

องค์ประกอบคอมพิวเตอร์และภาษาแอสเซมบลี: กรณีศึกษา Raspberry Pi

บทที่ 4 ภาษาแอสเซมบลีของ ARM ขนาด 32 บิต

ผศ.ดร.สุรินทร์ กิตติธรรมกุล

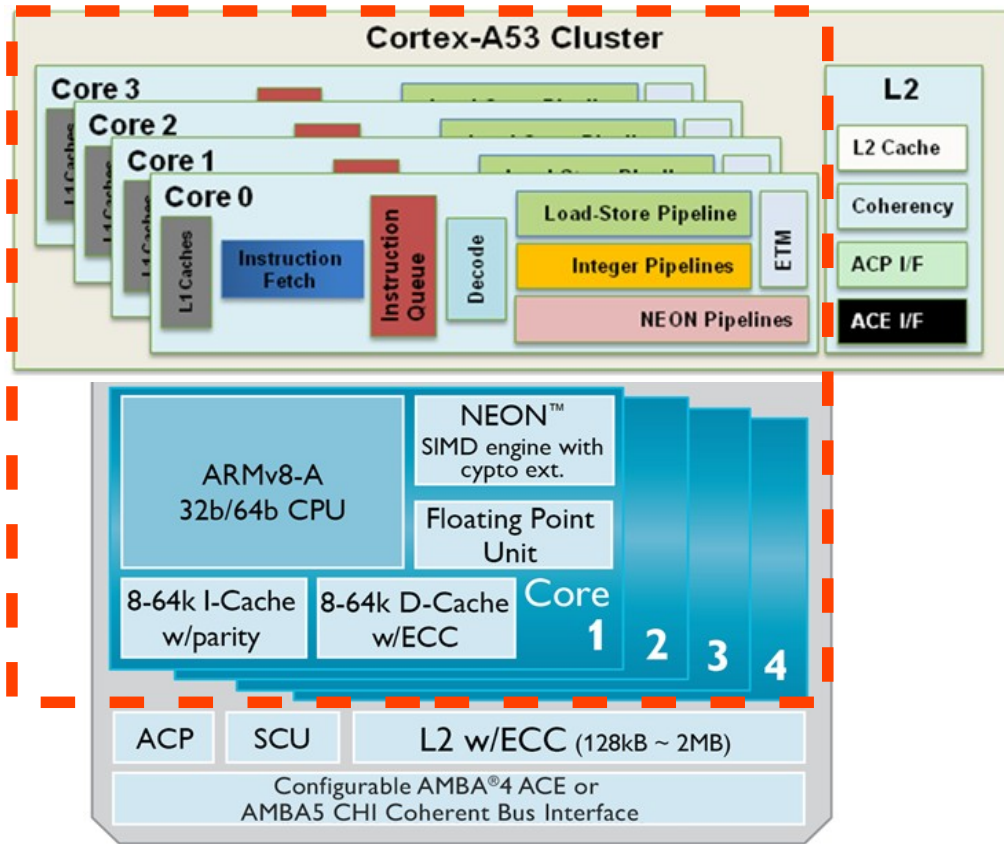
ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

สารบัญ

- 4.1 โครงสร้างของชิพ ARM Cortex A53 ใน BCM2837
- 4.2 สถาปัตยกรรมชุดคำสั่งภาษาแอสเซมบลี (Assembly Instruction Set Architecture)
- 4.3 ตัวอย่างคำสั่งภาษาเครื่องในหน่วยความจำ
- 4.4 การประกาศและตั้งค่าตัวแปรในหน่วยความจำหลัก
- 4.5 คำสั่งถ่ายโอนข้อมูลระหว่างหน่วยความจำและรีจิสเตอร์
- 4.6 คำสั่งประมวลผลข้อมูลในรีจิสเตอร์ (Register Data Processing Instructions)
- 4.7 คำสั่งควบคุมการทำงาน (Control Instructions)
- 4.8 การเรียกใช้ฟังก์ชัน (Function Call)
- 4.9 อุปกรณ์และวิวัฒนาการของชุดคำสั่ง ARM

4.1 โครงสร้างของชิป ARM Cortex A53 ใน BCM2837



- วงจรไปป์ไลน์สำหรับการอ่าน/เขียนค่าตัวแปรกับหน่วยความจำ (Load-Store Pipeline) เพื่ออ่าน/เขียนค่าของตัวแปรกับหน่วยความจำผ่านทางแคชข้อมูล (Data Cache หรือ D-Cache) ลำดับที่ 1
- วงจรไปป์ไลน์สำหรับการประมวลผลเลขจำนวนเต็ม (Integer Pipeline) ขนาด 8, 16, 32 และ 64 บิต ด้วยคำสั่งทางคณิตศาสตร์และตรรกศาสตร์
- วงจรไปป์ไลน์สำหรับการประมวลผลเลขทศนิยมฐานสอง ชนิดจุดลอยตัว ชื่อ NEON (NEON Floating-Point Pipeline) สามารถรองรับการประมวลผลเลขทศนิยมฐานสองชนิดจุดลอยตัว มาตรฐาน IEEE754

4.2 สถาปัตยกรรมชุดคำสั่งภาษาแอสเซมบลี (Assembly Instruction Set Architecture)

- Cortex A53 รองรับการทำงานในโหมด 32 และ 64 บิต นั่นคือ คำสั่งแอสเซมบลี (ภาษาเครื่อง) ความยาว 32 และ 64 บิตตามโหมดการทำงาน
- วิชาี้ จะอ้างอิงคำสั่งแอสเซมบลีเวอร์ชัน 32 บิตของ ARM เนื่องจากบอร์ดติดตั้งระบบปฏิบัติการ Raspbian ซึ่งทำงานในโหมด 32 บิต
 - คำสั่งที่เกี่ยวข้องกับคำสั่งเลขจำนวนเต็มไม่เกิน 32 บิตเท่านั้น
- ARM ยังมีคำสั่งแอสเซมบลีความยาว 16 บิต (Thumb) อีก แต่ทำงานซับซ้อนกว่านิดหน่อย

4.2 สถาปัตยกรรมชุดคำสั่งภาษาแอสเซมบลี (Assembly Instruction Set Architecture)

- ชนิดและขนาดของตัวแปร ผู้อ่านสามารถเทียบเคียงพื้นที่และขนาดของหน่วยความจำ (Memory Space) กับตารางที่ 2.1 โดยข้อมูลหรือตัวแปรแต่ละชนิดต้องการพื้นที่ไม่เท่ากัน ดังนี้
 - ไบต์ (Byte) มีขนาด 8 บิต เหมาะสำหรับตัวแปรชนิดอักขระ เช่น char สำหรับเก็บอักขระตามรหัสมาตรฐาน ASCII ด้วยพื้นที่ 8 บิตในหน่วยความจำ และ unsigned char สำหรับเลขจำนวนเต็มไม่มีเครื่องหมายความยาว 8 บิต ซึ่งได้สรุปไว้ในตารางที่ 2.13
 - ฮาล์ฟเวิร์ด (Halfword) มีขนาด 16 บิต เช่น short และ unsigned short เหมาะสำหรับตัวแปรชนิดอักขระตามมาตรฐาน Unicode
 - เวิร์ด (Word) มีขนาด 32 บิต เหมาะสำหรับตัวแปรชนิดจำนวนเต็ม เช่น int และ unsigned int เป็นต้น
 - ดับเบิลเวิร์ด (Doubleword) 64 บิต เหมาะสำหรับตัวแปรชนิดจำนวนเต็ม เช่น unsigned long long เป็นต้น

4.2 สถาปัตยกรรมชุดคำสั่งภาษาแอสเซมบลี (Assembly Instruction Set Architecture)

- **R15** เรียกว่า โปรแกรมเคาท์เตอร์ (Program Counter: PC) คือ รีจิสเตอร์สำหรับเก็บหมายเลขไบต์ของคำสั่งในเท็กซ์เซ็กเมนต์ของหน่วยความจำเสมือน ที่ซีพียูจะต้องอ่านหรือ เฟทช์ (Fetch) คำสั่งนั้นมาถอดรหัส (Decode) และปฏิบัติ (Execute) ตาม แล้วจึงเปลี่ยนแปลงเป็น $PC=PC+4$ เพื่อเก็บแอดเดรสของคำสั่งถัดไป หมายเลข 4 หน่วยเป็นไบต์ เนื่องจากทุกคำสั่งในตำราเล่มนี้มีความยาว 4 ไบต์ตามที่กล่าวมาข้างต้น
- **R14** เรียกว่า ลิงค์รีจิสเตอร์ (Link Register: LR) คือ รีจิสเตอร์สำหรับเก็บแอดเดรสของคำสั่งที่ต้องการจะรีเทิร์นกลับ โดยรีจิสเตอร์นี้ทำงานคู่กับคำสั่ง BL (Branch and Link) และคำสั่ง BX LR
- **R13** เรียกว่า สแต็คพอยน์เตอร์ (Stack Pointer: SP) คือ รีจิสเตอร์สำหรับเก็บแอดเดรสหรือตำแหน่งยอด (Top) ของสแต็คเซ็กเมนต์ ซึ่งเรียกสั้นๆ ว่า สแต็ค โดยรีจิสเตอร์นี้ทำงานคู่กับคำสั่งที่เกี่ยวข้องกับหน่วยความจำ ในหัวข้อที่ 4.5 ผู้อ่านสามารถทบทวนรายละเอียดเกี่ยวกับสแต็คในหัวข้อที่ 3.2.3 ที่ผ่านมา หมายเหตุ คำว่า สแต็ค ในตำราเล่มนี้หมายถึงสแต็คเซ็กเมนต์ มิได้หมายถึงโครงสร้างข้อมูล (Data Structure) ชนิดหนึ่ง

