Embedded System (ES)

Lecturer: Dr. Cheng-Kai Lu
Phone: (02)7749-3554
Office: TD302/BAIR Lab
Email: cklu@ntnu.edu.tw

國立臺灣師範大學
NATIONAL TAIWAN NORMAL UNIVERSITY

Outline

In this lecture, we will cover:

- Review on the **important points** which were covered in the previous lectures
- Major Classes of Assembly Instructions
- ARM Assembly Programming: Translating C Control Statements and Function Calls
- Translating C Control Statements and Function Calls, Loops, Interrupt Processing
- In-Class Lab

Recap: OS Scheduling Techniques

- Round-Robin Scheduling
- Pre-emptive Scheduling:

  - Rate Monotonic Scheduling (RMS): This is static priority-driven preemptive scheduling

$$U_{CPU\_RMS} = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq U_{CPU(bound)} = n(\sqrt[n]{2} - 1)$$

  - Earliest Deadline First (EDF): This is dynamic priority-driven preemptive scheduling

Schedulable or not?

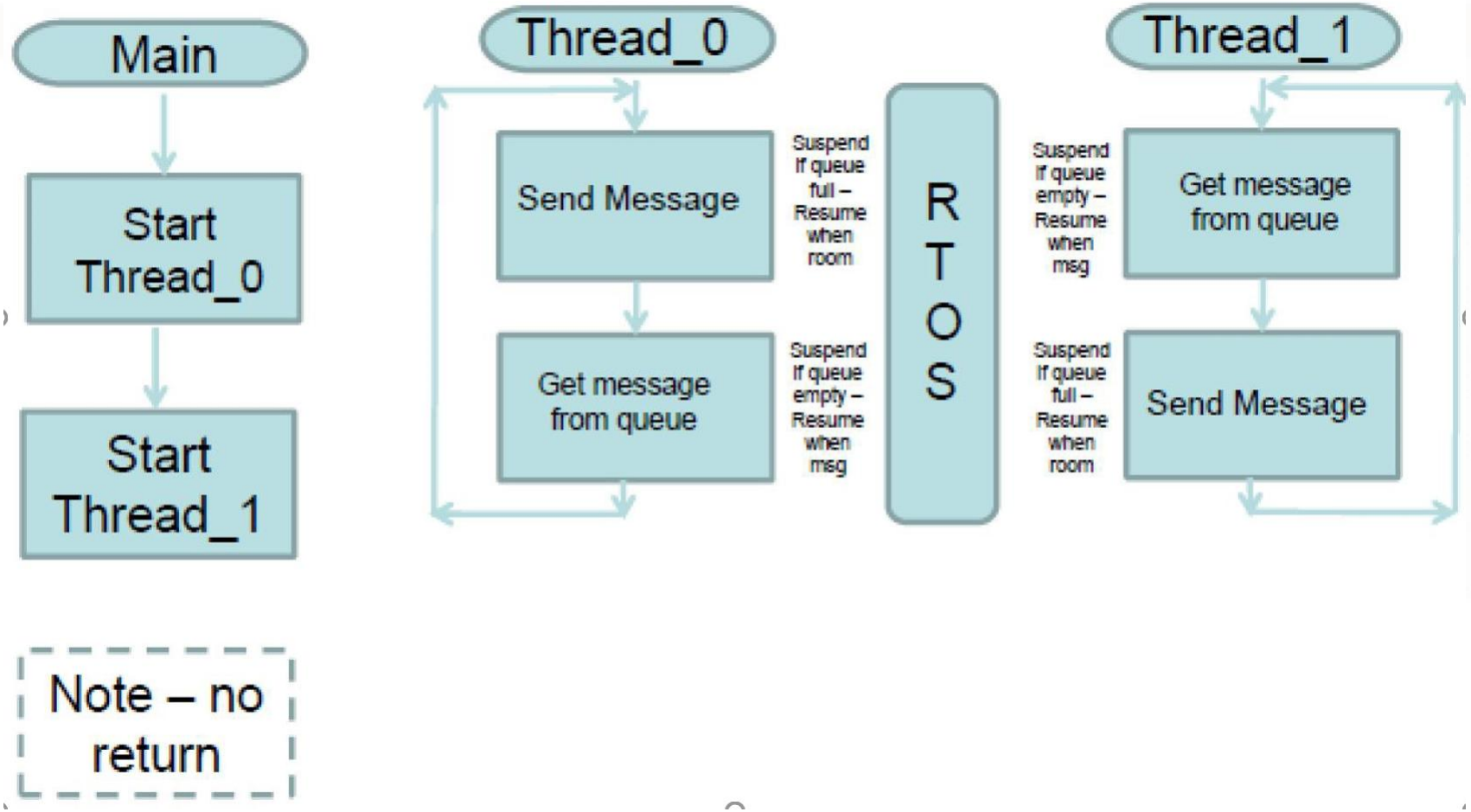
$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1,$$

- Cooperative (non-preemptive) Scheduling Techniques

Recap: Protecting Shared Resources and Synchronizing Threads

- Queue: pass messages (data) between threads
- Lock: allows only one thread to enter the “locked” section of code
- Mutex (MUTual EXclusion): Like a lock, but system wide (shared by multiple processes)
- Semaphore: allows multiple threads to enter a critical section of code

Recap: RTOS


Main
Start Thread_0
Start Thread_1
Note – no return
Thread_0
Send Message
Get message from queue
Suspend if queue full – Resume when room
Suspend if queue empty – Resume when msg
RTOS
Thread_1
Suspend if queue empty – Resume when msg
Get message from queue
Suspend if queue full – Resume when room
Send Message

Exercise 0

```
Semaphore S = 2;  // Counting semaphore initialized to 2
int shared_var = 1;
void TaskA() {
   P(S);
   shared_var += 2;
   P(S);
   shared_var *= 3;
   V(S);
   V(S);
}
void TaskB() {
   P(S);
   shared_var += 5;
   V(S);
}
void TaskC() {
   P(S);
   shared_var *= 2;
   V(S);
}
void main() {
   Start(TaskA);
   Start(TaskB);
   Start(TaskC);
}
```

- What are the possible final values of shared_var after all tasks complete?
- Which task(s) could block temporarily, and why?
- Total number of context switches caused by blocking

Assembly Instructions

Primitive Types and Sizes

Name	Number of Bytes sizeof()	Range
long	4	-2,147,483,648 to 2,147,483,647
signed long	4	-2,147,483,648 to 2,147,483,647
unsigned long	4	0 to 4,294,967,295
long long	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4	±1.175e-38 to ±3.402e38
double	Varies by platform	
double (on TM4C123)	8	±2.3E-308 to ±1.7E+308

- Primitive types in C: char, short, int, long, float, double default modifier on primitive types is **signed** (not unsigned)
- Note: char does not have a standard default, it depends on **Compiler settings**

Major Classes of Assembly Instructions

- Data Movement
  - Move data between registers
  - Move data in & out of Memory
  - Different addressing modes
- Logic & Arithmetic
  - Addition, subtraction, etc.
  - AND, OR, bit shift, etc.
- Control Flow
  - Control which sections of code should be executed (e.g. In C “IF”, “CASE”, “WHILE”, etc.
  - Function Calls

Instructions to move data: Brief Summary

- Move
  - MOV Rd, Rt Move Between Registers: *Rd ← Rt*
  - MOVW Rd, #Imm16 Constant to Register: *Rd ← #Imm16*
  - MOVT Rd, #Imm16 Constant to upper Register: *Rd ← #Imm16*
- Load (**L**oa**D** to **R**egister)
  - LDR Rd, [Rn, #Offset] Load 32-bit: *Rd ← [Rn + #Offset]*
  - LDRB Rd, [Rn, #Offset] Load 8-bit: *Rd ← [Rn + #Offset]*
  - LDRH Rd, [Rn, #Offset] Load 16-bit: *Rd ← [Rn + #Offset]*
- Store (**ST**ore from **R**egister)
  - STR Rt, [Rn, #Offset] Load 32-bit: *[Rn + #Offset] ← Rt*
  - STRB Rt, [Rn, #Offset] Load 8-bit: *[Rn + #Offset] ← Rt*
  - STRH Rt, [Rn, #Offset] Load 16-bit: *[Rn + #Offset] ← Rt*
  - STRD Rt, Rt2, [Rn, #Offset] Load 64-bit: *[Rn + #Offset]← Rt*
    [Rn + #Offset+4]← Rt2

Arithmetic Instruction

Brief overview of arithmetic instructions

Addition: ADD, ADC, ADDW

Subtraction: SUB, SBC

Logic: AND, ORR, EOR

Shift: LSL, LSR, ASR, ROR

Compliments: NEG

Multiplication: MUL, SMULL, UMULL

ADD

ADD: Add two registers without a carry flag

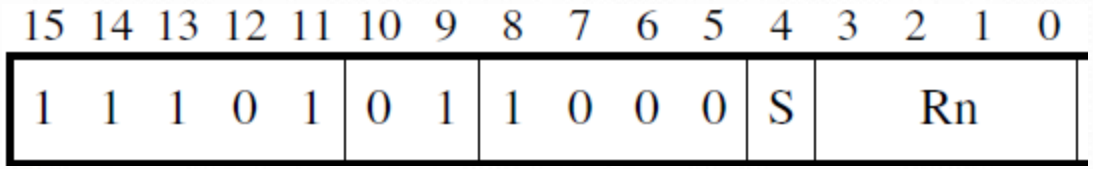
Syntax: ADD{S} Rd, Rn, Rm {,shift}

Operands: 0 ≤ d ≤ 15, 0 ≤ n ≤ 15, 0 ≤ m ≤ 15

Operations: Rd ← Rn + Rm (omitting shift), PC ← PC+4

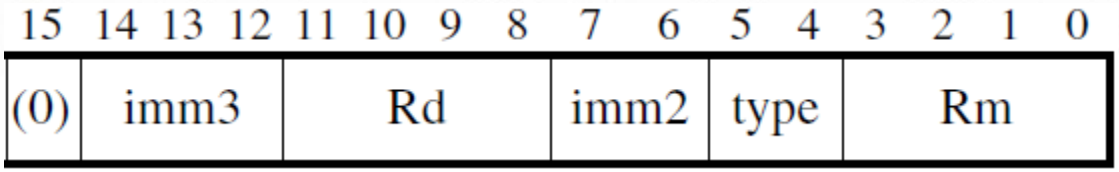
Binary Format:

`shift` omitted



imm3:imm2= 0

type = 0



Note: if the `shift` is not omitted, then Rm is shifted based on the encodings given in ARM Instruction Set

ADD with carry

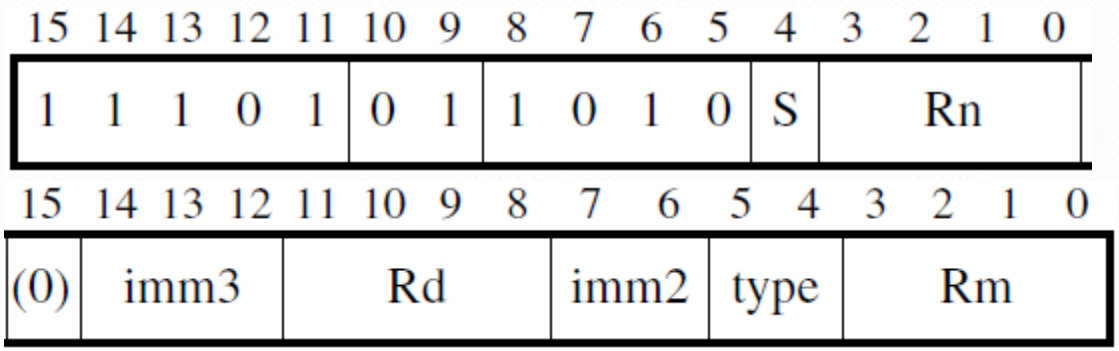
ADC: Add two registers with carry flag (C)

Syntax: ADC{S} Rd, Rn, Rm {,shift}

Operands: 0 ≤ d ≤ 15, 0 ≤ n ≤ 15, 0 ≤ m ≤ 15

Operations: Rd ← Rn + Rm + C (omitting shift), PC ← PC+4

Binary Format:



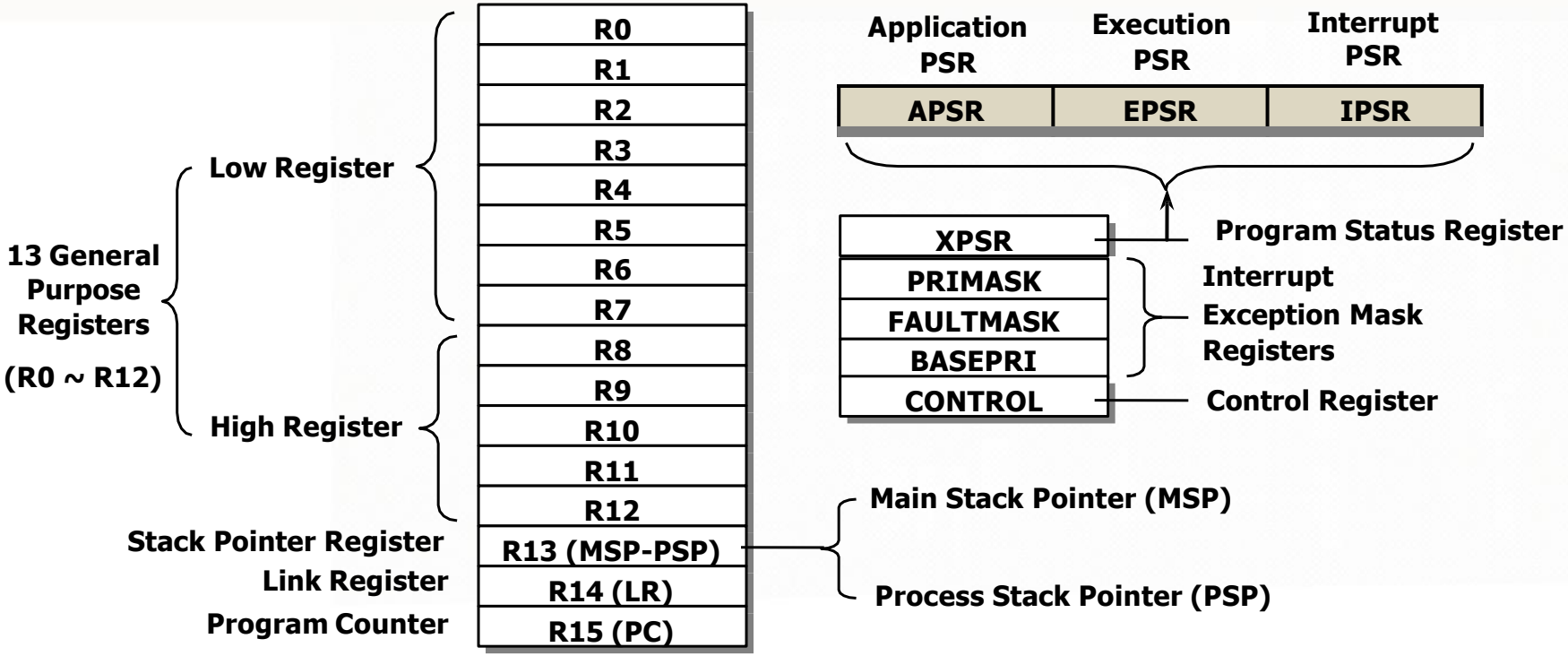
`shift` omitted

imm3:imm2= 0

type = 0

Note: if the `shift` is not omitted, then Rm is shifted based on the encodings given in ARM Instruction Set

General Purpose Register File



Status Register (SREG)

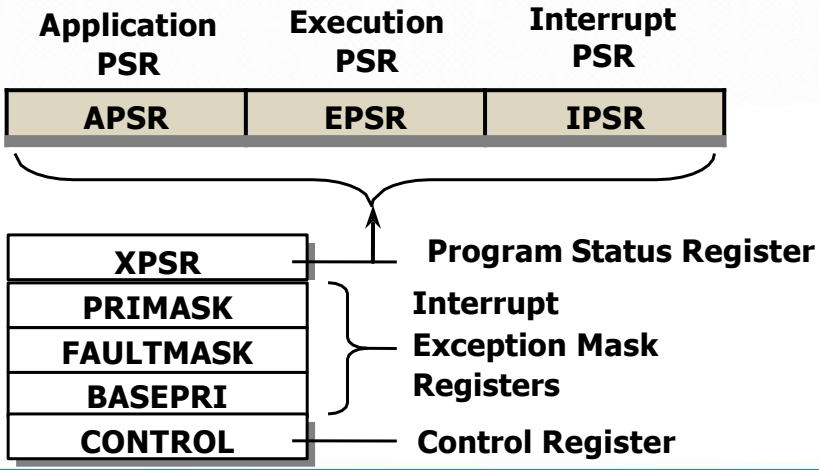
Bits	31 30 29 28 27	26:25	24	23:20	19:16	15:10	9	8 7 6 5 4 3 2 1 0
APSR	N Z C V Q				GE*	Reserved		
IPSR	Reserved							Exception Number
EPSR	Reserved	ICI/IT	T	Reserved		ICI/IT	Reserved	

(a) Three individual register – APSR, IPSR and EPSR.

Bits	31 30 29 28 27	26:25	24	23:20	19:16	15:10	9	8 7 6 5 4 3 2 1 0
PSR	N Z C V Q	ICI/IT	T		GE*	ICI/IT		Exception Number

(b) The combined register PSR.

Structure and bit functions in special registers.



Status Register (SREG): Common flags

- Z: **Zero flag**
  - Set to 1 when the result of an instruction is 0
- C: **Carry flag**
  - Set to 1 when the result of an instruction causes a carry to occur
- N: **Negative**
  - Set to 1 to indicate the result of an instruction is negative
- V: **Overflow**
  - Set to 1 to indicate the result of an instruction caused an overflow

Exercise: 64-bit add

```
//long long is a 64-bit type
long long A; //located at 0x1000_A000
long long B; //located at 0x1000_B000

A = A + B;
```

Data Memory

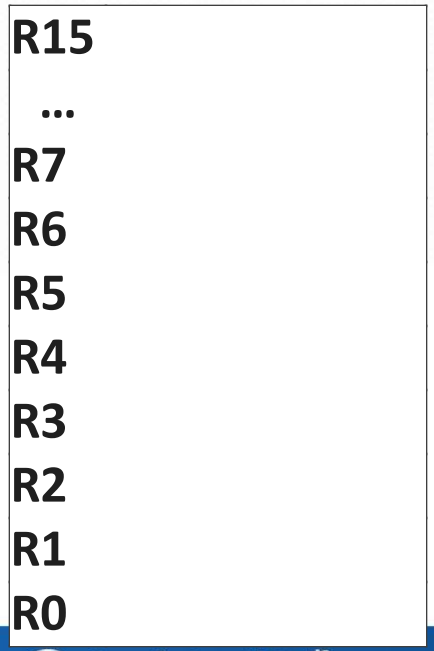
	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
	0x1000_B000	0x00
	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
B	0x1000_B000	0x01
	…	
	0x1000_A007	0x00
	0x1000_A006	0x00
	0x1000_A005	0x00
	0x1000_A004	0xFF
	0x1000_A003	0xFF
	0x1000_A002	0xFF
	0x1000_A001	0xFF
A	0x1000_A000	0xFF

Exercise: 64-bit add

```
//long long is a 64-bit type
long long A; //located at 0x1000_A000
long long B; //located at 0x1000_B000

A = A + B;
```

Register File



Data Memory

0x1000_B003	0x00
0x1000_B002	0x00
0x1000_B001	0x00
0x1000_B000	0x00
0x1000_B003	0x00
0x1000_B002	0x00
0x1000_B001	0x00
B 0x1000_B000	0x01
...	
0x1000_A007	0x00
0x1000_A006	0x00
0x1000_A005	0x00
0x1000_A004	0xFF
0x1000_A003	0xFF
0x1000_A002	0xFF
0x1000_A001	0xFF
A 0x1000_A000	0xFF

