01076114
องค์ประกอบและสถาปัตยกรรมคอมพิวเตอร์
Computer Organization and Architecture

ARM Instruction Set

Instruction Set

- ภาษาคอมพิวเตอร์ระดับสูงใดๆ เมื่อจะทำงาน จะต้องแปลงเป็นภาษาเครื่องเสียก่อนจึงจะทำงานได้
  - บางภาษา เช่น C แปลงเป็นภาษาเครื่องโดยตรง (เรียกว่า Compiler) ทำให้สามารถเรียกมารันได้เลย
  - บางภาษา เช่น Java จะแปลงเป็น Portable Byte Code เสียก่อน เมื่อจะรันจึงค่อยแปลงเป็นภาษาเครื่องอีกที (JIT Compiler) ทำให้สามารถนำโปรแกรม Java ไปรันในเครื่องต่างสถาปัตยกรรมกันได้
  - บางภาษา เช่น Python จะแปลงเมื่อมีการทำงานเท่านั้น (เรียกว่า Interpreter)

Instruction Set

- ในแต่ละสถาปัตยกรรมของ Processor จะมีการสร้างภาษาเครื่องเอาไว้จำนวนหนึ่ง เช่น อาจจะมี 100 คำสั่ง (Instruction) โดยเมื่อเรียกรวมกันก็จะเรียกว่า ชุดคำสั่ง (Instruction Set) ซึ่งจะแตกต่างกันไปในแต่ละ Processor
- เมื่อภาษาระดับสูงจะทำงาน ก็จะต้องแปลงเป็นภาษาเครื่อง (ภายใน 100 คำสั่ง) นี้เสียก่อน
- จากนั้นโครงสร้างทาง Hardware จึงจะทำหน้าที่ Execute คำสั่งเหล่านี้อีกที ภาษาเครื่องเหล่านี้จึงทำหน้าที่เป็นส่วนเชื่อมระหว่าง Software และ Hardware (HW/SW Interface) และเรียกส่วน Hardware นี้ว่า สถาปัตยกรรมชุดคำสั่ง (ISA : Instruction Set Architecture)
- โดยทั่วไปการสร้าง ISA มักจะมีเป้าหมาย 2 ประการ
  - ชุดคำสั่งควรจะทำให้ Hardware เรียบง่ายที่สุด เพื่อจะได้เร่งความเร็วได้ง่าย
  - ชุดคำสั่งควรมีเฉพาะคำสั่งพื้นฐาน เพื่อให้การ decode และ execute ทำได้ง่าย

Instruction Set

- ในโลกของ Processor จะมีสถาปัตยกรรมชุดคำสั่งอยู่ 2 แบบ
  - CISC (Complex Instruction Set Computer)
  - RISC (Reduce Instruction Set Computer)
  - ซึ่งจะกล่าวถึงต่อไป
- สำหรับวิชานี้จะอ้างอิง ISA ของ ARM ซึ่งเป็น ISA ที่มีการใช้งานกันมากที่สุดในโลก ตั้งแต่ปี 2005 ผลิต > 1,000 ล้านตัวต่อปี ยอดขายจนถึงปี 2017 ประมาณ 1 หมื่นล้านตัว

A basic ASM instruction

C code: a = b + c ;

Assembly code: (human-friendly machine instructions)

add a, b, c # a is the sum of b and c

Machine code: (hardware-friendly machine instructions)

00000010001100100000000100000

จงแปลง C code ต่อไปนี้ให้เป็น assembly code:

a = b + c + d + e;

Example

C code a = b + c + d + e;

เมื่อแปลงเป็นภาษาแอสเซมบลี จะได้ดังนี้:

```
add  a, b, c          add  a, b, c
add  a, a, d    or    add  f, d, e
add  a, a, e          add  a, a, f
```

- คำสั่งแอสเซมบลีจะเป็นคำสั่งง่ายๆ มีรูปแบบที่แน่นอน 1 บรรทัดมี 1 การกระทำ
- ใน 1 บรรทัดของภาษา C อาจแปลงเป็นแอสเซมบลีหลายบรรทัดก็ได้
- จากตัวอย่าง code ด้านซ้ายจะดีกว่าด้านขวา **เพราะไม่ต้องใช้ f**

Subtract Example

C code f = (g + h) – (i + j);

เมื่อแปลงเป็นภาษาแอสเซมบลี จะได้ดังนี้:

```
add  t0, g, h          add  f, g, h
add  t1,  i, j    or   sub  f, f, i
sub  f,   t0, t1       sub  f, f, j
```

- code ด้านซ้ายจะเป็น code ที่ compiler ส่วนใหญ่แปล ซึ่งเป็นการแปลตามรูปประโยค
- code ด้านขวาดูเหมือนจะดีกว่า เพราะใช้ตัวแปรน้อยกว่า แต่อาจให้ผลที่แตกต่าง หากเป็นกรณีของเลขทศนิยม (จะกล่าวถึงภายหลัง)