4.3 ตัวอย่างคำสั่งภาษาเครื่องในหน่วยความจำ

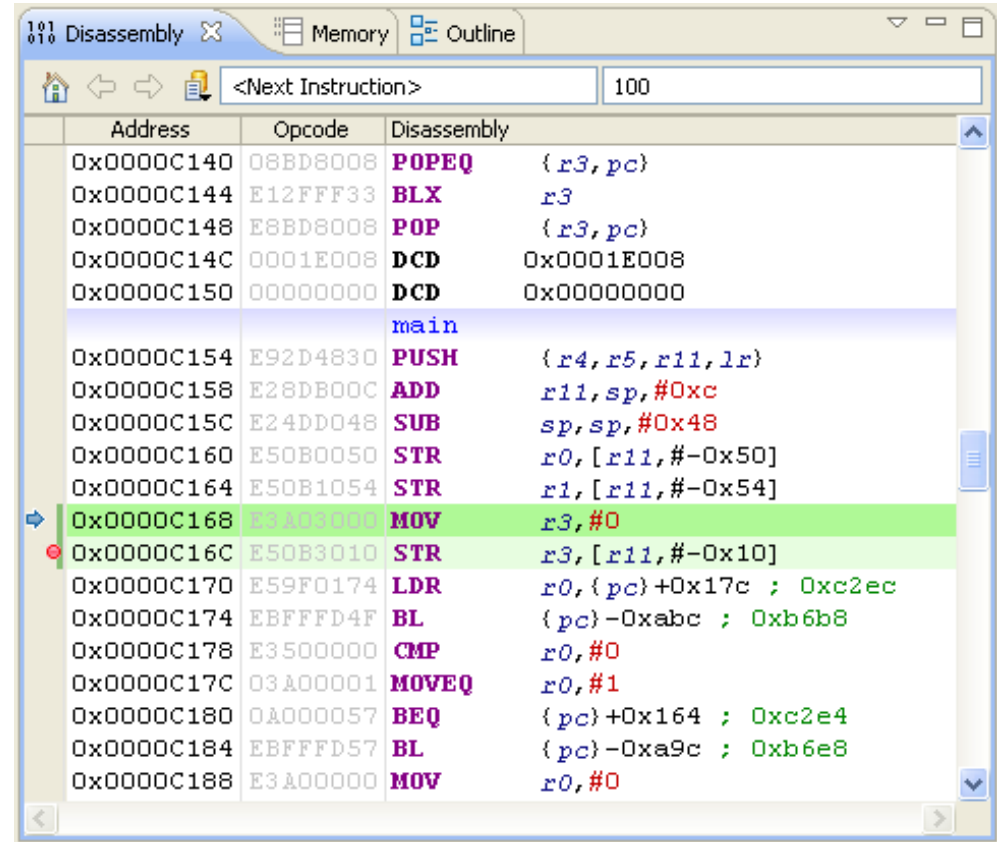
- ดิสแอสเซมบลี (Disassembly) คือ การแปลคำสั่งภาษาเครื่องจากคอลัมน์ออฟโค้ดให้กลับเป็นภาษาแอสเซมบลี ARM เพื่อให้ผู้ใช้โปรแกรมซิมูเลเตอร์เข้าใจ โดยจะแปลออฟโค้ดแต่ละบรรทัด เป็นคำสั่ง 1 คำสั่ง

ยกตัวอย่างเช่น ที่หน่วยความจำตำแหน่ง

PC=0x0000_C140 จำนวน 4 ไบต์บรรจุออฟโค้ด

0x08BD_8008 ซึ่งตรงกับคำสั่ง POPEQ r3, pc

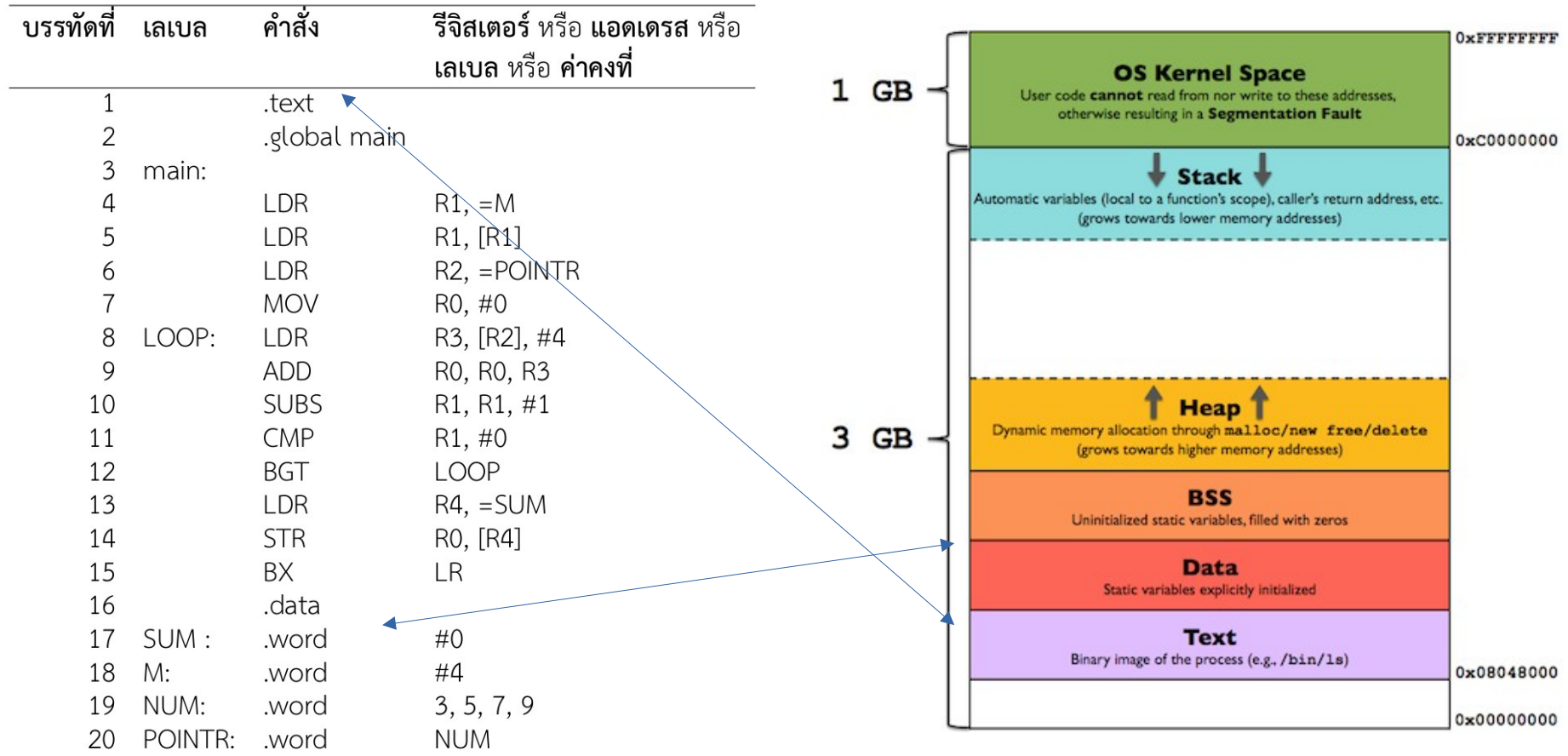
ตำแหน่งถัดไปคือ แอดเดรส PC=0x0000_C144 =
0x0000_C140 + 4 ห่างจากเดิมจำนวน 4 ไบต์



Address	Opcode	Disassembly
0x0000C140	08BD8008	POPEQ {r3, pc}
0x0000C144	E12FFF33	BLX r3
0x0000C148	E8BD8008	POP {r3, pc}
0x0000C14C	0001E008	DCD 0x0001E008
0x0000C150	00000000	DCD 0x00000000
main		
0x0000C154	E92D4830	PUSH {r4, r5, r11, lr}
0x0000C158	E28DB00C	ADD r11, sp, #0xc
0x0000C15C	E24DD048	SUB sp, sp, #0x48
0x0000C160	E50B0050	STR r0, [r11, #-0x50]
0x0000C164	E50B1054	STR r1, [r11, #-0x54]
0x0000C168	E3A03000	MOV r3, #0
0x0000C16C	E50B3010	STR r3, [r11, #-0x10]
0x0000C170	E59F0174	LDR r0, {pc}+0x17c ; 0xc2ec
0x0000C174	EBFFFD4F	BL {pc}-0xabc ; 0xb6b8
0x0000C178	E3500000	CMP r0, #0
0x0000C17C	03A00001	MOVEQ r0, #1
0x0000C180	0A000057	BEQ {pc}+0x164 ; 0xc2e4
0x0000C184	EBFFFD57	BL {pc}-0xa9c ; 0xb6e8
0x0000C188	E3A00000	MOV r0, #0