Exercise: 64-bit add

```
//long long is a 64-bit type
long long A; //located at 0x1000_A000
long long B; //located at 0x1000_B000

A = A + B;

MOVW R0, 0xA000;get A address
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	
R1	
R0	0x0000_A000

Data Memory

0x1000_B003	0x00
0x1000_B002	0x00
0x1000_B001	0x00
0x1000_B000	0x00
0x1000_B003	0x00
0x1000_B002	0x00
0x1000_B001	0x00
B 0x1000_B000	0x01
…	
0x1000_A007	0x00
0x1000_A006	0x00
0x1000_A005	0x00
0x1000_A004	0xFF
0x1000_A003	0xFF
0x1000_A002	0xFF
0x1000_A001	0xFF
A 0x1000_A000	0xFF

Exercise: 64-bit add

```
//long long is a 64-bit type
long long A; //located at 0x1000_A000
long long B; //located at 0x1000_B000

A = A + B;

MOVW R0, 0xA000;get A address
MOVT R0, 0x1000
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	
R1	
R0	0x1000_A000

Data Memory

	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
	0x1000_B000	0x00
	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
B	0x1000_B000	0x01
	…	
	0x1000_A007	0x00
	0x1000_A006	0x00
	0x1000_A005	0x00
	0x1000_A004	0xFF
	0x1000_A003	0xFF
	0x1000_A002	0xFF
	0x1000_A001	0xFF
A	0x1000_A000	0xFF

Exercise: 64-bit add

```
//long long is a 64-bit type
long long A; //located at 0x1000_A000
long long B; //located at 0x1000_B000

A = A + B;
```

```
MOVW R0, 0xA000;get A address
MOVT R0, 0x1000
MOVW R1, 0xB000;get B Address
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	
R1	0x0000_B000
R0	0x1000_A000

Data Memory

0x1000_B003	0x00
0x1000_B002	0x00
0x1000_B001	0x00
0x1000_B000	0x00
0x1000_B003	0x00
0x1000_B002	0x00
0x1000_B001	0x00
B 0x1000_B000	0x01
…	
0x1000_A007	0x00
0x1000_A006	0x00
0x1000_A005	0x00
0x1000_A004	0xFF
0x1000_A003	0xFF
0x1000_A002	0xFF
0x1000_A001	0xFF
A 0x1000_A000	0xFF

Exercise: 64-bit add

```
//long long is a 64-bit type
long long A; //located at 0x1000_A000
long long B; //located at 0x1000_B000

A = A + B;
```

```
MOVW R0, 0xA000;get A address
MOVT R0, 0x1000
MOVW R1, 0xB000;get B Address
MOVT R1, 0x1000
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	
R1	0x1000_B000
R0	0x1000_A000

Data Memory

	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
	0x1000_B000	0x00
	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
B	0x1000_B000	0x01
	…	
	0x1000_A007	0x00
	0x1000_A006	0x00
	0x1000_A005	0x00
	0x1000_A004	0xFF
	0x1000_A003	0xFF
	0x1000_A002	0xFF
	0x1000_A001	0xFF
A	0x1000_A000	0xFF

Exercise: 64-bit add

```
//long long is a 64-bit type
long long A; //located at 0x1000_A000
long long B; //located at 0x1000_B000

A = A + B;

MOVW R0, 0xA000;get A address
MOVT R0, 0x1000
MOVW R1, 0xB000;get B Address
MOVT R1, 0x1000

LDR R2, [R0, #0] ;load low A
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	
R2	0xFFFF_FFFF
R1	0x1000_B000
R0	0x1000_A000

Data Memory

	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
	0x1000_B000	0x00
	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
B	0x1000_B000	0x01
	…	
	0x1000_A007	0x00
	0x1000_A006	0x00
	0x1000_A005	0x00
	0x1000_A004	0xFF
	0x1000_A003	0xFF
	0x1000_A002	0xFF
	0x1000_A001	0xFF
A	0x1000_A000	0xFF

Exercise: 64-bit add

```
//long long is a 64-bit type
long long A; //located at 0x1000_A000
long long B; //located at 0x1000_B000

A = A + B;

MOVW R0, 0xA000;get A address
MOVT R0, 0x1000
MOVW R1, 0xB000;get B Address
MOVT R1, 0x1000

LDR R2, [R0, #0] ;load low A
LDR R3, [R0, #4] ;load hi A
```

Register File

R15	
…	
R7	
R6	
R5	
R4	
R3	0x0000_00FF
R2	0xFFFF_FFFF
R1	0x1000_B000
R0	0x1000_A000

Data Memory

	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
	0x1000_B000	0x00
	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
	0x1000_B000	0x01
B	…	
	0x1000_A007	0x00
	0x1000_A006	0x00
	0x1000_A005	0x00
	0x1000_A004	0xFF
	0x1000_A003	0xFF
	0x1000_A002	0xFF
	0x1000_A001	0xFF
A	0x1000_A000	0xFF

Exercise: 64-bit add

```
//long long is a 64-bit type
long long A; //located at 0x1000_A000
long long B; //located at 0x1000_B000

A = A + B;
```

```
MOVW R0, 0xA000;get A address
MOVT R0, 0x1000
MOVW R1, 0xB000;get B Address
MOVT R1, 0x1000

LDR R2, [R0, #0] ;load low A
LDR R3, [R0, #4] ;load hi A
LDR R4, [R1, #0] ;load low B
```

Register File

R15	
…	
R7	
R6	
R5	
R4	0x0000_0001
R3	0x0000_00FF
R2	0xFFFF_FFFF
R1	0x1000_B000
R0	0x1000_A000

Data Memory

	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
	0x1000_B000	0x00
	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
B	0x1000_B000	0x01
	…	
	0x1000_A007	0x00
	0x1000_A006	0x00
	0x1000_A005	0x00
	0x1000_A004	0xFF
	0x1000_A003	0xFF
	0x1000_A002	0xFF
	0x1000_A001	0xFF
A	0x1000_A000	0xFF

Exercise: 64-bit add

```
//long long is a 64-bit type
long long A; //located at 0x1000_A000
long long B; //located at 0x1000_B000

A = A + B;
```

```
MOVW R0, 0xA000;get A address
MOVT R0, 0x1000
MOVW R1, 0xB000;get B Address
MOVT R1, 0x1000

LDR R2, [R0, #0] ;load low A
LDR R3, [R0, #4] ;load hi A
LDR R4, [R1, #0] ;load low B
LDR R5, [R1, #4] ;load hi B
```

Register File

R15	
R7	
R6	
R5	0x0000_0000
R4	0x0000_0001
R3	0x0000_00FF
R2	0xFFFF_FFFF
R1	0x1000_B000
R0	0x1000_A000

Data Memory

	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
	0x1000_B000	0x00
	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
B	0x1000_B000	0x01
	…	
	0x1000_A007	0x00
	0x1000_A006	0x00
	0x1000_A005	0x00
	0x1000_A004	0xFF
	0x1000_A003	0xFF
	0x1000_A002	0xFF
	0x1000_A001	0xFF
A	0x1000_A000	0xFF

Exercise: 64-bit add

```
//long long is a 64-bit type
long long A; //located at 0x1000_A000
long long B; //located at 0x1000_B000

A = A + B;
```

```
MOVW R0, 0xA000;get A address
MOVT R0, 0x1000
MOVW R1, 0xB000;get B Address
MOVT R1, 0x1000

LDR R2, [R0, #0] ;load low A
LDR R3, [R0, #4] ;load hi A
LDR R4, [R1, #0] ;load low B
LDR R5, [R1, #4] ;load hi B

ADDS R2, R4   ; A+B low
```

Register File

R15	
R7	
R6	
R5	0x0000_0000
R4	0x0000_0001
R3	0x0000_00FF
R2	0x0000_0000
R1	0x1000_B000
R0	0x1000_A000

Data Memory

0x1000_B003	0x00
0x1000_B002	0x00
0x1000_B001	0x00
0x1000_B000	0x00
0x1000_B003	0x00
0x1000_B002	0x00
0x1000_B001	0x00
0x1000_B000 (B)	0x01
…	
0x1000_A007	0x00
0x1000_A006	0x00
0x1000_A005	0x00
0x1000_A004	0xFF
0x1000_A003	0xFF
0x1000_A002	0xFF
0x1000_A001	0xFF
0x1000_A000 (A)	0xFF

Exercise: 64-bit add

```
//long long is a 64-bit type
long long A; //located at 0x1000_A000
long long B; //located at 0x1000_B000

A = A + B;

MOVW R0, 0xA000;get A address
MOVT R0, 0x1000
MOVW R1, 0xB000;get B Address
MOVT R1, 0x1000

LDR R2, [R0, #0] ;load low A
LDR R3, [R0, #4] ;load hi A
LDR R4, [R1, #0] ;load low B
LDR R5, [R1, #4] ;load hi B

ADDS R2, R4   ; A+B low
ADCS R3, R5 ;A+B hi + carry
```

Register File

R15	
R7	
R6	
R5	0x0000_0000
R4	0x0000_0001
R3	0x0000_0100
R2	0x0000_0000
R1	0x1000_B000
R0	0x1000_A000

Data Memory

	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
	0x1000_B000	0x00
	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
B	0x1000_B000	0x01
	…	
	0x1000_A007	0x00
	0x1000_A006	0x00
	0x1000_A005	0x00
	0x1000_A004	0xFF
	0x1000_A003	0xFF
	0x1000_A002	0xFF
	0x1000_A001	0xFF
A	0x1000_A000	0xFF

Exercise: 64-bit add

```
//long long is a 64-bit type
long long A; //located at 0x1000_A000
long long B; //located at 0x1000_B000

A = A + B;

MOVW R0, 0xA000;get A address
MOVT R0, 0x1000
MOVW R1, 0xB000;get B Address
MOVT R1, 0x1000

LDR R2, [R0, #0] ;load low A
LDR R3, [R0, #4] ;load hi A
LDR R4, [R1, #0] ;load low B
LDR R5, [R1, #4] ;load hi B