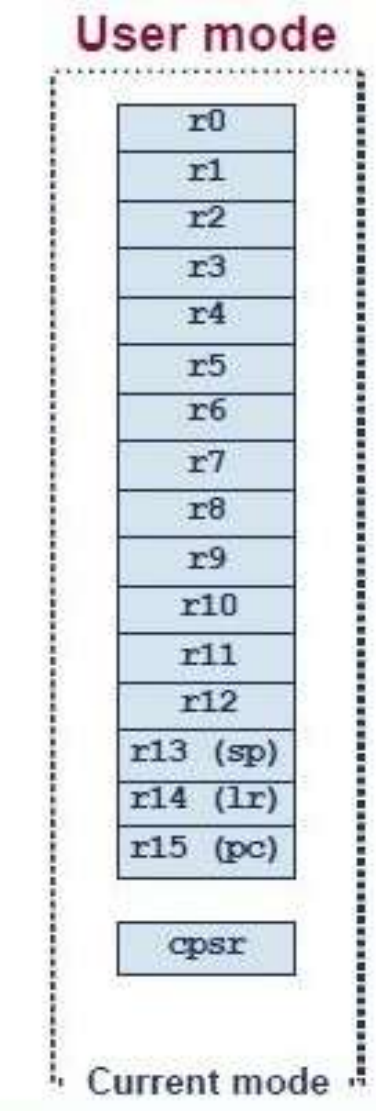
Registers

- ในภาษา C ตัวแปรจะอยู่ใน memory
- แต่ในระดับ Hardware การอ่านข้อมูลใน memory มีราคาแพง (คือใช้เวลาเยอะ) คำถาม : หน่วยความจำมี access time เท่าไร
- ดังนั้นหากอ้างถึงตัวแปรบ่อยๆ ถ้าตัวแปรนั้นมาอยู่ใน Processor ก็จะทำงานได้เร็วขึ้นมาก
- ตำแหน่งที่เก็บข้อมูลใน Processor จะเรียกว่า รีจิสเตอร์
- ใน Processor แบบ RISC (ซึ่ง ARM เป็นหนึ่งในนั้น) เน้นการออกแบบชุดคำสั่งที่เรียบง่าย ดังนั้นจึงกำหนดให้การทำงาน (เช่น add, sub) จะต้องกระทำกับรีจิสเตอร์เท่านั้น

Registers

- ARM ISA มีรีจิสเตอร์จำนวน 16 ตัว (x86 มี 8 ตัว)

  Why not more? Why not less?
- รีจิสเตอร์แต่ละตัวจะมีขนาด 32 บิต (Processor รุ่นใหม่ๆ ที่เป็น 64 บิต ก็จะมีรีจิสเตอร์ขนาด 64 บิตด้วย)
- การประมวลผลก็จะทำครั้งละ 32 บิต จึงเรียกขนาด 4 ไบต์ (32 บิต) นี้ว่า word
- ประกอบด้วย r0-r12 และ sp, lr, pc



Subtract Example

C code f = (g + h) – (i + j);

เมื่อแปลงเป็นภาษาแอสเซมบลี ARM จะได้ดังนี้:

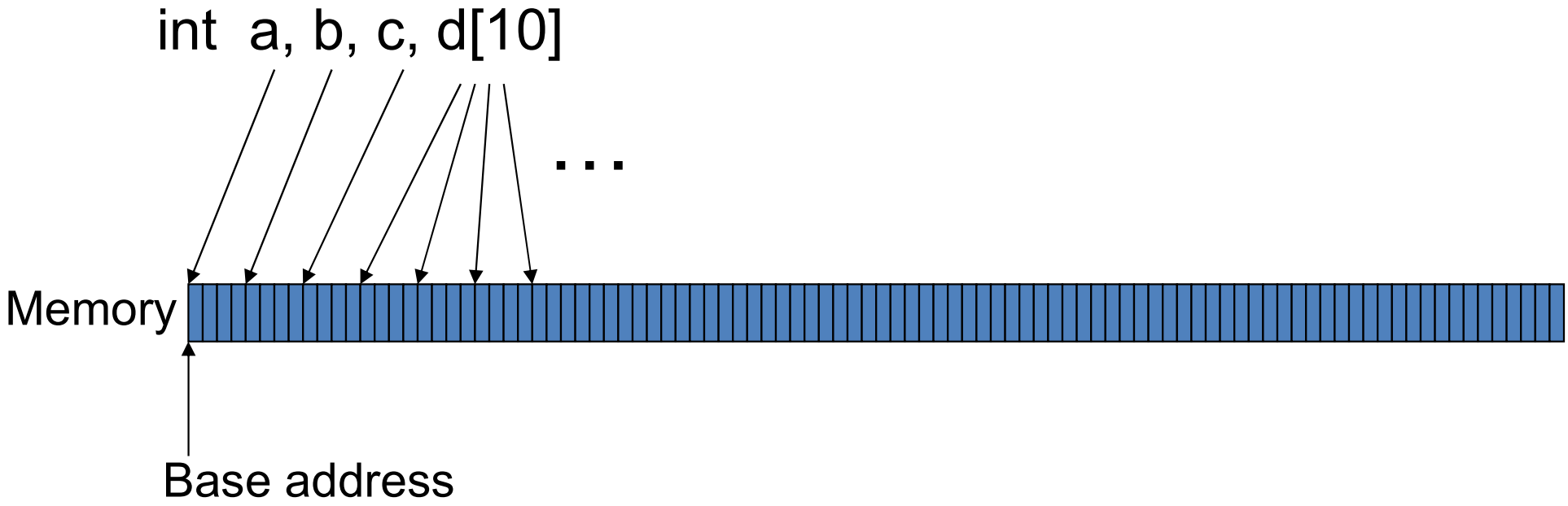
```
add  r5, r0, r1      ; register r5 contains g+h
add  r6, r2, r3      ; register r6 contains i+j
sub  r4, r5, r6      ; r4 gets r5 - r6 ((g + h) – (i + j)
```