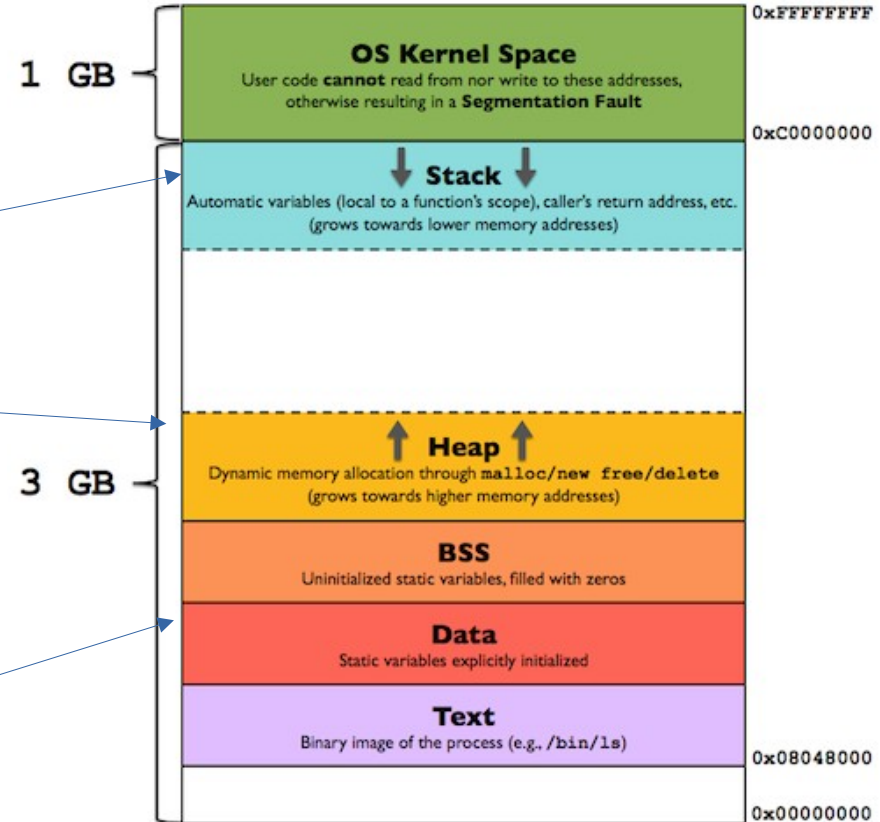
Simulator แสดงค่าใน Memory+Disassembbly

4.4 การประกาศและตั้งค่าตัวแปรในหน่วยความจำหลัก



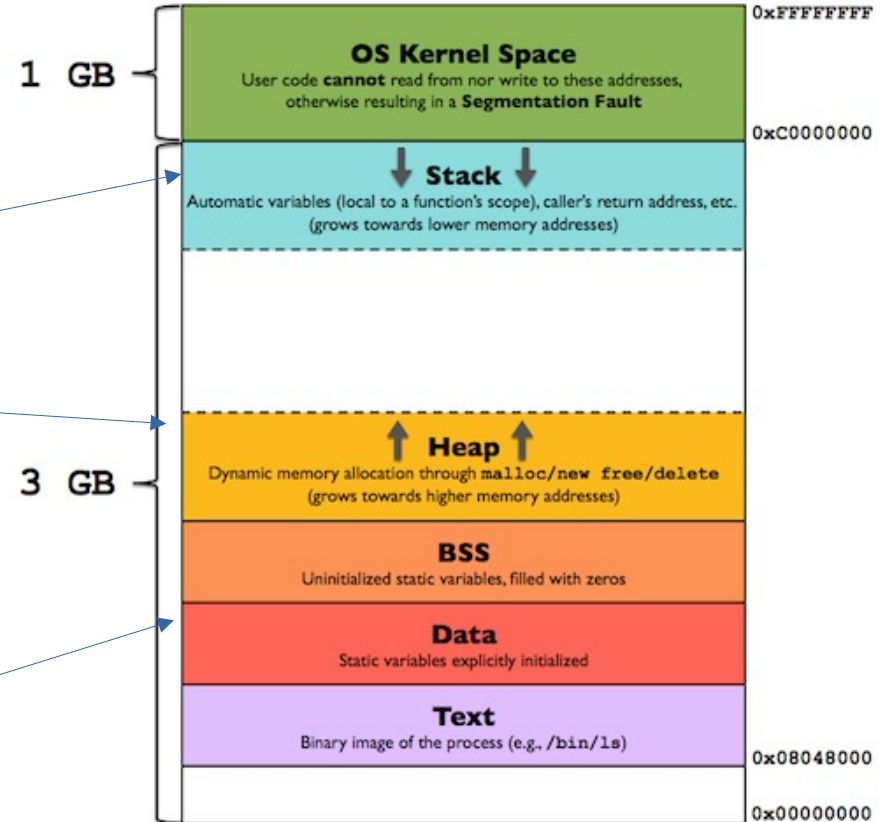
4.4 การประกาศและตั้งค่าตัวแปรในหน่วยความจำหลัก

- โปรแกรมคอมพิวเตอร์ คือ การกักการนำข้อมูลที่มีความสัมพันธ์ต่อกัน และเปลี่ยนแปลงค่าจนเสร็จสิ้นภารกิจ
- สแต็คเช็กเมนต์มีไว้สำหรับเก็บค่ารีจิสเตอร์ LR และรีจิสเตอร์อื่นๆ รวมถึงค่าตัวแปรชนิดโลคอล (Local)
- พื้นที่สำหรับตัวแปร เช่น อะเรย์ โครงสร้างข้อมูลต่างๆ สามารถจองเพิ่มเติมขณะที่กำลังรันโปรแกรมในพื้นที่เรียกว่า **Heap Segment** โดยระบบปฏิบัติการ
- ประกาศตัวแปรโกลบอล (Global) ในหน่วยความจำบริเวณ **Data Segment**



4.4 การประกาศและตั้งค่าตัวแปรในหน่วยความจำหลัก

- โปรแกรมคอมพิวเตอร์ คือ การกักการนำข้อมูลที่มีความสัมพันธ์ต่อกัน และเปลี่ยนแปลงค่าจนเสร็จสิ้นภารกิจ
- สแต็คเช็กเมนต์มีไว้สำหรับเก็บค่ารีจิสเตอร์ LR และรีจิสเตอร์อื่นๆ รวมถึงค่าตัวแปรชนิดโลคอล (Local)
- พื้นที่สำหรับตัวแปร เช่น อะเรย์ โครงสร้างข้อมูลต่างๆ สามารถจองเพิ่มเติมขณะที่กำลังรันโปรแกรมในพื้นที่เรียกว่า **Heap Segment** โดยระบบปฏิบัติการ
- ประกาศตัวแปรโกลบอล (Global) ในหน่วยความจำ บริเวณ **Data Segment**



4.4 การประกาศและตั้งค่าตัวแปรในหน่วยความจำหลัก

รูปแบบ	ความหมาย (Rm และ Rn คือ หมายเลขรีจิสเตอร์มีค่าเท่ากับ R0-R15)	<p>OS Kernel Space User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault 0xFFFFFFFF</p> <p>↓ Stack ↓ Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)</p> <p>↑ Heap ↑ Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)</p> <p>BSS Uninitialized static variables, filled with zeros</p> <p>Data Static variables explicitly initialized</p> <p>Text Binary image of the process (e.g., /bin/ls) 0x08048000</p> <p>0x00000000</p>	
LDR Rd, [Rn]	Rd = Mem[Rn] สำเนาข้อมูล 32 บิต จากหน่วยความจำที่แอดเดรส Rn ไปเขียนในรีจิสเตอร์ Rd		
STR Rd, [Rn]	Mem[Rn] = Rd สำเนาข้อมูล 32 บิตในรีจิสเตอร์ Rd ไปเขียนในหน่วยความจำที่แอดเดรส Rn		
LDRB Rd, [Rn]	Rd = Mem[Rn] สำเนาข้อมูลจำนวน 8 บิตจากหน่วยความจำที่แอดเดรส Rn ไปเขียนในรีจิสเตอร์ Rd		
STRB Rd, [Rn]	Mem[Rn] = Rd สำเนาข้อมูล 8 บิตล่างสุดในรีจิสเตอร์ Rd ไปเขียนในหน่วยความจำที่แอดเดรส Rn		
#	เลเบล	คำสั่ง	คอมเมนต์
1		.data	@Variable definition
2		.ballign 4	@Align variable to 4-byte space
3	wordvar1:	.word 7	@wordvar1=7
4		.ballign 4	@Align variable to 4-byte space
5	wordvar2:	.word 3	@wordvar2=3

4.5 คำสั่งถ่ายโอนข้อมูลระหว่างหน่วยความจำและรีจิสเตอร์

ตารางที่ 4.3: ตัวอย่างโปรแกรมภาษาแอสเซมบลีเพื่อคำนวณประโยค $x = (a + b) - c$

#	เลเบล	คำสั่ง	คอมเมนต์
1		LDR R4, =a	@ get address of variable a
2		LDR R0, [R4]	@ assign value of variable a to R0
3		LDR R4, =b	@ get address of variable b
4		LDR R1, [R4]	@ assign value of variable b to R1
5		ADD R3, R0, R1	@ a+b
6		LDR R4, =c	@ get address of variable c
7		LDR R2, [R4]	@ assign value of variable c to R2
8		SUB R3, R3, R2	@ $x = (a+b)-c$
9		LDR R4, =x	@ get address of variable x
10		STR R3, [R4]	@ store value of R3 to variable x

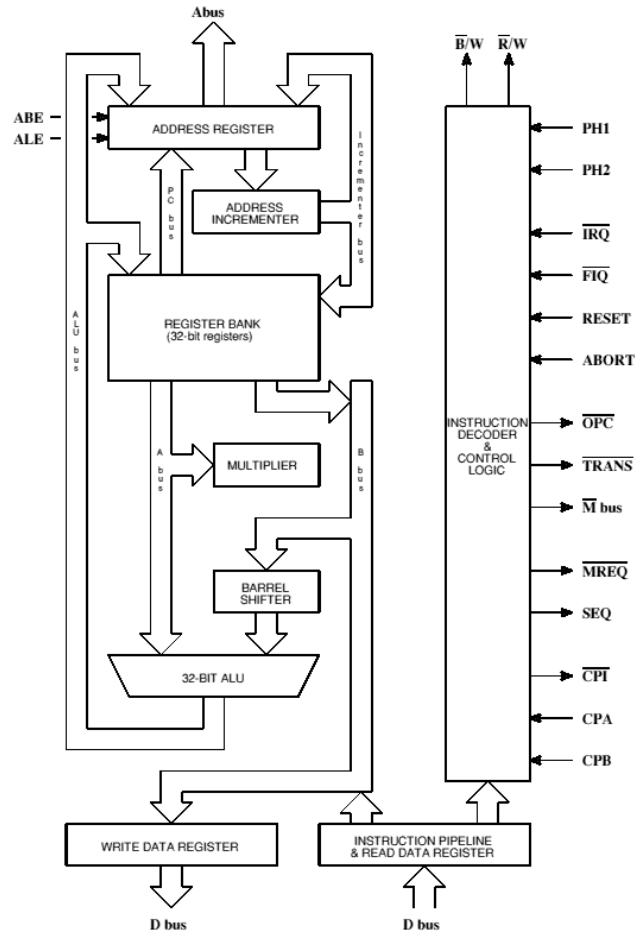
4.5 คำสั่งถ่ายโอนข้อมูลระหว่างหน่วยความจำและรีจิสเตอร์