ADDS R2, R4   ; A+B low
ADCS R3, R5 ;A+B hi + carry

STR R2, [R0, #0];Store A low
```

Register File

R15	
R7	
R6	
R5	0x0000_0000
R4	0x0000_0001
R3	0x0000_0100
R2	0x0000_0000
R1	0x1000_B000
R0	0x1000_A000

Data Memory

	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
	0x1000_B000	0x00
	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
B	0x1000_B000	0x01
	…	
	0x1000_A007	0x00
	0x1000_A006	0x00
	0x1000_A005	0x00
	0x1000_A004	0xFF
	0x1000_A003	0x00
	0x1000_A002	0x00
	0x1000_A001	0x00
A	0x1000_A000	0x00

Exercise: 64-bit add

```
//long long is a 64-bit type
long long A; //located at 0x1000_A000
long long B; //located at 0x1000_B000

A = A + B;

MOVW R0, 0xA000;get A address
MOVT R0, 0x1000
MOVW R1, 0xB000;get B Address
MOVT R1, 0x1000

LDR R2, [R0, #0] ;load low A
LDR R3, [R0, #4] ;load hi A
LDR R4, [R1, #0] ;load low B
LDR R5, [R1, #4] ;load hi B

ADDS R2, R4   ; A+B low
ADCS R3, R5 ;A+B hi + carry
 STR R2, [R0, #0];Store A low
STR R3, [R0, #4];Store A hi
```

Register File

R15	
R7	
R6	
R5	0x0000_0000
R4	0x0000_0001
R3	0x0000_0100
R2	0x0000_0000
R1	0x1000_B000
R0	0x1000_A000

Data Memory

	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
	0x1000_B000	0x00
	0x1000_B003	0x00
	0x1000_B002	0x00
	0x1000_B001	0x00
B	0x1000_B000	0x01
	…	
	0x1000_A007	0x00
	0x1000_A006	0x00
	0x1000_A005	0x01
	0x1000_A004	0x00
	0x1000_A003	0x00
	0x1000_A002	0x00
	0x1000_A001	0x00
A	0x1000_A000	0x00

ADD Immediate

ADDW: Add a 12-bit constant to a register

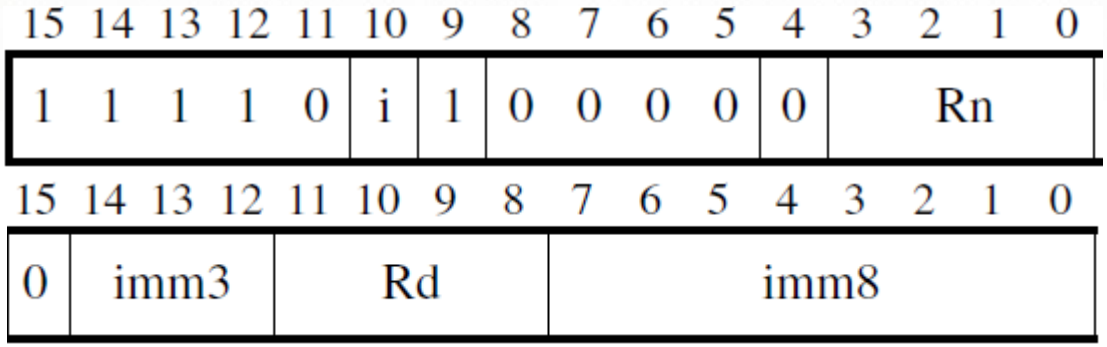
Syntax: ADDW {Rd,} Rn, #Imm12

Operands: 16 ≤ d ≤ 31, 0 ≤ n ≤ 31, 0 ≤ #Imm12 ≤ 2047

#Imm12 = i:imm3:Imm8

Operations: Rd ← Rn+#Imm12, PC ← PC+4

Binary Format:



Logical AND

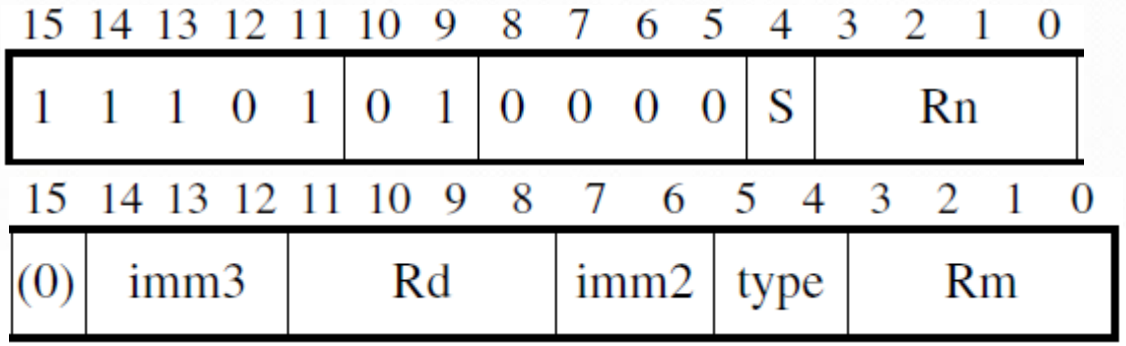
AND: Bitwise AND of two registers

Syntax: AND{S} Rd, Rn, Rm {,shift}

Operands: 0 ≤ d ≤ 15, 0 ≤ n ≤ 15, 0 ≤ m ≤ 15

Operations: Rd ← Rn `and` Rm (omitting shift), PC ← PC+4

Binary Format:



`shift` omitted

imm3:imm2= 0

type = 0

Note: if the `shift` is not omitted, then Rm is shifted based on the encodings given in ARM Instruction Set

Logical AND with Immediate

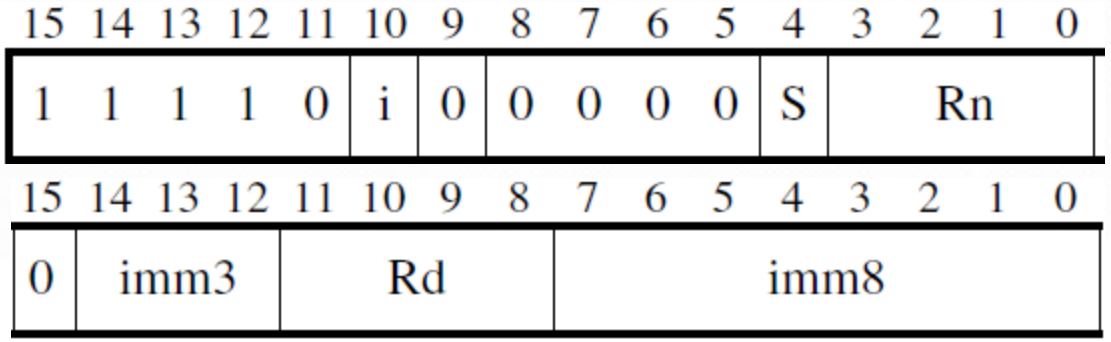
AND: Bitwise AND of register with constant

Syntax: AND{S} Rd, Rn, #const

Operands: 0 ≤ d ≤ 15, 0 ≤ n ≤ 15, 0 ≤ const ≤ 255

Operations: Rd ← Rn `and` #const, PC ← PC+4

Binary Format:



0 ≤ const ≤ 255

`const=`

`i:imm3:imm8`

Note: If `const`is greater than 255, See section 4.2 of ARM Instruction Set how to determine valid values for `const`.

Multiply (32-bit result)

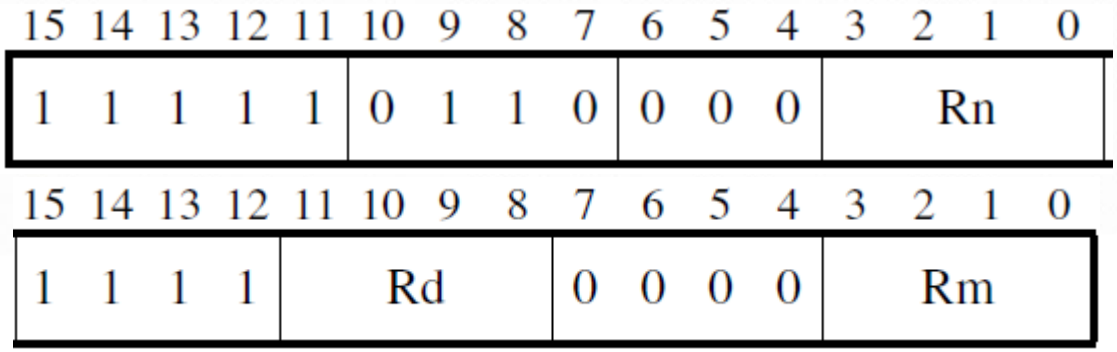
MUL: Multiply two registers give lower 32 bits of result

Syntax: MUL {Rd,} Rn, Rm

Operands: $0 \le d \le 15$, $0 \le n \le 15$, $0 \le m \le 15$

Operations: Rd ← Rn * Rm, PC ← PC+4

Binary Format:



Multiply Unsigned (64-bit result)

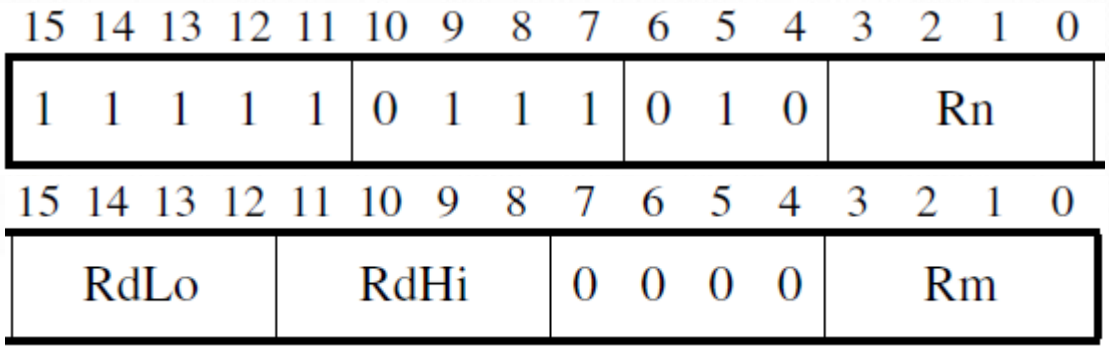
UMULL: Multiply two registers gives 64 bit result

Syntax: UMULL RdLo, RdHi, Rn, Rm

Operands: $0 \le d \le 15$, $0 \le n \le 15$, $0 \le Lo \le 15$, $0 \le Hi \le 15$

Operations: RdHi:RdLo $\leftarrow$ Rn * Rm, PC $\leftarrow$ PC+4

Binary Format:



Major Classes of Assembly Instructions

- Data Movement
  - Move data between registers
  - Move data in & out of Memory
  - Different addressing modes
- Logic & Arithmetic
  - Addition, subtraction, etc.
  - AND, ORR, EOR (Exclusive OR), bit shift, etc.
- Control Flow
  - Control which sections of code should be executed (e.g. In C “IF”, “CASE”, “WHILE”, etc.
  - Function Calls

ARM Assembly Programming: Translating C Control Statements and Function Calls

C Control Statements

Recall control statements in C:

If statement:

```
if (cond)
{
   if-body;
}
```

If-else statement:

```
if (cond)
{
   if-body
}
else
{
   else-body;
}
```

C Control Statements

Loop statements:

While statement:

```
while (cond)
{
   loop-body;
}
```

Do-While statement:

```
do
{
   loop-body
} while (cond);
```

For statement:

```
for (init-expr; cond-expr; incr-expr)
{
   loop-body;
}
```

Control Flow

- **Control Flow**: mechanisms that allow a program to make **decisions** about what instruction to be executed next from program memory
- Without control flow:
  - A CPU would **always just execute the next** instruction in program (code) memory
  - A computer would be limited in the types of functionality it could execute
- Two assembly instructions are typically part of any assembly langue to enable the implementation of control flow
  - **Compare:** Typically **subtracts** two things and **updates the flags** in the Status Register (i.e. Zero, Carry, Negative, Overflow flags)
  - **Branch:** Based on which flags of the Status Register are set, it directs the CPU to go to the next instruction in code memory or to “branch” to an alternative place in code memory.

Control Flow Basics

Steps to implement control flow:

1. **Set flags** in the Status Register
2. Use a **Branch** instruction to check for particular Status Register flag values
   - A. **Branch not taken:** If the **conditions** being checked are **not met**, then go to the next instruction in program (code) memory.
   - B. **Branch taken:** if the conditions being check are met, then go to an alternative location in code memory, typically indicated by a label.

Example:

```
MOVW R1, 0xFFFF
MOVW R2, 0x0000
CMP R1, R2  ;compare R1, and R2 and set flags
BEQ endif   ;check flags, and branch to "endif"
    ⋮       ;if r1 == r2 (i.e. Zero flag(Z) =1)
endif:
```

Setting Status Register Flags

- **Which instructions can update the Status Register Flags?**:
  - In general, consult the Assembly Instruction Set manual
- Some examples:
  - **Compare**: e.g. CMP, often used to help check for ==, !=, >, ≥, <, ≤.
  - **Test**: e.g. TST, TEQ, often used to check if given bits are **set** in a register, or check if a register **is equal to** a given value
  - **Arithmetic:** Most ARM arithmetic instructions when appended with an "**S**" update the Status Register
  - **Logic**: Most ARM logic instructions when appended with an "**S**" update the Status Register
  - **Move:** ARM has a MOV**S** that updates the Status Register

Compare (register)

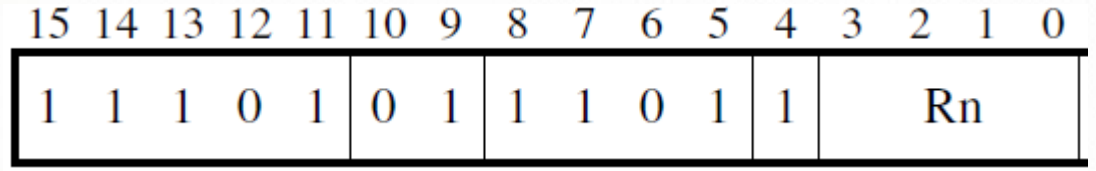
CMP: Subtracts two registers. Updates Status register

Syntax: CMP Rn, Rm {,<shift>}

Operands: 0 ≤ n ≤ 15, 0 ≤ m ≤ 15

Operations: Rn − Rm (omitting shift), PC ← PC+4

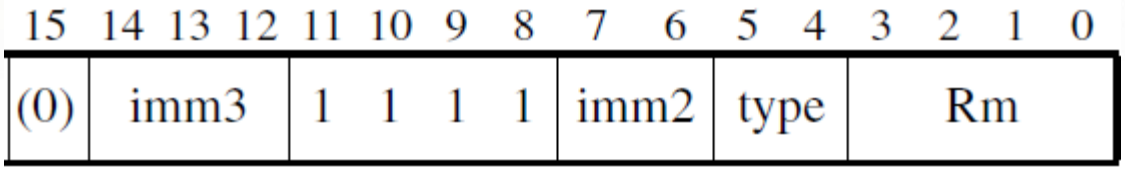
Binary Format:



`shift` omitted

imm3:imm2= 0

type = 0



Note: if the `shift` is not omitted, then Rm is shifted based on the encodings given in ARM Instruction Set

Compare (Immediate)

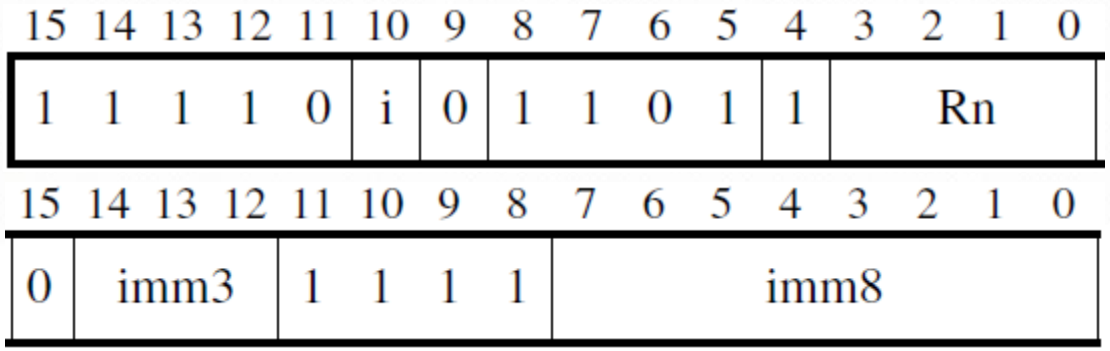
CMP: Subtract constant from reg. Update Status Register.

Syntax: CMP Rn, #const

Operands: $0 \le n \le 15$, $0 \le const \le 255$

Operations: Rn − #const, PC ← PC+4

Binary Format:



$0 \le const \le 255$

`const=`

`i:imm3:imm8`

Note: If `const` is greater than 255, See section 4.2 of ARM Instruction Set how to determine valid values for `const`.

Test (Register)

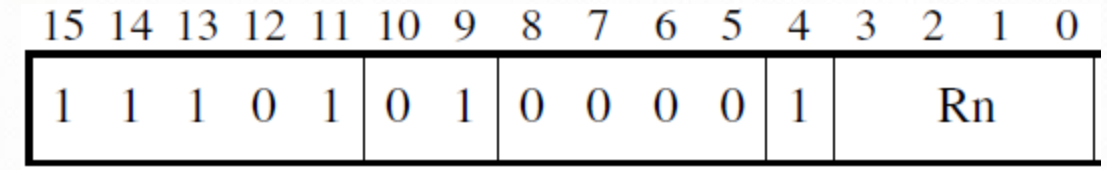
TST: Bitwise AND of two registers. Updates Status Register

Syntax: TST Rn, Rm {,<shift>}

Operands: 0 ≤ n ≤ 15, 0 ≤ m ≤ 15

Operations: Rn `and` Rm (omitting shift), PC ← PC+4

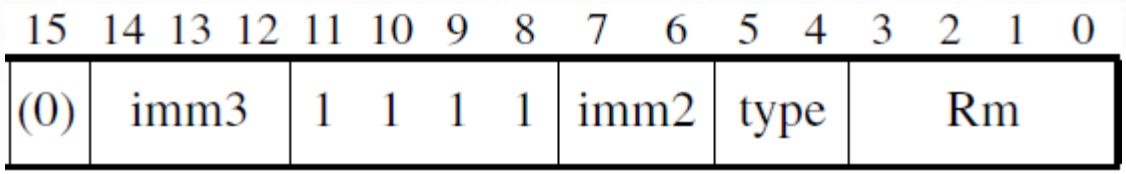
Binary Format:



`shift` omitted

imm3:imm2= 0

type = 0



Note: if `shift` is not omitted, then Rm is shifted based on the encodings giving in ARM Instruction Set

Test (Immediate)

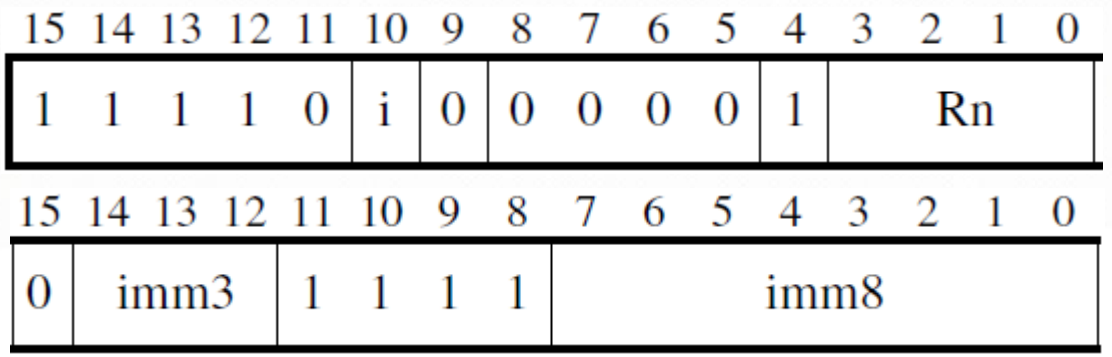
TST: Bitwise AND reg with constant. Update Status Register

Syntax: TST Rn, #const

Operands: 0 ≤ n ≤ 15, 0 ≤ const ≤ 255

Operations: Rn `and` #const, PC ← PC+4

Binary Format:



0 ≤ const ≤ 255

`const=`

`i:imm3:imm8`

Note: If `const` is greater than 255, See section 4.2 of ARM Instruction Set how to determine valid values for `const`.

Checking Status Register Flags

- Branch instructions check if particular *conditions* are satisfied by the Status Register flags.
  - If conditions are satisfied, then CPU branches to a label
  - If conditions are not satisfied CPU goes to next instruction.
- Some Branch conditions:
  - **BEQ / BNE**: **Is Zero flag (Z) = 1 or 0**, used for comparing signed or unsigned values
  - **BGT / BGE / BLT / BLE**: comparing Signed values
  - **BHI/BHS/BLO/BLS:** comparing Unsigned values

Branch (Conditional)

B{cond}: Branch to a target address if condition satisfied

Syntax: B{cond} label

Operands: **cond** (see 3.4.1 of ARM Instruction set)

Assembler encodes **label** into:PC+S:J2:J1:imm6:imm11:'0'

Operation: if(cond), then PC ←PC+ S:J2:J1:imm6:imm11:'0'

else PC ←PC+ 4

Binary Format:

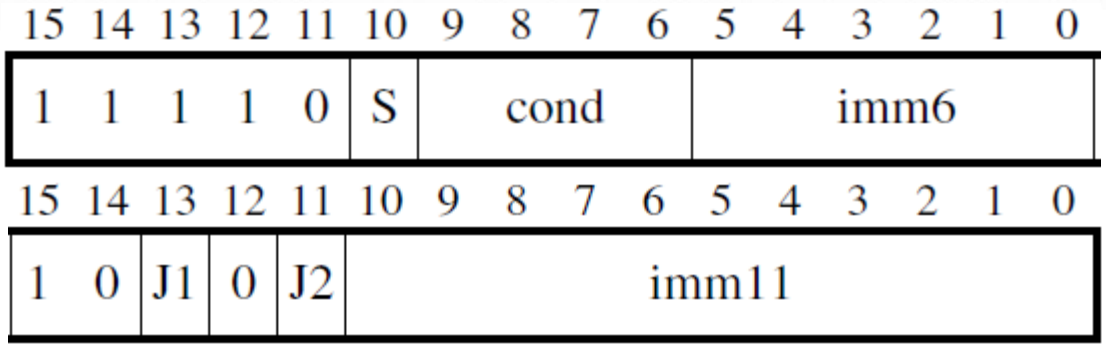


Table of Branch Conditions

Condition Encoding (cond)	Branch Type	Meaning	Status Flag State
0000	BEQ	Equal	Z = 1
0001	BNE	Not equal	Z = 0
0010	BHS	Higher or Same (Unsigned)	C = 1
0011	BLO	Lower (Unsigned)	C = 0
0100	BMI	Negative	N = 1
0101	BPL	Positive	N=0
0110	BVS	Overflow	V=1
0111	BCV	No overflow	V=0
1000	BHI	Higher (Unsigned)	C=1 & Z=0
1001	BLS	Lower or Same (Unsigned)	C=0 | Z=1
1010	BGE	Greater than or Equal (Signed)	N = V
1011	BLT	Less than (Signed)	N != V
1100	BGT	Greater than (Signed)	N=V & Z=0
1101	BLE	Less than or Equal (Signed)	N != V | Z=1

Table of Branch Conditions

Condition Encoding (cond)	Branch Type	Meaning	Status Flag State
0000	BEQ	Equal	Z = 1
0001	BNE	Not equal	Z = 0
0010	BHS	Higher or Same (Unsigned)	C = 1
0011	BLO	Lower (Unsigned)	C = 0
0100	BMI	Negative	N = 1
0101	BPL	Positive	N=0
0110	BVS	Overflow	V=1
0111	BCV	No overflow	V=0
1000	BHI	Higher (Unsigned)	C=1 & Z=0
1001	BLS	Lower or Same (Unsigned)	C=0 | Z=1
1010	BGE	Greater than or Equal (Signed)	N = V
1011	BLT	Less than (Signed)	N != V
1100	BGT	Greater than (Signed)	N=V & Z=0
1101	BLE	Less than or Equal (Signed)	N != V | Z=1

Example 1

Assume: a located @ 0x1000_A000, b located @ 0x1000_B000

Exercises: Write a sequence of instructions

(a) Branch to label if a < b, a and b are variables of “signed int”

(b) type Branch to label if a >= b, a and b are vars of “unsigned

(c) int” type Branch to label if a == b, a and b are “int” type

variables

Answer to Example 1

Assume: a located @ 0x1000_A000, b located @ 0x1000_B000

Exercise: Write a sequence of instructions

1. Branch to label if a < b (a and b signed int type)