Exercise

- จงเขียน ARM Assembly จากภาษา C ต่อไปนี้ โดยใช้จำนวนคำสั่งให้น้อยที่สุด (กรณีบวกค่าคงที่ ให้เขียนเป็น #2)
  - f = g + (j + 2)
  - f = f + g + h + i + j + 2

Memory Address

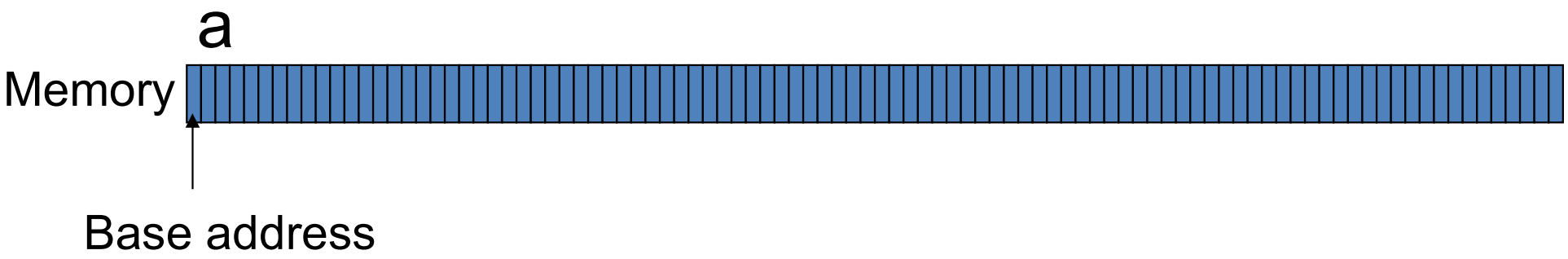
- ตัวแปร (และข้อมูล) ในภาษาระดับสูงจะเก็บใน memory
- สมมติว่ากำหนดตัวแปร int a,b,c,d[10]; จะเก็บในหน่วยความจำดังนี้ (ตัวแปรแต่ละตัวจะใช้เนื้อที่ 4 ไบต์



Data Transfer Instruction

- เนื่องจากรีจิสเตอร์มีขนาด 32 บิต จึงสามารถอ้างหน่วยความจำได้ 2^{32} $(2^{10} \times 2^{10} \times 2^{10} \times 2^{2}) = 4$ GB
- เนื่องจากข้อมูลหรือตัวแปรจะเก็บใน memory ดังนั้นก่อนที่จะทำคำสั่ง (add, sub) จะต้องโหลดข้อมูลจาก memory เข้ามาเก็บยัง register เสียก่อน
- เช่น ถ้าจะทำคำสั่ง a=a+b; ก็จะต้องเอา a ที่อยู่ใน memory เข้ามาเสียก่อน (สมมติว่า r3 ชี้ตำแหน่งที่เก็บ a) **[ ] หมายถึง อยู่ในหน่วยความจำ**

LDR r5, [r3, #0]



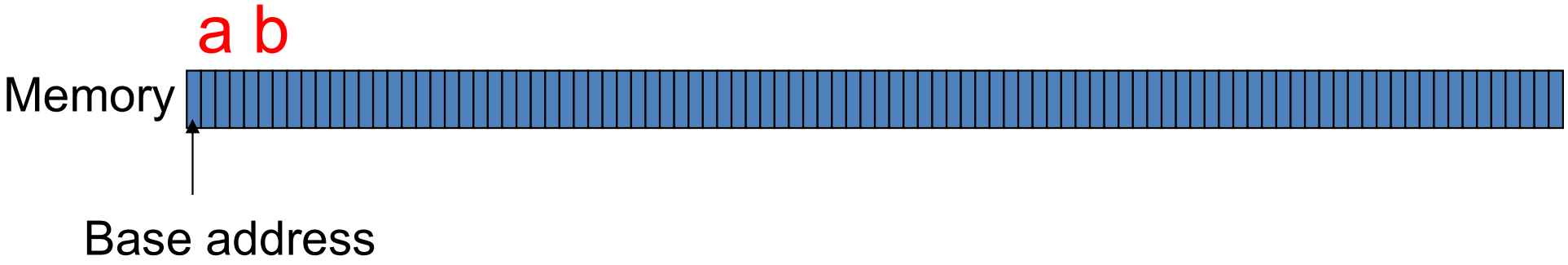
Example

C code a = a+b;

เมื่อแปลงเป็นภาษาแอสเซมบลี ARM จะได้ดังนี้:

```
LDR     r5, [r3,#0]     ; load a locate offset 0
LDR     r6, [r3,#4]     ; load b locate offset 4
ADD     r5, r5, r6      ; a = a+b
```