- เซ็กเมนต์ต่างๆ ยกเว้น Text Segment เป็นพื้นที่สำหรับตัวแปรและโครงสร้างข้อมูลชนิดต่างๆ
- โปรแกรมจะต้องอ่านหรือโหลด (Load) ข้อมูลจากตำแหน่งที่ตัวแปรตั้งอยู่ในหน่วยความจำมาพักในรีจิสเตอร์ก่อน
- โปรแกรมจะคำนวณข้อมูลที่เก็บในรีจิสเตอร์เท่านั้น ด้วยคำสั่งประมวลผลข้อมูลในหัวข้อที่ 4.6
- เมื่อคำนวณแล้วเสร็จ โปรแกรมจะสามารถนำค่าจากรีจิสเตอร์เก็บหรือสโตร์ (Store) ค่าในหน่วยความจำได้ เพื่อให้ผู้ใช้บันทึกตามรูปแบบไฟล์ในอุปกรณ์เก็บรักษาข้อมูลต่อไป
- ภาษาแอสเซมบลี ประเภทนี้ เรียกว่า สถาปัตยกรรมโหลด/สโตร์ (Load/Store Architecture)

4.6 คำสั่งประมวลผลข้อมูลในรีจิสเตอร์ (Register Data Processing Instructions)

- คำสั่งทางคณิตศาสตร์ เพื่อคำนวณเลขจำนวนเต็มชนิดมีและไม่มีเครื่องหมาย
- คำสั่งเลื่อนบิต เพื่อเลื่อนบิตข้อมูลไปทางซ้ายและขวา
- คำสั่งทางคณิตศาสตร์และเลื่อนบิต เพื่อคำนวณเลขจำนวนเต็มชนิดมีและไม่มีเครื่องหมาย หลังจากที่มีการเลื่อนบิตไปทางซ้ายหรือขวา
- คำสั่งทางตรรกศาสตร์ เพื่อคำนวณค่าทางตรรกศาสตร์ของเลขจำนวนเต็มชนิดมีและไม่มีเครื่องหมาย

4.6 คำสั่งประมวลผลข้อมูลในรีจิสเตอร์ (Register Data Processing Instructions)



รูปแบบ

ความหมาย

ADD Rd, Rn, Rm

$Rd = Rn + Rm$

ADD Rd, Rn, #Imm

$Rd = Rn + \#Imm$

SUB Rd, Rn, Rm

$Rd = Rn - Rm$

SUB Rd, Rn, #Imm

$Rd = Rn - \#Imm$

RSB Rd, Rn, Rm

$Rd = Rm - Rn$ (Reverse Subtract)

RSB Rd, Rn, #Imm

$Rd = \#Imm - Rn$ (Reverse Subtract)

MUL Rd, Rn, Rm

$Rd = (Rn * Rm)$ (Only lower 32 bits)

UMULL Rhi, Rlo, Rn, Rm

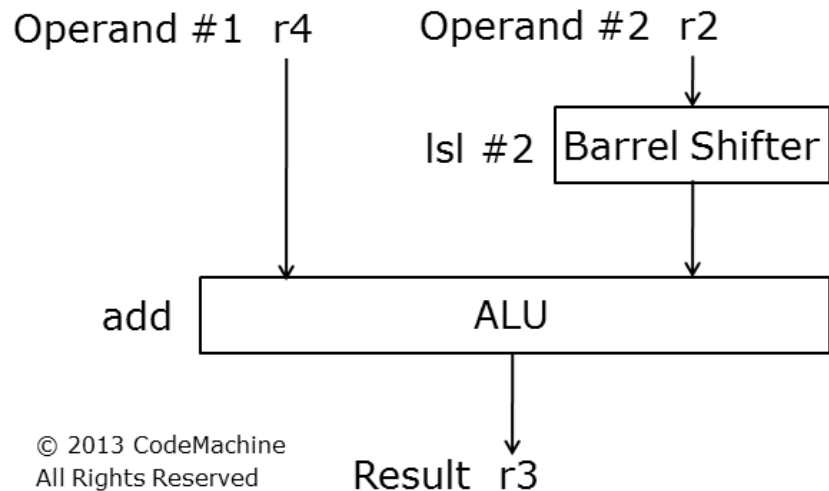
$[Rhi\ Rlo] = (Rn * Rm)$ (Unsigned)

SMULL Rhi, Rlo, Rn, Rm

$[Rhi\ Rlo] = (Rn * Rm)$ (Signed)

4.6 คำสั่งประมวลผลข้อมูลในรีจิสเตอร์ (Register Data Processing Instructions)

```
add r3, r4, r2, lsl #2; r3 = r4 + (r2 << 2)
```



รูปแบบ	ความหมาย
ADD Rd, Rn, Rm LSL #shmt	$Rd = Rn + (Rm \ll \#shmt)$
ADD Rd, Rn, Rm LSR #shmt	$Rd = Rn + (Rm \gg \#shmt)$
ADD Rd, Rn, Rm ASR #shmt	$Rd = Rn + (Rm \gg \#shmt)$ (Signed)

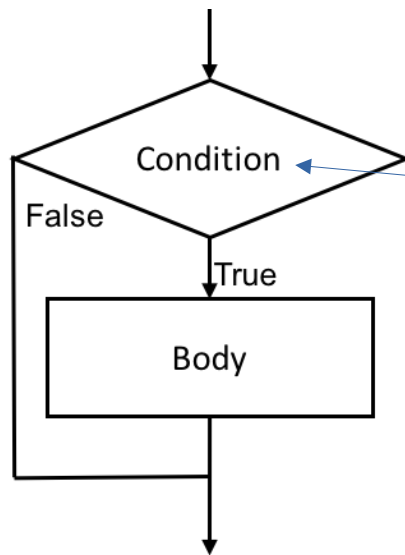
4.6 คำสั่งประมวลผลข้อมูลในรีจิสเตอร์ (Register Data Processing Instructions)

```
unsigned int a = 0xffff0000;
unsigned int b = 0x0000ffff;
unsigned int c;
c = a & b; /* c = 0x00000000 AND */
c = a | b; /* c = 0xffffffff OR */
c = !b;    /* c = 0xffff0000 NOT */
c = a ^ b; /* c = 0xffffffff Ex-OR */
```

รูปแบบ	ความหมาย
AND Rd, Rn, Rm	$Rd = Rn \& Rm$ (bitwise AND)
AND Rd, Rn, #Imm	$Rd = Rn \& \#Imm$ (bitwise AND)
ORR Rd, Rn, Rm	$Rd = Rn Rm$ (bitwise OR)
ORR Rd, Rn, #Imm	$Rd = Rn \#Imm$ (bitwise OR)
MVN Rd, Rm	$Rd = \bar{Rm}$ (bitwise Inverse)
MVN Rd, #Imm	$Rd = \bar{\#Imm}$ (bitwise Inverse)
EOR Rd, Rn, Rm	$Rd = Rn \oplus Rm$ (bitwise XOR)
EOR Rd, Rn, #Imm	$Rd = Rn \oplus \#Imm$ (bitwise XOR)

4.7 คำสั่งควบคุมการทำงาน (Control Instructions): if

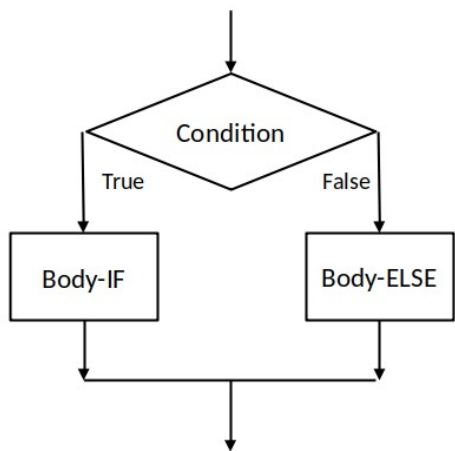
```
if ((a+b)>c) {  
    x+=y; /* Body */  
}
```



#	เลเบล	คำสั่ง	คอมเม้นท์
1		LDR R4, =a	@ get address of variable a
2		LDR R0, [R4]	@ get value of variable a
3		LDR R4, =b	@ get address of variable b
4		LDR R1, [R4]	@ get value of variable b
5		ADD R3, R0, R1	@ compute a+b
6		LDR R4, =c	@ get address of variable c
7		LDR R2, [R4]	@ get value of variable c
8		CMP R3, R2	@ compute (a+b)-c
9		BLE exit	@ jump to exit if R3 <= R2
10		LDR R4, =x	@ get address of variable x
11		LDR R5, [R4]	@ get value of variable x
12		LDR R4, =y	@ get address of variable y
13		LDR R6, [R4]	@ get value of variable y
14		ADD R5, R5, R6	@ x += y
15		LDR R4, =x	@ get address of variable x
16		STR R5, [R4]	@ store value of variable x
17	exit	...	@ exit label

4.7 คำสั่งควบคุมการทำงาน (Control Instructions): if-else

```
if ((a+b)>c) {  
    x+=y; /* Body-IF */  
}  
else {  
    x-=y; /* Body-ELSE */  
}
```