```
MOVW R0, 0xA000;get a address
MOVT R0, 0x1000
MOVW R1, 0xB000;get b Address
MOVT R1, 0x1000
LDR R2, [R0, #0] ;load a
LDR R3, [R1, #0] ;load b
CMP R2, R3
BLT label                ; check if a is less than b
  ⋮
label:
```

Translate If-Statement

Example: Assume: a @ 0x1000_A000, b @ 0x1000_B000

```
if (a < b) // assume a and b are signed ints
{
  a = -a;
}
```

Translate If-Statement

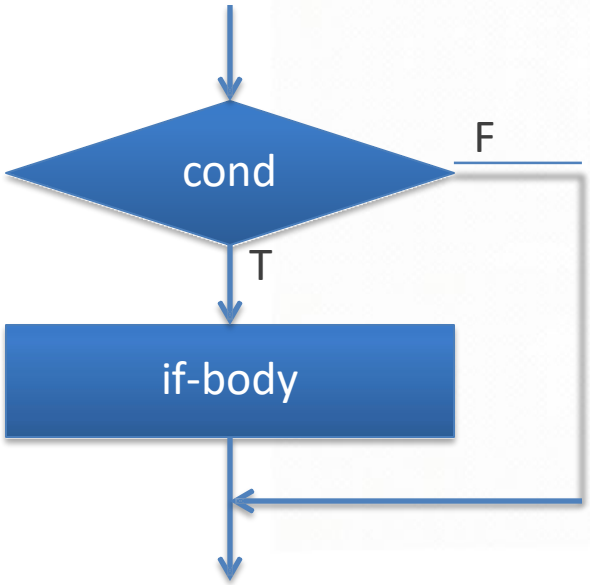
Example: Assume: a @ 0x1000_A000, b @ 0x1000_B000

```
if (a < b)
{
  a = -a;
}
```

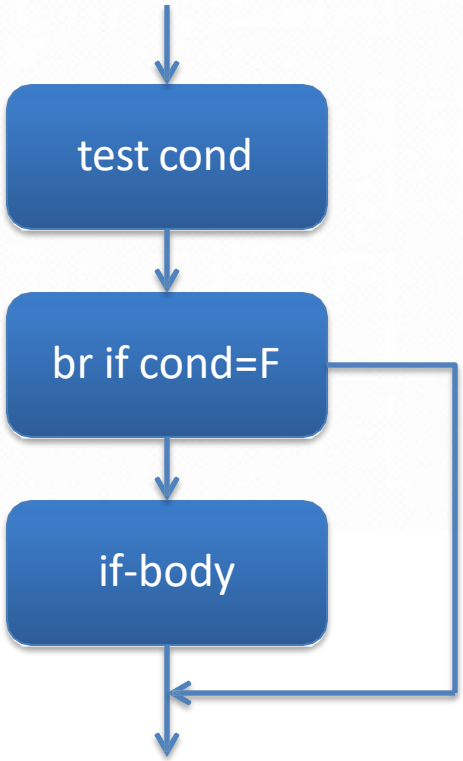
```
    MOVW R0, 0xA000;get a address
    MOVT R0, 0x1000
    MOVW R1, 0xB000;get b Address
    MOVT R1, 0x1000
    LDR R2, [R0, #0] ;load a
    LDR R3, [R1, #0] ;load b
    CMP R2, R3
    BGE endif ; take branch if complement true
    NEG R2     ; a = -a
    STR R2 [R0, #0] ;store a
endif:
```

If-Statement: Structure

Control and Data Flow Graph



Linear Code Layout



If-Statement: Structure

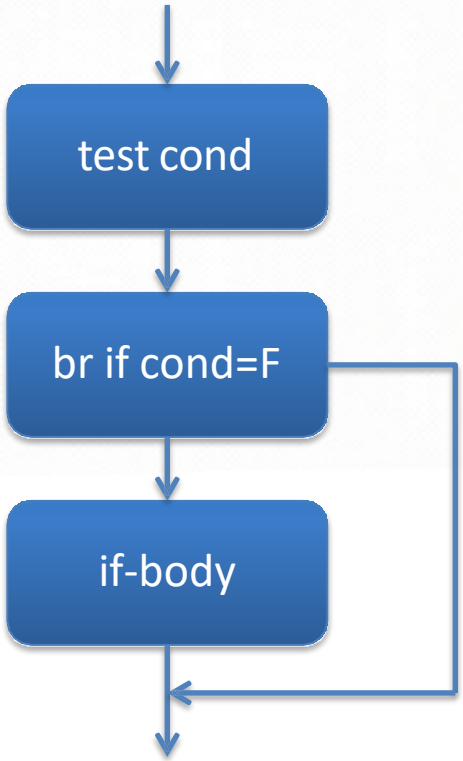
```
if (a < b)  // C code is testing for less than
{
   a = -a;
}
```

```
 MOVW R0, 0xA000;get a address
 MOVT R0, 0x1000
 MOVW R1, 0xB000;get b Address
 MOVT R1, 0x1000
 LDR R2, [R0, #0] ;load a
 LDR R3, [R1, #0] ;load b
 CMP R2, R3