ในกรณีนี้ r3 จะเรียกว่า base register เนื่องจากใช้ในการอ้างตำแหน่ง



Example

- ในหน้าที่แล้ว ยังขาดการทำงานอยู่ 1 อย่าง คือ การนำผลการบวกที่เก็บในรีจิสเตอร์ r5 ไปเก็บในหน่วยความจำ ซึ่งจะเขียนเป็นคำสั่งดังนี้

STR r5, [r3, #0]

เมื่อเขียนเป็นภาษาแอสเซมบลี ARM ครบถ้วนจะได้ดังนี้:

```
LDR     r5, [r3,#0]      ; load a locate offset 0
LDR     r6, [r3,#4]      ; load b locate offset 4
ADD     r5, r5, r6       ; a = a+b
STR     r5, [r3,#0]      ; store a to offset 0
```

Immediate Operands

- จากคำสั่ง a=a+b ยังขาดการทำงานอยู่ 1 อย่าง เพราะเรายังสมมติว่า r3 ชี้ที่ a แต่ในการเขียนโปรแกรมจริงๆ เราจะต้องจัดการให้ r3 ไปชี้ที่ a
- สำหรับ operand ที่เป็นค่าคงที่จะเรียกว่า immediate operand
- ในการเขียนโปรแกรม จะมีหลายครั้งที่ต้องมีการกำหนดค่าคงที่ เพื่อนำมาประมวลผล เช่น a=a+5 จะใช้คำสั่ง

```
ADD  r5, r5, #5
```

- สำหรับการกำหนดค่าคงที่ให้กับรีจิสเตอร์จะใช้คำสั่ง MOV เช่น

```
MOV  r3, #0
```

Example

a = a+b; เมื่อเขียนเป็นภาษาแอสเซมบลี ARM ครบถ้วนจะได้ดังนี้:

```
MOV     r3, #0
LDR     r5, [r3,#0]       ; load a locate offset 0
LDR     r6, [r3,#4]       ; load b locate offset 4
ADD     r5, r5, r6        ; a = a+b
STR     r5, [r3,#0]       ; store a to offset 0
```

Exercise

- ให้เขียนภาษา assembly ของ arm สำหรับคำสั่ง a = b+d[5];

```
MOV     r6, #0          ; base
LDR     r1, [r6, #4]    ; b
LDR     r5, [r6, #32]   ; d[5]
ADD     r1, r1, r5
STR     r1, [r6, #0]    ; a
```

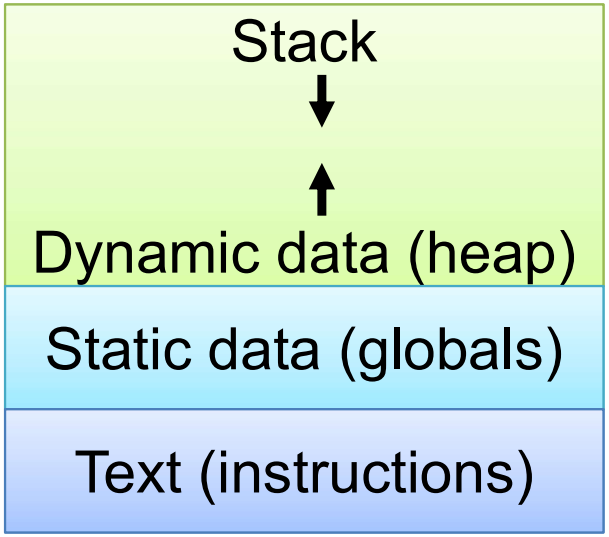
Exercise

- กำหนดให้ตัวแปร f, g, h, i, j เก็บค่าโดยรีจิสเตอร์ r0, r1, r2, r3 และ r4 ตามลำดับ และ base address ของ A และ B ชี้โดยรีจิสเตอร์ r6 และ r7 จงเขียนโปรแกรม assembly ของ ARM สำหรับ

```
f = g – A[B[4]];
```

Memory Organization

- ในโปรแกรมแต่ละโปรแกรมจะมีการจัดโครงสร้างของหน่วยความจำตามรูป
- ส่วนของโปรแกรมและข้อมูลจะอยู่ส่วนต้น
- ถัดขึ้นไปจะเป็น dynamic memory เช่น object หรือการจองหน่วยความจำ malloc() ซึ่งจะเรียกรวมๆ กันว่า heap
- ท้ายสุดจะเป็น stack



Recap – Numeric Representations

- Decimal $35_{10} = 3 \times 10^1 + 5 \times 10^0$
- Binary $00100011_2 = 1 \times 2^5 + 1 \times 2^1 + 1 \times 2^0$
- Hexadecimal (compact representation)

  $0x\ 23$ or $23_{hex} = 2 \times 16^1 + 3 \times 16^0$

0-15 (decimal) → 0-9, a-f (hex)