#	เลเบล	คำสั่ง	คอมเมนต์	
1		LDR R4, =a	14	ADD R5, R5, R6
2		LDR R0, [R4]	15	LDR R4, =x
3		LDR R4, =b	16	STR R5, [R4]
4		LDR R1, [R4]	17	B exit
5		ADD R3, R0, R1	18	else
6		LDR R4, =c	19	
7		LDR R2, [R4]	20	
8		CMP R3, R2	21	LDR R4, =y
9		BLE else	22	LDR R6, [R4]
10		LDR R4, =x	23	SUB R5, R5, R6
11		LDR R5, [R4]	24	LDR R4, =x
12		LDR R4, =y	25	STR R5, [R4]
13		LDR R6, [R4]	25	exit
14		ADD R5, R5, R6	25	

4.7 คำสั่งควบคุมการทำงาน (Control Instructions)

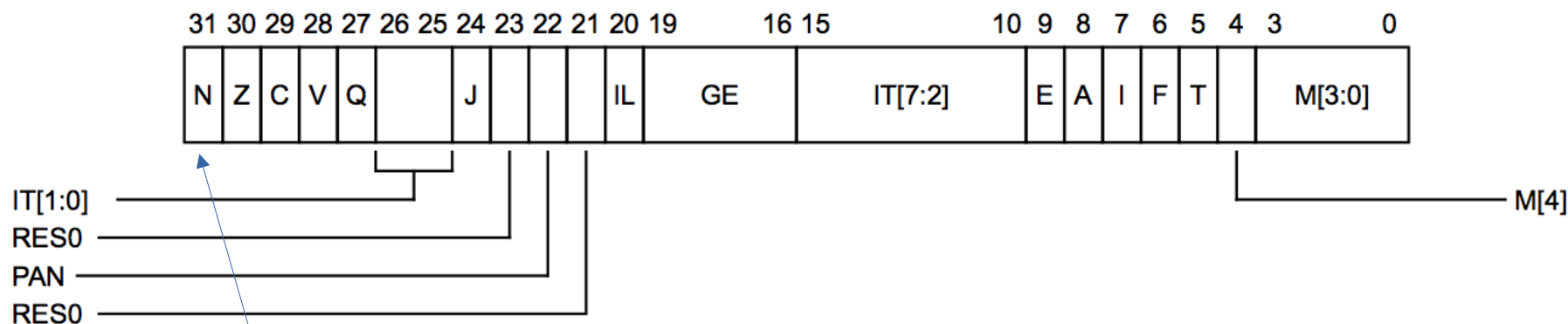
คำสั่งควบคุมการทำงาน (Control Instructions) ในภาษาสูงแบ่งเป็น ประโยคการตัดสินใจ เช่น ประโยค IF, IF-ELSE, Switch-Case และประโยคการวนรอบชนิดต่างๆ เช่น FOR, WHILE, DO-WHILE เป็นต้น โครงสร้างคำสั่งภาษาแอสเซมบลีของ ARM ที่รองรับประโยคเหล่านี้ ประกอบด้วย

- คำสั่ง **CMP** (Compare) ทำหน้าที่เปรียบเทียบระหว่างค่ารีจิสเตอร์กับค่าคงที่ หรือระหว่างค่ารีจิสเตอร์ 2 ตัว โปรแกรมเมอร์สามารถเขียนคำสั่งนี้ได้ 2 รูปแบบตามตารางต่อไปนี้

รูปแบบ	ความหมาย
CMP Rn, Rm	$NZCV \leftarrow Rn - Rm$
CMP Rn, #Imm	$NZCV \leftarrow Rn - \#Imm$

ค่าบิต NZCV ในรีจิสเตอร์ CPSR (หัวข้อที่ 4.6.1) คือ ผลลัพธ์ที่ได้จากคำสั่ง CMP โดยคำสั่งนี้ทำการเปรียบเทียบเลขจำนวนเต็มสองค่า ($Rn - Rm$) หรือ ($Rn - \#Imm$)

4.7 คำสั่งควบคุมการทำงาน (Control Instructions)



BGT label pc = address(label)

เฟทช์คำสั่งที่ตามหลัง label หากผลลัพธ์การเปรียบเทียบมากกว่า

BLT label pc = address(label)

เฟทช์คำสั่งที่ตามหลัง label หากผลลัพธ์การเปรียบเทียบน้อยกว่า

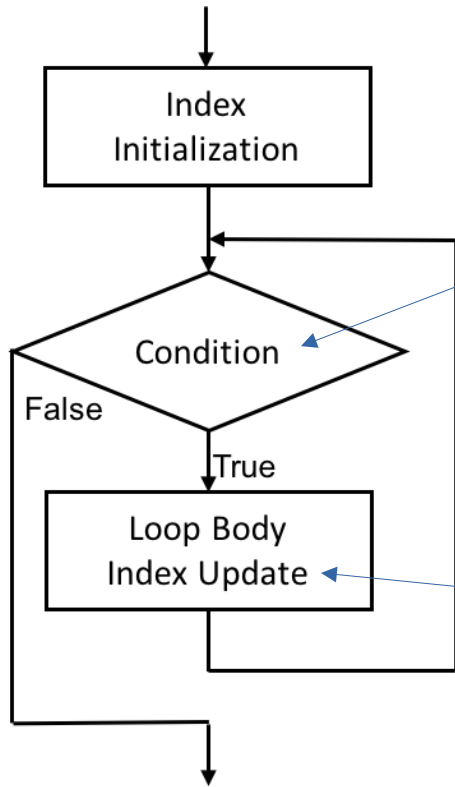
BGE label pc = address(label)

เฟทช์คำสั่งที่ตามหลัง label หากผลลัพธ์การเปรียบเทียบมากกว่าหรือเท่ากับ

BLE label pc = address(label)

เฟทช์คำสั่งที่ตามหลัง label หากผลลัพธ์การเปรียบเทียบน้อยกว่าหรือเท่ากับ

4.7 คำสั่งควบคุมการทำงาน (Control Instructions): ลูป for

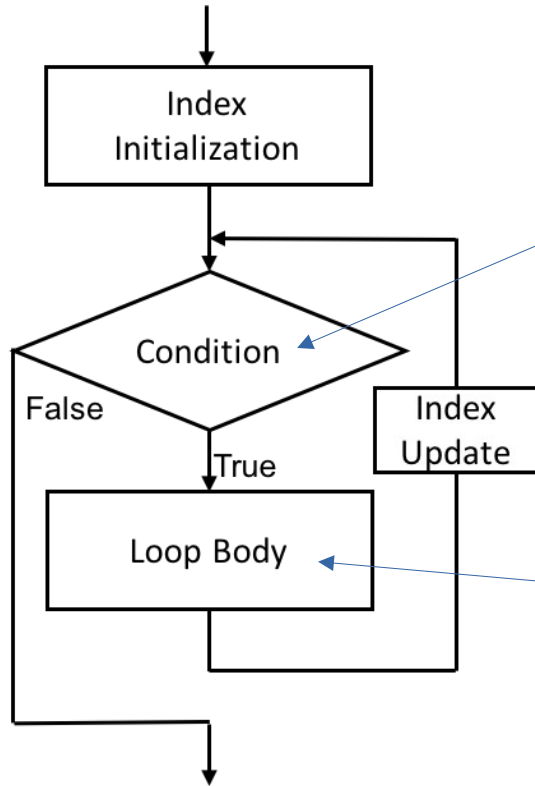


```
x=0;
for (i=1; i<=10; i=i+1) {
    x=x+i;  /* Body */
}
```

$$x = \sum_{i=1}^{10} i$$

#	เลเบล	คำสั่ง	คอมเม้นท์
1		...	@ Initialize R0=0
2		...	@ Initialize R1=1
3	for:	CMP R1, #10	@ Compare R1 with 10
4		BGT end	@ if greater than goto end
5		ADD R0, R0, R1	@ else R0 = R0 + R1
6		ADD R1, R1, #1	@ Increment R1 by 1
7		B for	@ Branch back to Label for
8	end:	...	@ End of for loop

4.7 คำสั่งควบคุมการทำงาน (Control Instructions): ลูป while

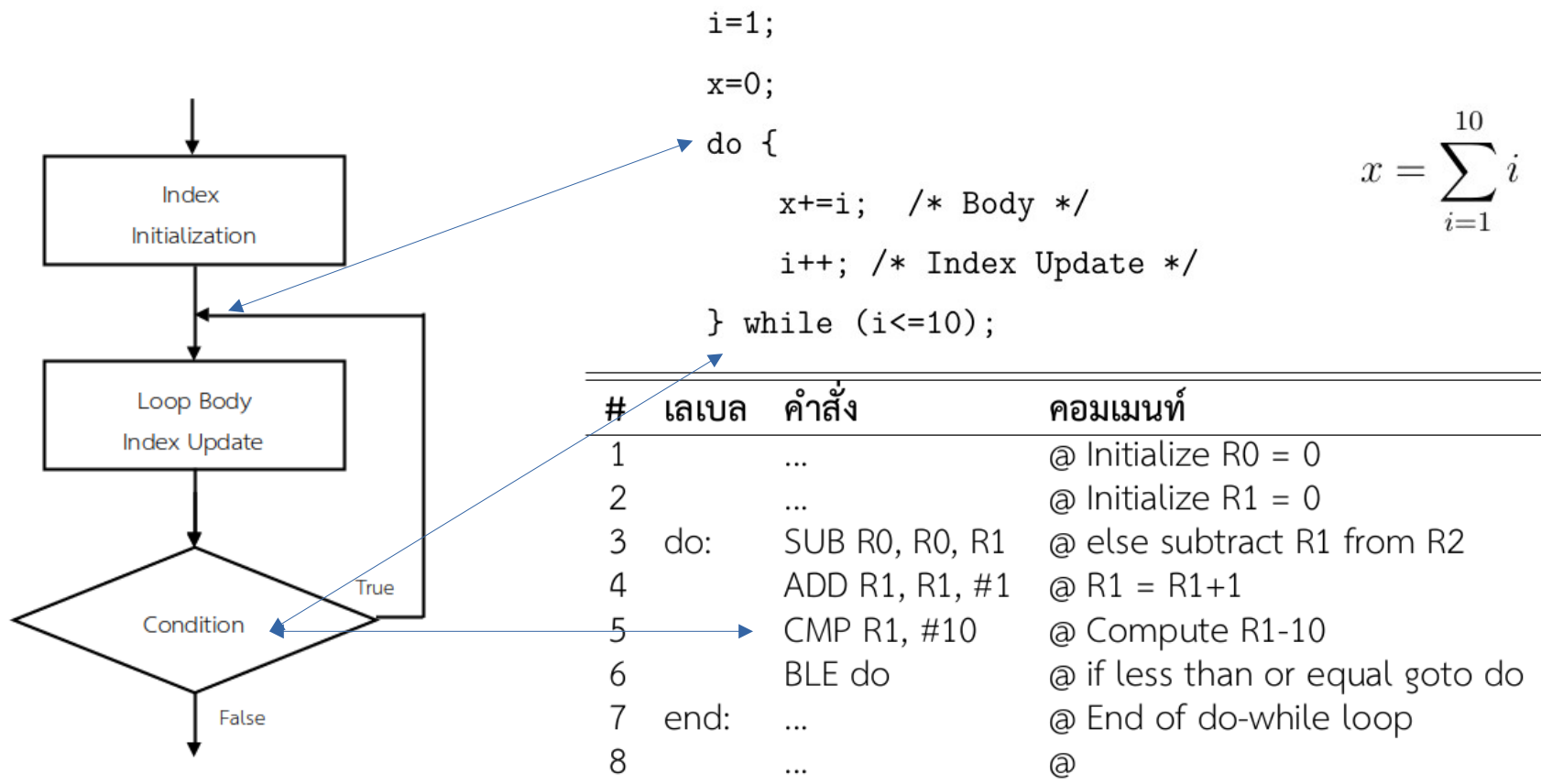


```
i=1;
x=0;
while (i<=10) {
    x+=i; /* Body */
    i++; /* Index Update */
}
```

