สามารถ update version ของ Lib ได้ โดยไม่ต้อง Compile ใหม่
  - ประหยัดเนื้อที่ในหน่วยความจำ
- เมื่อเรียกครั้งแรก จะโหลด id ใน data จากนั้น จะเรียก DLL Loader ซึ่งจะนำ id ไปหาไฟล์ที่ต้องโหลด และโหลดเข้าสู่หน่วยความจำ



(a) First call to DLL routine

(b) Subsequent calls to DLL routine

Sort Example

- เพื่อให้เข้าใจมากขึ้น จะยกตัวอย่างโปรแกรมที่ซับซ้อนขึ้น คือ โปรแกรม Sort
- โดยมีโปรแกรมดังนี้

```
void sort (int v[ ], int n)
{
    int i, j;
    for (i=0; i<n; i+=1) {
        for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
            swap (v,j);
        }
    }
}
```

```
void swap (int v[ ], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Sort Example

- เริ่มจาก swap
- กำหนดให้ r0=v, r1=k, temp=r2, temp2=r3
  และ v[k] = r12

```
ADD     r12, r0, r1, LSL #2   ; vkaddr = v+(k*4)
LDR     r2, [r12, #0]         ; temp = v[k]
LDR     r3, [r12, #4]         ; temp2 = v[k+1]
STR     r3, [r12, #0]         ; v[k] = temp2
STR     r2, [r12, #4]         ; v[k+1] = temp
MOV     pc, lr
```

```
void swap (int v[ ], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Sort Example

- กำหนดให้
  - r0 = v
  - r1 = n
  - r2 = i
  - r3 = j
  - r12 = vjaddr
  - r4 = v[j]
  - r5 = v[j+1]
  - r6 = vcopy
  - r7 = ncopy

```
void sort (int v[ ], int n) {
    int i, j;
    for (i=0; i<n; i+=1) {
        for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1)
            swap (v,j);
    }
}
```

Sort Example

- เริ่มจาก for (i=0; i<n; i+=1) กำหนดให้ i = r2, n = r1 เนื่องจากภาษา assembly ไม่สามารถเขียนต่อเนื่องกัน ซึ่ง for loop นี้จะแบ่งเป็น 3 ส่วน คือ ส่วน initial ส่วนเปรียบเทียบ และส่วน increment ซึ่งต้องแยกจากกัน ดังนั้นเราจะเขียนโครงของ for แบบนี้

```
        MOV     r2, #0          ; i = 0
for1:   CMP     r2, r1          ; if i >= n
        BGE     exit1           ; goto exit if I >= n

        ; body of for 1

        ADD     r2, r2, #1      ; i = i + 1
        B       for1
exit1:
```

Sort Example

- ต่อไปเราก็จะทำโครงของ inner loop ในลักษณะเดียวกัน โดยกำหนด r3=j,

  for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1)

```
            SUB     r3, r2, #1              ; j = i - 1
for2:       CMP     r3, #0                  ; if j < 0
            BLT     exit2                   ; goto exit2 if j<0
            ADD     r12, r0, r3, LSL #2     ; r12 = v + (j*4)
            LDR     r4, [r12, #0]           ; r4 = v[j]
            LDR     r5, [r12, #4]           ; r5 = v[j+1]
            CMP     r4, r5                  ; if v[j] <= v[j+1]
            BLE     exit2                   ; goto exit2
            ; body of for 2

            SUB     r3, r3, #1              ; j = j - 1
            B       for2
exit2:
```