Dec	Binary	Hex	Dec	Binary	Hex	Dec	Binary	Hex	Dec	Binary	Hex
0	0000	00	4	0100	04	8	1000	08	12	1100	0c
1	0001	01	5	0101	05	9	1001	09	13	1101	0d
2	0010	02	6	0110	06	10	1010	0a	14	1110	0e
3	0011	03	7	0111	07	11	1011	0b	15	1111	0f

Instruction Formats

- ต่อไปจะมาดูว่าคำสั่งแอสเซมบลี เมื่อแปลงเป็นภาษาเครื่องมีรูปแบบอย่างไร

ADD r5, r1, r2

- แต่ละช่องจะเรียกว่า field
  - ฟิลด์ที่ 4 (4) หมายถึง operation ในที่นี้คือ ADD
  - ฟิลด์ที่ 6 (1) หมายถึง รีจิสเตอร์ ที่เป็น source operand ตัวแรก ในที่นี้ คือ r1
  - ฟิลด์ที่ 8 (2) หมายถึง รีจิสเตอร์ ที่เป็น source operand ตัวที่ 2 ในที่นี้ คือ r2
  - ฟิลด์ที่ 7 (5) หมายถึง รีจิสเตอร์ ที่เป็น destination operand ในที่นี้ คือ r5

Instruction Formats

- คำสั่งข้างต้นเขียนเป็นเลขฐาน 2 ได้ดังนี้

1110	00	0	0100	0	0001	0101	000000000010
4 bits	2 bits	1 bits	4 bits	1 bits	4 bits	4 bits	12 bits

- โครงสร้างตามรูป จะเรียกว่า Instruction Format ซึ่งประกอบด้วยฟิลด์ย่อยๆ ที่เมื่อรวมกันแล้วจะได้ 32 บิต
- โดยสามารถจะเขียนเป็นเลขฐาน 16 เพื่อให้ดูง่ายขึ้นได้เป็น

E081 5002

Instruction Formats

- รูปแบบคำสั่งของ ARM (กรณี F=0) มีโครงสร้างดังนี้

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

  - Opcode : คำสั่ง
  - Rd : รีจิสเตอร์ที่เป็น destination operand ของการทำงาน
  - Rn : รีจิสเตอร์ source operand ตัวที่ 1
  - Operand2 : source operand ตัวที่ 2
  - I : Immediate ถ้าบิตนี้มีค่า = 0 ; operand ตัวที่ 2 เป็นรีจิสเตอร์
    ถ้าบิตนี้มีค่า = 1 ; operand ตัวที่ 2 เป็นค่าคงที่ 12 บิต
  - S, Cond จะกล่าวถึงภายหลัง
  - F : รูปแบบ Format คำสั่งของ ARM (F=0 : Data Processing Format)

Example

- ให้แปลงคำสั่งต่อไปนี้ให้เป็นภาษาเครื่อง (ใช้ cond=14, F=0, S=0)

ADD r3, r3, #4

- Opcode = 4 (คำสั่ง ADD)
- Rn = 3 (r3 : first source operand)
- Rd = 3 (r3 : destination operand)
- I = 1 : Operand2 = Constant
- Operand2 = 4 (4 : Constant)

Cond	F	I	Opcode	S	Rn	Rd	Operand2
4 bits	2 bits	1 bits	4 bits	1 bits	4 bits	4 bits	12 bits
14	0	1	4	0	3	3	4

Example

- ให้แปลงคำสั่งต่อไปนี้ให้เป็นภาษาเครื่อง

LDR r5, [r3, #32]

- คำสั่งนี้จะใช้รูปแบบที่ต่างไป (F=1) อยู่ในกลุ่ม data transfer
- รูปแบบคำสั่งนี้จะมีเพียง 6 ฟิลด์ โดยตัดฟิลด์ I และ S ออก
- สำหรับคำสั่ง LDR จะมี opcode = 24 ดังนั้นคำสั่งนี้จะแปลงเป็นภาษาเครื่องได้ดังนี้

Cond	F	Opcode	Rn	Rd	Offset12
4 bits	2 bits	6 bits	4 bits	4 bits	12 bits
14	1	24	3	5	32