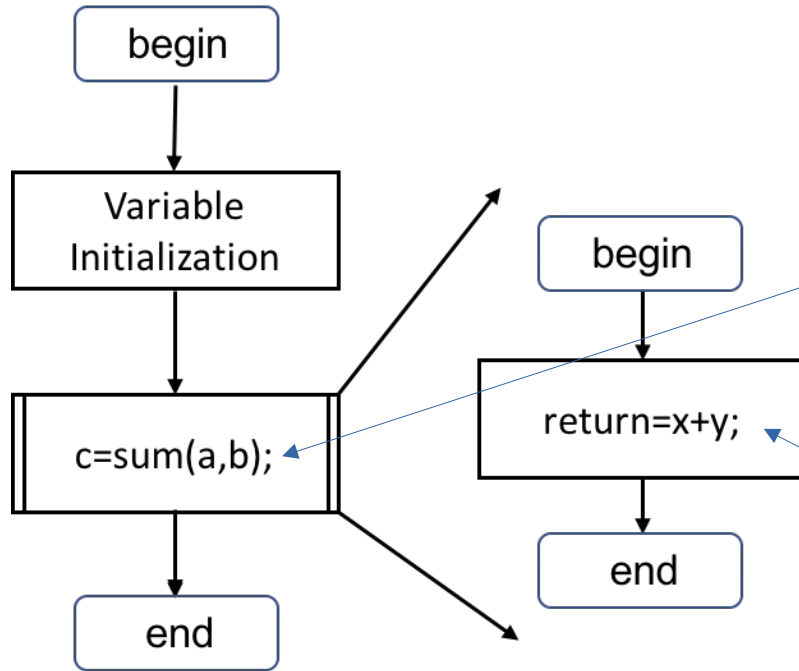
$$x = \sum_{i=1}^{10} i$$

#	เลเบล	คำสั่ง	คอมเม้นท์
1	@ Initialize R0=0
2	@ Initialize R1=1
3	while:	CMP R1, #10	@ Compare R1 with 10
4		BGT end	@ if greater than goto end
5		ADD R0, R0, R1	@ R0 = R0+R1
6		ADD R1, R1, #1	@ Increment R1 by 1
7		B while	@ Branch back to Label while
8	end:	...	@ End of while loop
9	@

4.7 คำสั่งควบคุมการทำงาน (Control Instructions): ลูป do_while

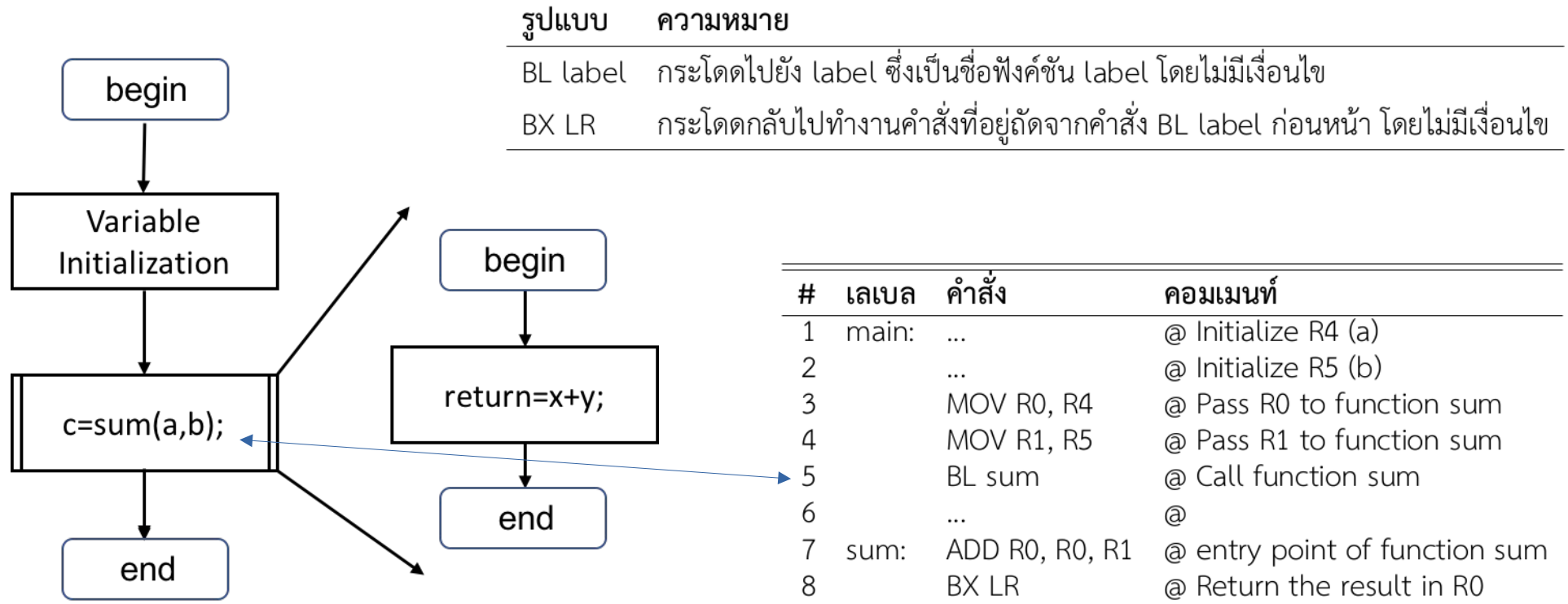


4.8 การเรียกใช้ฟังก์ชัน (Function Call)

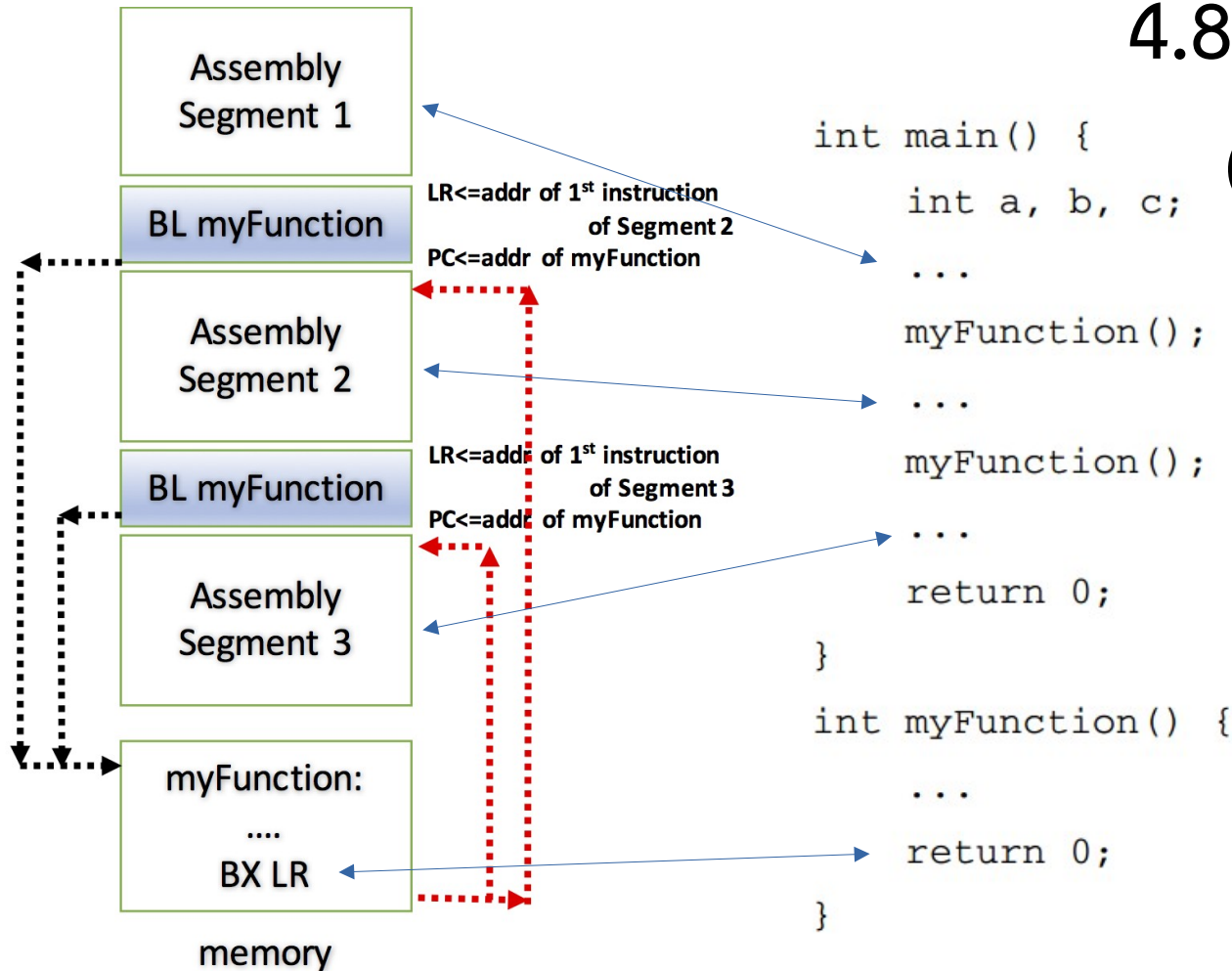


```
int main() {  
    int a, b, c;  
    ...  
    c = sum(a,b);  
    ...  
    return 0;  
}  
  
int sum(int x, int y) {  
    return x+y;  
}
```