Sort Example

- จากนั้นนำมารวมกัน พร้อมจัดทำส่วน save register โดยกำหนดให้ pass parameter ผ่าน v (r0) และ n (r1)

```
        SUB     sp, sp, #20
        STR     lr, [sp, #16]
        STR     r6, [sp, #12]
        STR     r7, [sp, #8]
        STR     r3, [sp, #4]
        STR     r2, [sp, #0]
        MOV     r6, r0          ; backup v
        MOV     r7, r1          ; backup n
        MOV     r2, #0          ; i = 0
for1:   CMP     r2, r1          ; if i >= n
        BGE     exit1           ; exit1
        SUB     r3, r2, #1      ; j = i - 1
for2:   CMP     r3, #0          ; if j < 0
        BLT     exit2           ; exit2
        ADD     r12, r0, r3, LSL #2     ; r12 = v + (j*4)
        LDR     r4, [r12, #0] ; r4 = v[j]
        LDR     r5, [r12, #4] ; r5 = v[j+1]
        CMP     r4, r5          ; if v[j] <= v[j+1]
        BLE     exit2           ; goto exit2
        MOV     r0, r6
        MOV     r1, r3
        BL      swap
        SUB     r3, r3, #1      ; j = j-1
        B       for2
exit2:  ADD     r2, r2, #1      ; i = i + 1
        B       for1
exit1:  LDR     r2, [sp, #0]
        LDR     r3, [sp, #4]
        LDR     r7, [sp, #8]
        LDR     r6, [sp, #12]
        LDR     lr, [sp, #16]
        ADD     sp, sp, #20
        MOV     pc. lr
```

Arrays vs. Pointers

- จุดเด่นที่ทำให้ภาษา c เป็นที่นิยมอย่างมาก คือ pointer ดังนั้นจะแสดงความแตกต่างในการทำงานของ array กับ pointer โดยใช้โปรแกรมต่อไปนี้

```
clear1 (int array[ ], int size)
{
      int i;
      for (i = 0; i < size ; i+=1)
            array[i] = 0;
}
clear2 (int *array, int size)
{
      int *p;
      for (p = &array[0]; p < array[size] ; p = p + 1)
            *p = 0;
}
```

Arrays vs. Pointers

- เวอร์ชัน array กำหนดให้ r0=array, r1=size, r2=i, r3=zero

```
clear1 (int array[ ], int size)
{
    int i;
    for (i = 0; i < size ; i+=1)
        array[i] = 0;
}
```

```
        MOV     r2, #0              ; i = 0
        MOV     r3, #0              ; r3 keep zero
Loop1:  STR     r3, [r0, r2, LSL#2] ; array[i]=0
        ADD     r2, r2, #1          ; i = i+1
        CMP     r2, r1              ; I < size
        BLT     loop1
```

Arrays vs. Pointers

- เวอร์ชัน pointer กำหนดให้ r0=array, r1=size, r2=p, r3=zero, r12=arraysize

```
clear2 (int *array, int size)
{
    int *p;
    for (p = &array[0]; p < array[size] ; p = p + 1)
        *p = 0;
}
```

```
        MOV     r2, r0              ; p = &array[0]
        MOV     r3, #0              ; r3 = 0
        ADD     r12, r0, r1, LSL #2 ; arraySize = address of last array
Loop2:  STR     r3, [r2], #4        ; memory[p] = 0; p = p+4
        CMP     r2, r12             ; p < & array[size]
        BLT     loop2
```

Arrays vs. Pointers

- เมื่อเปรียบเทียบทั้ง 2 code
  - ในรูปแบบ array จำเป็นต้องมีการคูณ (หรือ shift) ใน loop
  - แต่ในรูปแบบ pointer สามารถใช้ posted-index immediete ทำให้ code สั้นลง
  - ดังนั้นในโปรแกรมนี้ Pointer จึงทำงานได้เร็วกว่า

```
        MOV     r2, #0          ; i = 0
        MOV     r3, #0          ; r3 keep zero
Loop1:  STR     r3, [r0, r2, LSL#2] ; array[i]=0
        ADD     r2, r2, #1      ; i = i+1
        CMP     r2, r1          ; I < size
        BLT     loop1
```

```
        MOV     r2, r0          ; p = &array[0]
        MOV     r3, #0          ; r3 = 0
        ADD     r12, r0, r1, LSL #2
                ; arraySize = address of last array
Loop2:  STR     r3, [r2], #4    ; memory[p] = 0
                                ; p = p+4
        CMP     r2, r12         ; p < & array[size]
        BLT     loop2
```

For your attention