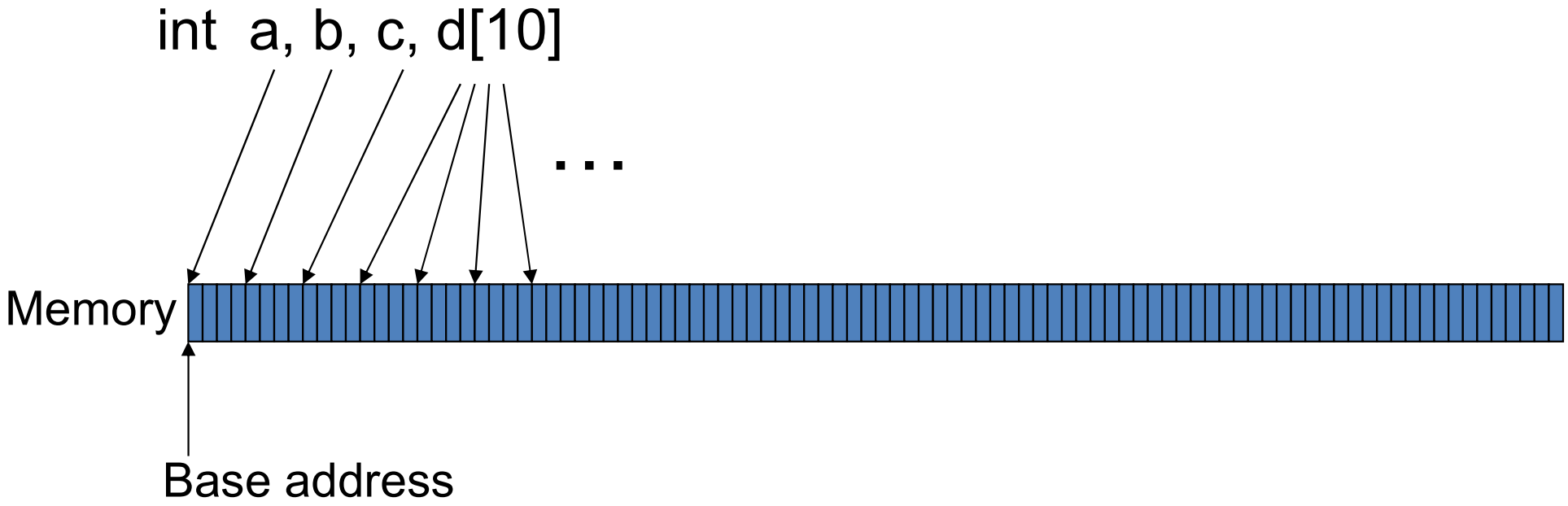
Instruction Formats

- ถึงแม้จะมี format ของคำสั่งหลายรูปแบบ แต่ก็ยังคงความคล้ายคลึงกันของแต่ละรูปแบบ ทั้งนี้เพื่อให้ Hardware ที่ใช้ถอดรหัสคำสั่งซับซ้อนน้อยที่สุด
- เช่น 2 ฟิลด์แรกกับ 3 ฟิลด์หลังจะเหมือนกัน
- สรุปรูปแบบของ Instruction Format ของคำสั่งประเภท data processing (DTP และ data transfer (DT) ที่ผ่านมาได้ดังนี้

Instruction	Format	Cond	F	I	Opcode	S	Rn	Rd	Operand2
ADD	DP	14	0	0	4	0	reg	reg	reg
SUB	DP	14	0	0	2	0	reg	reg	reg
ADD (Immediate)	DP	14	0	1	4	0	reg	reg	constant
LDR	DT	14	1	n.a.	24	n.a.	reg	reg	address
STR	DT	14	1	n.a.	25	n.a.	reg	reg	address

Exercise

- จากข้อมูลในหน่วยความจำตามรูป
- จงเขียนภาษาแอสเซมบลีของ d[5] = c + [d5];
- และแปลงเป็นภาษาเครื่อง กำหนดให้ r3 ชี้ที่ 0



Exercise

- d[5] = c + d[5]; เขียนเป็นภาษา assembly ได้ดังนี้

```
LDR     r5, [r3,#8]      ; load c locate offset 8
LDR     r2, [r3,#32]     ; load d[5] locate offset 32
ADD     r5, r2, r5       ; d[5] = c + d[5]
STR     r5, [r3,#8]      ; store result to offset 8
```

- สร้างเป็นภาษาเครื่องดังนี้

Instruction	Cond	F	I	Opcode	S	Rn	Rd	Operand2
LDR r5, [r3,#8]	1110	01		11000		0011	0101	0000 0000 1000
LDR r2, [r3,#32]	1110	01		11000		0011	0010	0000 0010 0000
ADD r5, r2, r5	1110	00	0	0100	0	0010	0101	0000 0000 0101
STR r5, [r3,#8]	1110	01		11001		0011	0101	0000 0000 1000

Exercise

- จาก bit pattern นี้ เป็นคำสั่งอะไร

Cond	F	I	Opcode	S	Rn	Rd	Operand2
14	0	0	4	0	0	1	2

1. ADD R0, R1, R2
2. ADD R1, R0, R2
3. ADD R2, R1, R0
4. ADD R2, R0, R1

Exercise

- จาก Bit Pattern ต่อไปนี้ แทนคำสั่งใด

  1010 1110 0000 1011 0000 0000 0000 0100

  1000 1101 0000 1000 0000 0000 0100 0000

Exercise

- จากคำสั่งต่อไปนี้ ให้เขียนเป็น bits of opcode เป็นเลขฐานสิบหก
  - ADD r0, r0, r5
  - LDR r1, [r3, #4]

Status Register

- ในแต่ละ Processor จะมีการสร้างรีจิสเตอร์พิเศษขึ้นตัวหนึ่ง ใช้สำหรับเก็บสถานะที่เกิดขึ้นจากการทำงาน เรียกว่า status register (ในระบบอื่นๆ อาจใช้ชื่อต่างกันออกไป)