4.8 การเรียกใช้ฟังก์ชัน (Function Call)

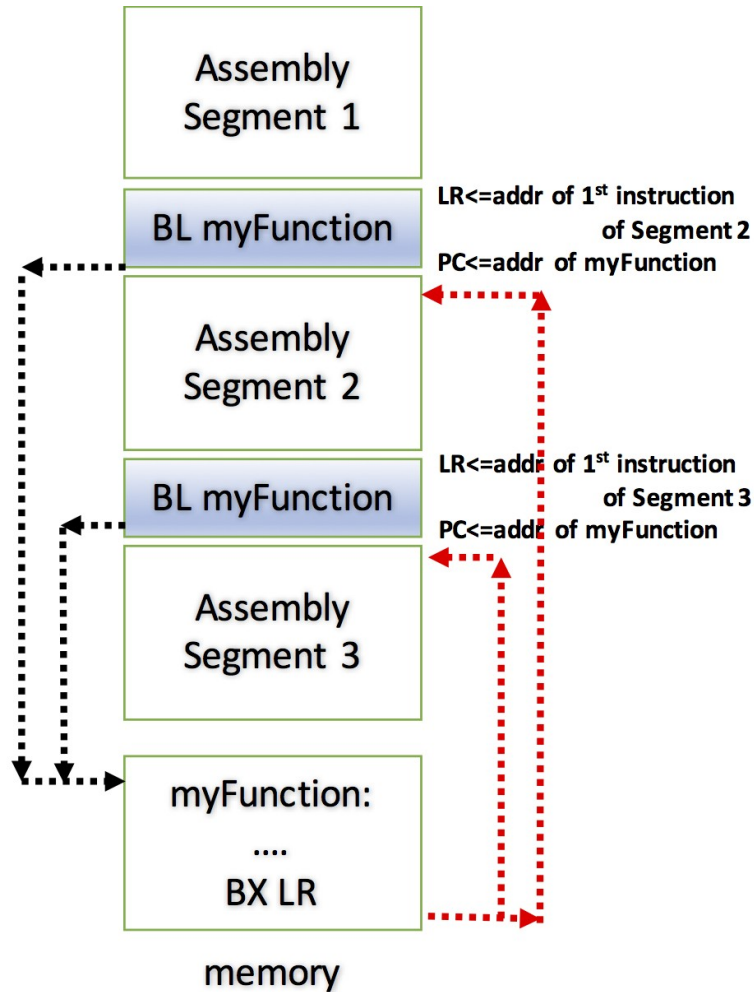


4.8 การเรียกใช้ฟังก์ชัน (Function Call)



4.8 การเรียกใช้ฟังก์ชัน (Function Call)

MyFunction ถูกเรียกใช้ซ้ำจากคำสั่ง BL myFunction สองประโยค แต่ LR จะเก็บแอดเดรสเพื่อรีเทิร์นกลับยังแต่ละตำแหน่งได้อย่างถูกต้อง โดย LR จะเก็บแอดเดรสคำสั่งถัดไป หลังจากทำงาน myFunction แล้วเสร็จ ด้วยประโยค BX LR



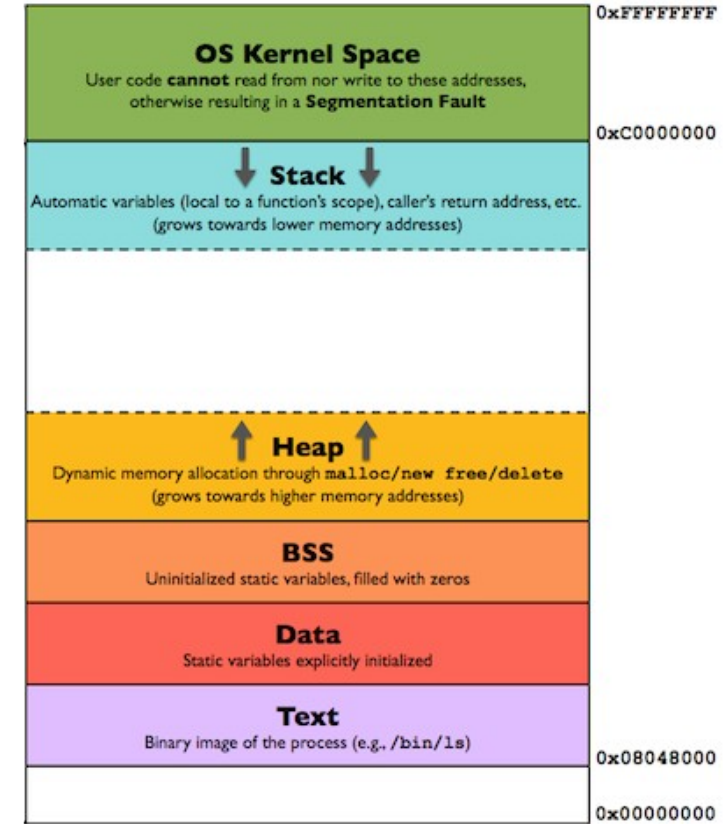
4.8 การเรียกใช้ฟังก์ชัน (Function Call)

- R14 เรียกว่า **ลิงค์รีจิสเตอร์** (Link Register: LR) คือ รีจิสเตอร์สำหรับเก็บแอดเดรสของคำสั่งที่ต้องการจะรีเทิร์นกลับ โดยรีจิสเตอร์นี้ทำงานคู่กับคำสั่ง BL (Branch and Link) และคำสั่ง BX LR
- R13 เรียกว่า **สแต็คพอยน์เตอร์** (Stack Pointer: SP) คือ รีจิสเตอร์สำหรับเก็บแอดเดรสหรือตำแหน่งยอด (Top) ของสแต็คเช็กเมนต์ ซึ่งเรียกสั้นๆ ว่า **สแต็ค** โดยรีจิสเตอร์นี้ทำงานคู่กับคำสั่งที่เกี่ยวข้องกับหน่วยความจำ ในหัวข้อที่ 4.5 ผู้อ่านสามารถทบทวนรายละเอียดเกี่ยวกับสแต็คในหัวข้อที่ 3.2.3 ที่ผ่านมา หมายเหตุ คำว่า สแต็ค ในตำราเล่มนี้หมายถึงสแต็คเช็กเมนต์ มิได้หมายถึงโครงสร้างข้อมูล (Data Structure) ชนิดหนึ่ง
- R4-R12 เป็นรีจิสเตอร์สำหรับใช้งานทั่วไป
- R0-R3 เป็นรีจิสเตอร์สำหรับใช้งานทั่วไป และใช้ส่งค่าพารามิเตอร์ (Parameter) ไปให้ฟังก์ชัน โดยรีจิสเตอร์เหล่านี้ทำงานร่วมกับคำสั่ง BL (Branch and Link)
- R0 เป็นรีจิสเตอร์สำหรับรีเทิร์นค่าข้อมูลจากฟังก์ชัน โดยทำงานร่วมกับคำสั่ง BX LR

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13(SP)
R14(LR)
R15(PC)

4.8 การเรียกใช้ฟังก์ชัน (Function Call)

PUSH {register list}	<p>สำเนาข้อมูล 32 บิตจากรีจิสเตอร์ที่ปรากฏในรายชื่อรีจิสเตอร์ไปวางบนยอดสแต็คชั่วคราวตามลำดับจากซ้ายไปขวา</p> <p>ปรับเปลี่ยนค่ารีจิสเตอร์ SP ให้สอดคล้องกับคำสั่ง</p>
POP {register list}	<p>สำเนาข้อมูล 32 บิตจากยอดสแต็คไปบรรจุในรีจิสเตอร์ที่ปรากฏในรายชื่อรีจิสเตอร์ตามลำดับจากขวาไปซ้าย</p> <p>ปรับเปลี่ยนค่ารีจิสเตอร์ SP ให้สอดคล้องกับคำสั่ง</p>



R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13(SP)
R14(LR)
R15(PC)

4.8 การเรียกใช้ฟังก์ชัน (Function Call)