 BGE endif ; if complement true

 NEG R2     ; a = -a
 STR R2 [R0, #0] ;store a

endif:
```



Branch (Unconditional)

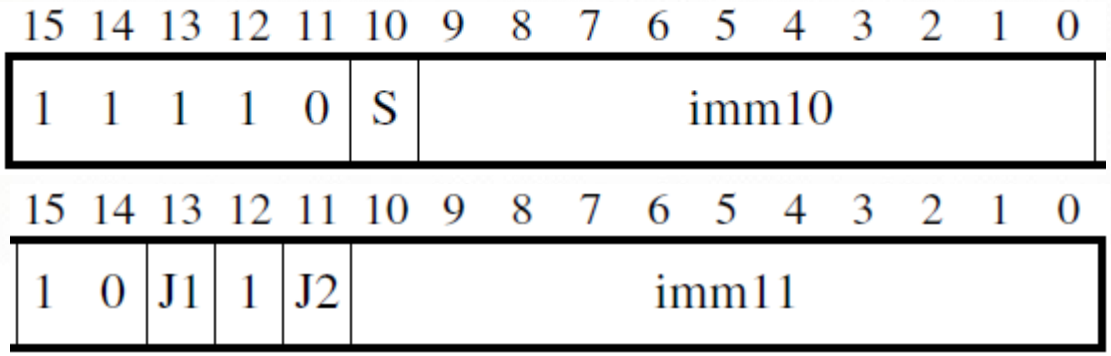
B: Branch to a target address

Syntax: B label

Operands: **`label`** encoded into: S:I1:I2:imm10:imm11:'0'

Operation: PC ←PC+ S:I1:I2:imm10:imm11:'0'

Binary Format:



I1 = NOT(J1 EOR S)

I2 = NOT(J2 EOR S)

IF-Else Statement

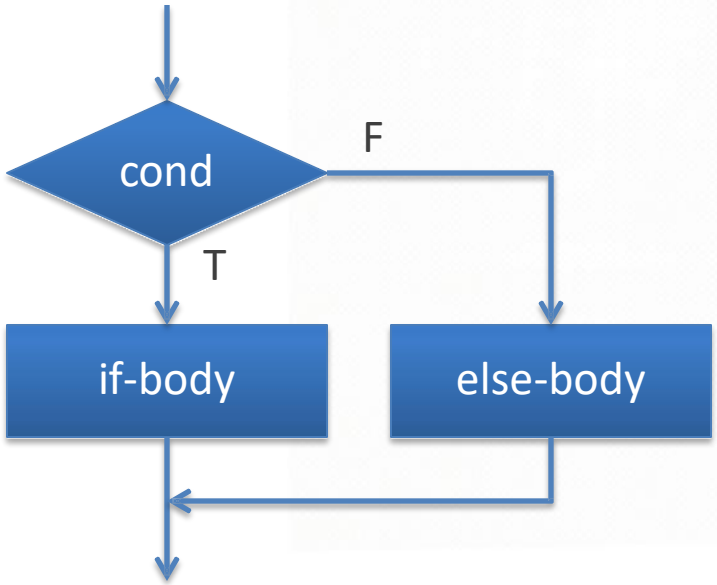
```
if (cond)
  if-body
else
  else-body;
```

Example:

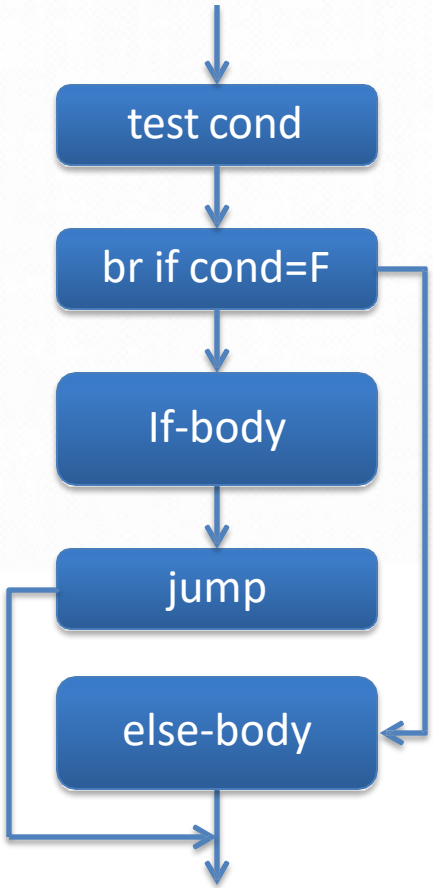
```
int max, a, b;
if (a < b)
  max = b;
else
  max = a;
```

If-Else Statement: Structure

Control and Data Flow Graph



Linear Code Layout

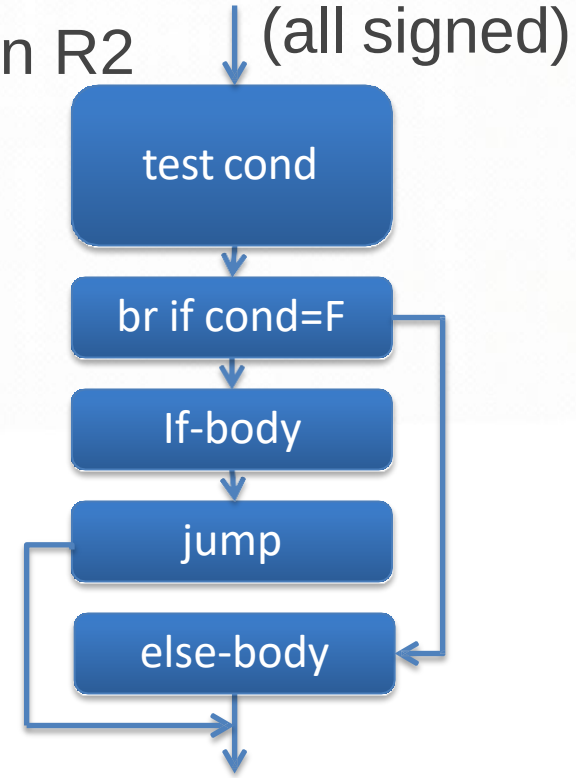


If-Else Statement: Structure

```
int max, a, b;
 if (a < b)
 max = b;
 else
   max = a;
```

```
; assume a in R0, b in R1, max address in R2
        CMP     R0, R1
        BGE     else

        STR     R1,  [R2, #0]
        B       endif
else:
        STR   R0, [R2, #0]
endif:  …
```



If and If-Else summary

For simple If and If-Else statement, the behavior of C and Assembly are complementary. So complement condition is used.

Examples:

C	Assembly
if (a >= b)	branch if a<b, use BLT
if (a > b)	branch if a<b, use BLE
if(a = b)	branch if a!=b, use BNE

Caveat when comping against a constant

C **Assembly (assume a in R0)**

If (5 > a), branch if 5 <= a?

```
        CMP #5,  R0
        BLE    endif
```

What is the issue?

Caveat when comping against a constant

C **Assembly (assume a in R0)**

If (5 > a), branch if 5 <= a?

~~CMP #5, R0~~

BLE endif

Problem: Constant must be **second** argument of compare!

Caveat when comping against a constant

Translate the C code into an equivalent condition, then compile into assembly using complement condition.

	C	translated C	Assembly
Case 1:	if (5 > a)	⇔ if (a < 5),	branch if a ≥ 5
			CMP R0, #5
			BGE endif

Compound Condition (&&)

if (a >= b && b< 10 && … && a > 20) // signed ints

; If only consists of Boolean **AND's**, follow complement rule

```
; assume a in R0, and b in R1
        CMP R0, R1

        BLT else        ; complement of ≥
        CMP R1, #10
        BGE else        ; complement of <
          …
        CMP R0, #20
        BLE else        ; complement of >
          …   ; if-body
        B endif
else:
          …   ; else-body
endif:
```

Recall C uses Lazy Evaluation https://en.wikipedia.org/wiki/Lazy_evaluation

Compound Condition (||)

if (a >= b || b< 10 || … || a > 20) // signed ints

;If only Boolean **OR's**, then complement only the last condition

```
; assume a in R0, and b in R1
            CMP R0, R1
            BGE if-body      ;  no complement, and br to if-body
            CMP R1, #10
            BLT if-body      ;  no complement, and br to if-body
              …
            CMP R0, #20
            BLE else          ;  complement of >
if-body:
              …   ; if-body
            B endif

else:         …   ; else-body
endif:
```

Again, Recall C uses Lazy Evaluation

Exercise 1