31	30	29	28	27...8	7	6	5	4	3	2	1	0
N	Z	C	V		I	F	T	MODE				

  - N : Negative Flag เมื่อเป็นเลขลบ (2's compliment) ตรงกับบิตที่ 31
  - Z : Zero Flag จะเป็น 1 เมื่อผลลัพธ์ของการทำงานเป็น 0 เช่น SUBS R1,R1,R1
  - C : Carry Flag จะเป็น 1 เมื่อผลลัพธ์ของการทำงานมีการทดเกิดขึ้น
  - V : Overflow Flag จะเป็น 1 เมื่อมีการทดจากบิต 30->31 ถ้าเป็น signed จะแสดงว่าผลลัพธ์เป็นเลขลบ

Arithmetic

- คำสั่งอื่นๆ ในกลุ่มการคำนวณ ได้แก่ [op Rd, Rn, Op2]
  - ADC ให้รวม Carry bit เข้ามาในการบวกด้วย
  - SBC ลบ Op2 จาก Rn ถ้า Carry Flag = 0 ให้ลดผลลัพธ์ลง 1
  - RSB (Reverse SuBtract) ให้ลบ Rn ออกจาก Op2 แล้วไปเก็บที่ Rd
  - RSC (Reverse SuBtract with Carry) ให้ลบ Rn ออกจาก Op2 ถ้า Carry Flag = 0 ให้ลดผลลัพธ์ลง 1
  - MUL Rn * Op2 แล้วเก็บผลลัพธ์ใน Rd
  - MLA ดูตัวอย่างหน้าต่อไป

Arithmetic

```
ADC r1, r2, r3          ; r1=r2+r3+ C(arry Flag)
SBC r1, r2, r3          ; r1=r2-r3 +C -1
RSB r1, r2, r3          ; r1= r3-r2;
RSC r1, r2, r3          ; r1=r3-r2 +C -1
MUL r0, r1, r2          ; r0 = r1 * r2
MLA r0, r1, r2, r3      ; r0 = (r1 * r2) + r3
[U|S]MULL r4, r5, r2, r3      ; r5:r4 = r2 * r3
[U|S]MLAL r4, r5, r2, r3      ; r5:r4 = (r2 * r3) + r5:r4
```

Data Transfer

- คำสั่งอื่นๆ ในกลุ่ม Data Transfer
  - LDRH load register half word
    โหลดข้อมูล 16 บิต
  - STRH store register half word
    เก็บข้อมูล 16 บิต
  - LDRB load register byte
  - STRB store register byte
  - SWP สลับข้อมูลระหว่างรีจิสเตอร์กับ memory

Logical Operations

Logical ops	C operators	Java operators	ARM instr
Shift Left	<<	<<	LSL
Shift Right	>>	>>>	LSR
Bit-by-bit AND	&	&	AND
Bit-by-bit OR	|	|	ORR
Bit-by-bit NOT	~	~	MVN

Logical Operations

- กำหนดให้

r2 = 0000 0000 0000 0000 0000 1101 1100 0000

r1 = 0000 0000 0000 0000 0011 1100 0000 0000

หลังจากทำคำสั่ง AND r5,r1,r2 แล้ว r5 มีค่าเท่าใด?

r5 = 0000 0000 0000 0000 0000 1100 0000 0000

หลังจากทำคำสั่ง ORR r5,r1,r2 แล้ว r5 มีค่าเท่าใด?

r5 = 0000 0000 0000 0000 0011 1101 1100 0000

หลังจากทำคำสั่ง MVN r5,r1 (move not) แล้ว r5 มีค่าเท่าใด?

r5 = 1111 1111 1111 1111 1100 0011 1111 1111

Logical Operations

- สำหรับ operation shift นั้น หาก shift left จะคล้ายกับการคูณ 2 และ shift right จะคล้ายกับการหาร 2
- คำสั่ง LSL (shift left) และ LSR (shift right) สำหรับ ARM แล้ว สามารถใช้เป็น **การทำงานที่ 2** ของคำสั่ง data processing ได้ด้วย

```
ADD r5, r1, r2, LSL #2    ; r5 = r1 + (r2 << 2)
```

- คำสั่งข้างต้นหมายถึง นำ r2 ไป shift left จำนวน 2 ครั้ง จากนั้นจึงบวกกับ r1 ก่อนจะนำไปเก็บที่ r5

```
MOV r6, r5 LSR r3         ; r6 = r5 >> r3
```

- จะหมายถึง นำ r5 ไป shift right ตามตัวเลขใน r3 แล้วนำไปเก็บที่ r6

Logical Operations

- สำหรับ Instruction Format ของคำสั่ง Logical Operation จะรวมอยู่ใน Operand 12 บิต โดยมีโครงสร้างดังนี้ (บิตที่ 25 (I) = 0)

11-8	7	6	5	4	3	2	1	0
Shitf_imm		Shift		0	Rm			
Rs	0	Shift		1	Rm			

- บิตที่ 4 ถ้าเป็น 0 จะเป็น Shift ตามจำนวนตัวเลข (Immediate) ถ้าเป็น 1 หมายถึงใช้ข้อมูลในรีจิสเตอร์ในการ shift
- ฟิลด์ shift ถ้าเป็น 0 หมายถึง shift left ถ้าเป็น 1 หมายถึง shift right
- ฟิลด์ Rm คือ second source operand
- ฟิลด์ Rs คือ register shift length
- ถ้าเป็นคำสั่งปกติ แบบไม่มีการ shift บิตที่ 11-4 จะเป็น 0 คือ shift 0 บิต