PUSH {register list}	<p>สำเนาข้อมูล 32 บิตจากรีจิสเตอร์ที่ปรากฏในรายชื่อรีจิสเตอร์ไปวางบนยอดสแต็คชั่วคราวตามลำดับจากซ้ายไปขวา</p> <p>ปรับเปลี่ยนค่ารีจิสเตอร์ SP ให้สอดคล้องกับคำสั่ง</p>
POP {register list}	<p>สำเนาข้อมูล 32 บิตจากยอดสแต็คไปบรรจุในรีจิสเตอร์ที่ปรากฏในรายชื่อรีจิสเตอร์ตามลำดับจากขวาไปซ้าย</p> <p>ปรับเปลี่ยนค่ารีจิสเตอร์ SP ให้สอดคล้องกับคำสั่ง</p>

```

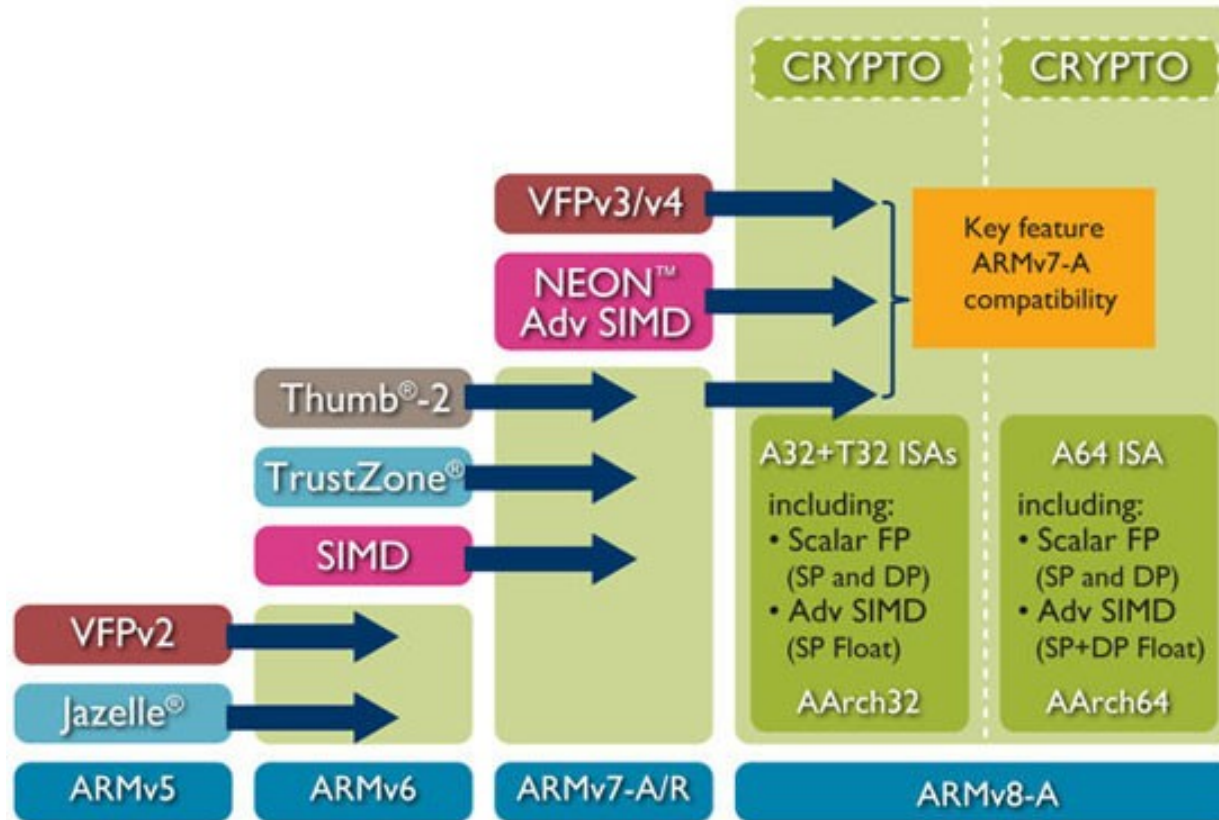
ion> 100
Disassembly
POPEQ    {r3, pc}
BLX      r3
POP      {r3, pc}
DCD      0x0001E008
DCD      0x00000000
main
PUSH     {r4, r5, r11, lr}
ADD      r11, sp, #0xc
SUB      sp, sp, #0x48
STR      r0, [r11, #-0x50]
STR      r1, [r11, #-0x54]
MOV      r3, #0
STR      r3, [r11, #-0x10]
LDR      r0, {pc}+0x17c ; 0xc2ec
BL       {pc}-0xab8 ; 0xb6b8
CMP      r0, #0
MOVEQ    r0, #1
BEQ      {pc}+0x164 ; 0xc2e4
BL       {pc}-0xa9c ; 0xb6e8
MOV      r0, #0

```

4.9 อุปกรณ์และวิวัฒนาการของชุดคำสั่ง ARM

	Devices Shipped (Million of Units)	2010 Devices	Chips/ Device	TAM 2010 Chips	2010 ARM	2010 Share	TAM 2015 Devices	Chips/ Unit	TAM 2015 Chips
Mobile	Smart Phone	280	2-5	1,200	1,100	90%	1,100	3-5	4,000
	Feature Phone	760	1-3	1,900	1,700	90%	650	2-3	2,000
	Low End Voice	570	1	570	540	95%	700	1-2	1,300
	Portable Media Players	150	1-3	300	220	70%	120	1-3	250
	Mobile Computing* (apps only)	230	1	230	25	10%	750	1	750
Non-Mobile	PCs & Servers (apps only)	220	1	220	0	0%	250	1	250
	Digital Camera	130	1-2	200	160	80%	150	1-2	250
	Digital TV & Set-top-box	350	1-2	450	160	35%	500	1-4	1,200
	Networking	670	1-2	750	185	25%	800	1-2	1,400
	Printers	120	1	120	75	65%	200	1	200
	Hard Disk & Solid State Drives	670	1	670	560	85%	1,100	1	1,100
	Automotive	1,800	1	1,800	180	10%	2,200	1	2,200
	Smart Card	5,400	1	5,400	330	6%	7,700	1	7,700
	Microcontrollers	5,800	1	5,800	560	10%	9,000	1	9,000
	Others **	1,700	1	1,800	270	15%	2,000	1	2,000
Total		19,000		22,000	6,100	28%	27,000		34,000

4.9 อุปกรณ์และวิวัฒนาการของชุดคำสั่ง ARM



สรุปท้ายบท

คำสั่งภาษาแอสเซมบลีของ ARM มีลักษณะเด่นเมื่อเปรียบเทียบกับภาษาอื่นๆ เช่น การมีคำสั่งเลื่อนบิตภายในคำสั่งคณิตศาสตร์ การตรวจสอบเงื่อนไขก่อนการปฏิบัติตามคำสั่งนั้น เป็นต้น ทำให้ชิพซีพียูที่ผลิตตามการออกแบบของ ARM บริโภคพลังงานน้อย จึงได้รับความนิยมในอุปกรณ์เคลื่อนที่มาเป็นระยะเวลาต่อเนื่อง ที่ผ่านมาสถาปัตยกรรมของคำสั่งสามารถแบ่งออกได้เป็น

- **สถาปัตยกรรมโหลด/สโตร์** (Load/Store Architecture) ARM ออกแบบคำสั่งภาษาแอสเซมบลีตามหลักการ RISC (Reduced Instruction Set Computer) ข้อมูลเพิ่มเติมที่ [wikipedia](#) และแตกต่างจากคำสั่งจากสถาปัตยกรรมอื่นๆ ดังนี้
- **สถาปัตยกรรม x86** (x86 Architecture) ในภาษาแอสเซมบลีของ Intel 80x86 หรือเรียกย่อๆ ว่า x86 ซึ่งบางคำสั่งใช้การอ่านค่าจากหน่วยความจำเพื่อประมวลผล ทำให้การประมวลซับซ้อนและล่าช้า ข้อมูลเพิ่มเติมที่ [wikipedia](#)
- **สแต็คแมชชีน** (Stack Machine) ได้แก่ ภาษาจาวาไบต์โค้ด (Java Bytecode) จัดเป็นคำสั่งภาษาแอสเซมบลีเมื่อแปลจากซอร์สโค้ดภาษา Java เทคโนโลยีที่น่าสนใจของ ARM เรียกว่า Jazelle สามารถรองรับการประมวลผลคำสั่ง Java Bytecode ด้วยฮาร์ดแวร์ทำให้ชิพบริโภคพลังงานต่ำโดยเฉพาะโทรศัพท์สมาร์โฟนที่ใช้ระบบปฏิบัติการ Android ซึ่งซอฟต์แวร์ส่วนใหญ่ใช้ภาษา Java พัฒนา รายละเอียดเพิ่มเติมใน [wikipedia](#)

References

- https://www.researchgate.net/figure/Block-Diagram-of-Micro-SD-card_fig6_306236972
- <https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/>
- <https://freedompenguin.com/articles/how-to/learning-the-linux-file-system>
- <https://www.techpowerup.com/174709/arm-launches-cortex-a50-series-the-worlds-most-energy-efficient-64-bit-processors>
- https://www.researchgate.net/figure/NVIDIA-Tegra-2-mobile-processor-11_fig1_221634532
- Harris, D. and S. Harris (2013). Digital Design and Computer Architecture (1st ed.). USA: Morgan Kauffman Publishing.
- <https://learn.adafruit.com/resizing-raspberry-pi-boot-partition/edit-partitions>

References

- https://en.wikipedia.org/wiki/Human%E2%80%93computer_interaction
- <https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/programmer-s-guide-for-armv8-a>
- https://xdevs.com/article/rpi3_oc/
- https://www.gsmarena.com/a_look_inside_the_new_proprietary_apple_a6_chipset-news-4859.php
- https://www.slideshare.net/kleinerperkins/2012-kpcb-internet-trends-yearend-update/25-Global_Smartphone_Tablet_Shipments_Exceeded
- <https://www.aliexpress.com/item/32329091078.html>
- <https://www.raspberrypi.org/forums/viewtopic.php?t=63750>
- <https://www.youtube.com/watch?v=2ciyXehUK-U>