Logical Operations

- คำสั่งอื่นๆ ใน logical operation

r0: 01101001

r1: 11000111

```
ORR r3, r0,r1      ; r3: 11101111
AND r3,r0,r1       ; r3: 01000001
EOR r3,r0,r1       ; r3: 10101110 # Exclusive OR
BIC r3, r0, r1     ; r3: 00101000 # Bit Clear r3 = r0 & (!r1)
```

Making Decision

- ในภาษาระดับสูง ในการตัดสินใจ จะใช้คำสั่ง if แต่ในภาษา assembly จะทำเงื่อนไขดังนี้

  if (i=j) f = g+h; else f = g-h;

- กำหนดให้ r3=i, r4=j, f=r0, g=r1, h=r2;

```
        CMP     r3,r4
        BNE     Else            ; Branch not equal
        ADD     r0,r1,r2        ; f = g + h
        B       Exit            ; Exit if
Else:   SUB     r0,r1,r2        ; f = g – h
Exit:
```

Loop

- สำหรับ Loop เมื่อพิจารณาแล้ว ก็คือการทำเงื่อนไขอย่างหนึ่ง

```
while (save[i] == k)
    i += 1;
```

- กำหนดให้ i=r3, k=r5 โดย r6 เป็น base register ชี้ที่ตำแหน่งเริ่มต้นของ save
- r12 เป็น temp = i*4 + r6 ดังนั้น r0 จะเก็บข้อมูลใน save[i]

```
Loop:   ADD     r12, r6, r3, LSL #2     ; r12 = addr of save[i]
        LDR     r0, [r12, #0]           ; r0 = save[i]
        CMP     r0, r5
        BNE     Exit                    ; goto Exit if save[i] != k
        ADD     r3, r3, #1              ; i = i+1
        B       Loop                    ; to while loop
Exit:
```

Exercise

- เขียนคำสั่งภาษา assembly ที่เทียบเท่า

```
for (i = 0 ;  i < 10 ; i++)
    sum += data[i];
```

Branch Instruction

- นอกเหนือจาก BNE แล้ว คำสั่ง Branch ยังมีคำสั่งอื่นๆ อีก ดังนี้

Cond	Meaning	Cond	Meaning
0	EQ (EQual)	8	HI (unsigned HIgher)
1	NE (Not Equal)	9	LS (unsigned Lower or Same)
2	HS (unsigned Higher or Same)	10	GE (singed Greater than or Equal)
3	LO (unsigned LOwer)	11	LT (signed Less Than)
4	MI (MInus, <0)	12	GT (signed Greater Than)
5	PL (PLus, >=0)	13	LE (signed Less Than or Equal)
6	VS (oVerflow Set, overflow)	14	AL (Always)
7	VC (oVerflow Clear, no overflow)	15	NV (reserved)

- คำสั่งกลุ่ม BHS, BLO, BHI, BLS จะใช้กับข้อมูลแบบ Unsigned โดยจะพิจารณาจาก flag Z และ C
- คำสั่งกลุ่ม BGE, BLT, BGT, BLE จะใช้กับข้อมูลแบบ Signed โดยจะพิจารณาจาก flag Z และ V

Branch Instruction

- กำหนดให้ r0 = FFFF FFFFh, r1 = 0000 0001h
- คำสั่งต่อไปนี้

```
CMP r0, r1
```

- คำสั่ง Branch ใดจะให้ผลอย่างไร

```
BLO L1      ; unsigned branch
BLT L2      ; signed branch
```

Branch Instruction Format

- สำหรับ Instruction Format ของคำสั่ง Branch จะค่อนข้างต่างจากคำสั่งที่ผ่านๆ มา ดังนั้นจึงต้องใช้คำสั่งรูปแบบที่ 3 ดังนี้

Cond	Opcode		Address
4 bits	4 bits		24 bits
	101	L	

- ประเภทของเงื่อนไขจะอยู่ในฟิลด์ Cond
- การกระโดดเป็นแบบ relative คืออ้างอิงจากตำแหน่งของคำสั่งถัดไปจากปัจจุบัน
- สำหรับ 24 บิตจะนับเป็น word ดังนั้นจะสามารถกระโดดได้ไกลสุด = ± 64 MB (2^24 * 4)

Cond Field

- จากที่กล่าวไปว่าเงื่อนไขจะอยู่ในฟิลด์ Cond
- และถ้าสังเกต จะเห็นว่าในทุกคำสั่งจะมีฟิลด์ Cond อยู่ด้วย แปลว่า ทุกคำสั่งสามารถทำเป็นเงื่อนไขได้ด้วย (แจ่มเลย!) ดังนั้นโปรแกรมนี้

```
        CMP     r3,r4
        BNE     Else            ; Branch not equal
        ADD     r0,r1,r2        ; f = g + h
        B       Exit            ; Exit if
Else:   SUB     r0,r1,r2        ; f = g – h
Exit:
```

- สามารถเขียนใหม่ได้ดังนี้

```
        CMP     r3,r4
        ADDEQ   r0,r1,r2        ; f = g + h
        SUBNE   r0,r1,r2        ; f = g – h
```

For your attention