```
if ((a + b < c && flag1) || (flag2 && !(a == b)))
{
   // if-body
} else
{
   // else-body
}
```

Function Call Convention

What is required for supporting a function call?

- Passing parameters
- Getting the **return value**
- **Sharing registers** between Caller and Callee
- Local storage (typically placed in reg or on the Stack)
- Jumping to the Callee
- Returning to the Caller

Why we study the C calling convention

- Must follow it when mixing C with assembly or using pre-compiled C library functions
- The calling convention is **NOT** part of the instruction set architecture. It is an agreed-upon convention to allow a compiler and human to generate code that can work together

ARM Function Call Standard: Passing Parameters and Return Value

Function parameters

- Use R0, R1, R2, R3
- Parameters passed to R0 – R3 in order.
- Parameter size: 2, or 1 bytes
  - Value is Zero or Signed extended before placed in a register
- Parameter size 8 bytes
  - Use a pair of registers (e.g. R1:R0)
- Extra parameters placed on the Stack

Function return value

- 8-bit in R0 (Zero or Signed extended)
- 16-bit in R0 (Zero or Signed extended)
- 32-bit in R0
- 64-bit in R1:R0

ARM Function Call Standard: Sharing Registers

How to share registers between Caller and Callee?

Non-volatile(Callee must preserve):R4-R11, SP

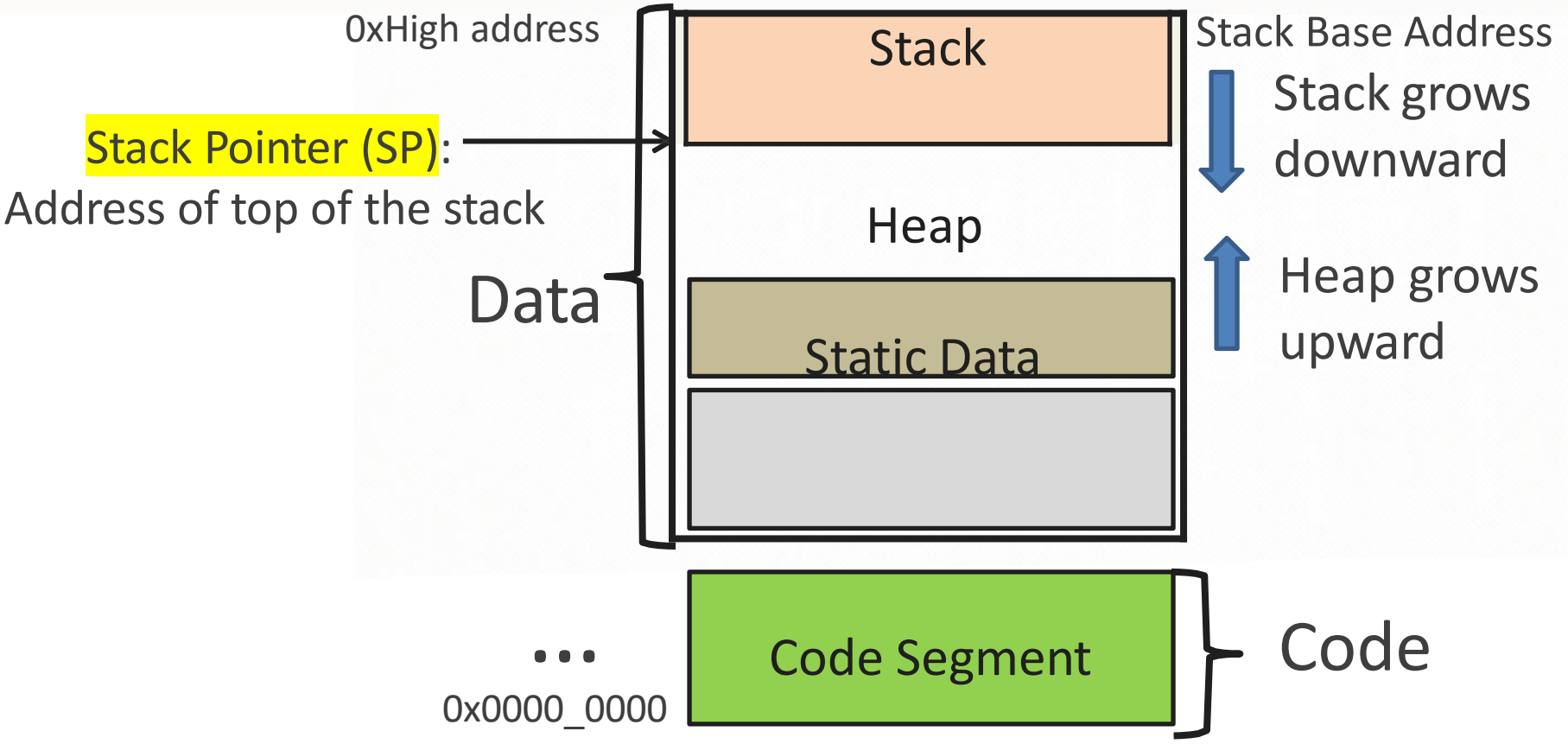
Callee must make sure these register values are preserved/restored. In other words, the Caller can assume the value of these registers after the Callee function returns will be the same as before the Callee function was called.

Volatile (Callee can freely modify): R0-R3, LR

If the Caller wants the values of these registers to be the same before and after the Callee function executes, then the **Caller** must preserve them before the Callee function executes, and restore them after Callee function executes.

Preserving and Restoring Registers

- Preserve a Register Value
  - Place the register value somewhere "safe"
  - A place in **Data Memory** called the Stack is this "safe" place
- Restore a Register Value
  - Place the value saved to the Stack back to the register
- Where is the Stack and how does it work?
  - Starts at the **highest address** in Data Memory (Bottom of Stack)
  - Values are placed and removed from the Top of the Stack
  - Placing a value on the Stack causes it to grow (Downward)
  - Removing a value from the Stack cause it to shrink (Upward)
  - Location of the Top of the Stack is always in register **`SP`** (R13).
- Assembly Instructions used
  - **`PUSH Rn`**: Place the value of Rn to the "top" of the Stack. Then decrease the value of **`SP`** by 4. **`[SP] ← Rn, SP ← SP-4`**.
  - **`POP Rn`**: Increase the value of **`SP`** by 4, then remove value at "top" of the Stack and place into Rn. **`SP←SP+4, Rn ← [SP]`**


0xHigh address
Stack
Stack Base Address
Stack grows downward
Stack Pointer (SP):
Address of top of the stack
Heap
Data
Heap grows upward
Static Data
Code Segment
Code
0x0000_0000

Summary Uses of the Stack

- Preserving and Restoring Registers
  - See previous slide
- Passing Function parameters when there are more parameters than parameter registers (R0 – R3)
  - Extra parameters **are PUSHed** onto the stack before a function call, **and POPed after** the function returns
- Store local variables, when it is not convenient to use registers
  - Note: it is more efficient to access data from a register, than from Data Memory (i.e. the Stack)
- Function Stack-Frame: Contains information associated with a called function
  - E.g. Function args, local vars, return address, preserved values

PUSH

PUSH: Place a register value onto the top of the Stack

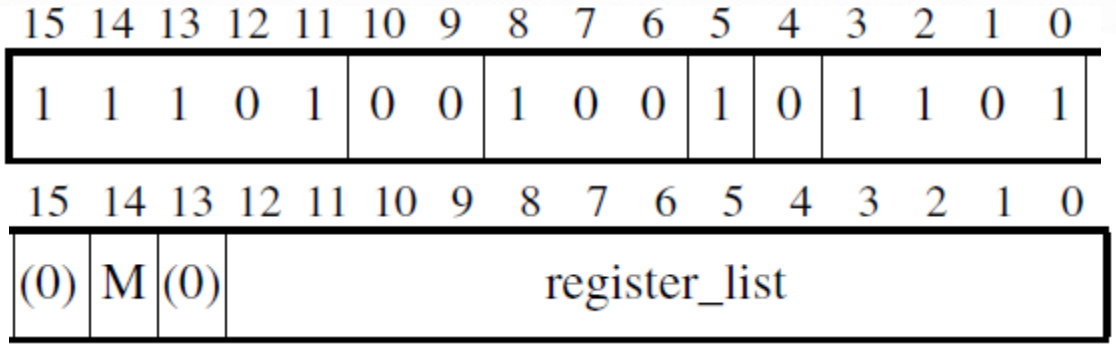
Syntax: PUSH Rn

Operands: 0 ≤ n ≤ 12, n=14: Note Rn can be a list of registers, encoded as: registers = '0':M:'0':register_list

Operation: [SP] ← Rn; SP ←SP - 4

PC ←PC+ 4

Binary Format:



POP

POP: Remove value at the top of Stack and place in a register.

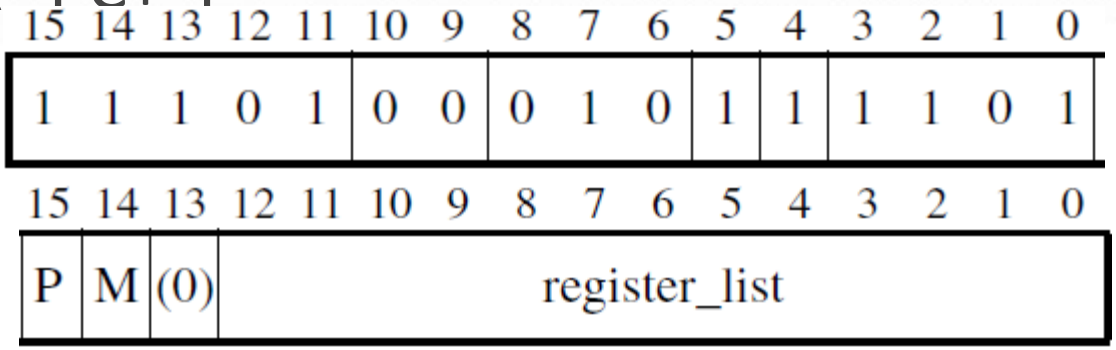
Syntax: POP Rn

Operands: 0 ≤ n ≤ 12, n=14, n=15: Note Rn can be a list of registers, encoded as: registers = P:M:'0':register_list

Operation: SP ←SP + 4; Rn ← [SP]

PC ←PC+ 4

Binary Format:



Jump to and return from a function

- Jump to Callee
  - Save address to use for returning to Caller
  - Update PC to address of first instruction of Callee function
- Return to Caller
  - Callee updates PC to get back to the Caller
- Assembly Instructions used
  - **`BL label`**: Branch with link, places PC + 4 into the LR (R14), and **`PC ← label`**. Where the label is the name of the function being called.
  - **`BX LR`**: Branch eXchange, places the value of LR (R14) into the PC. i.e. **`PC ← LR`**. Assuming LR (R14) contains the return address.

Branch with Link

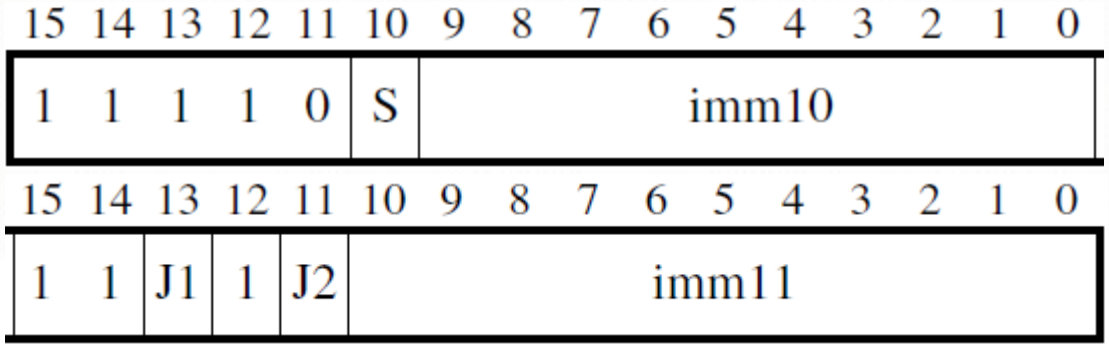
BL: Branch to target address & store PC+4 to Link Register

Syntax: BL label

Operands: **`label`** encoded into: S:I1:I2:imm10:imm11:'0'

Operation: LR ← PC+4; PC ←PC+ S:I1:I2:imm10:imm11:'0',

Binary Format:



I1 = NOT(J1 EOR S)

I2 = NOT(J2 EOR S)

Branch eXchange

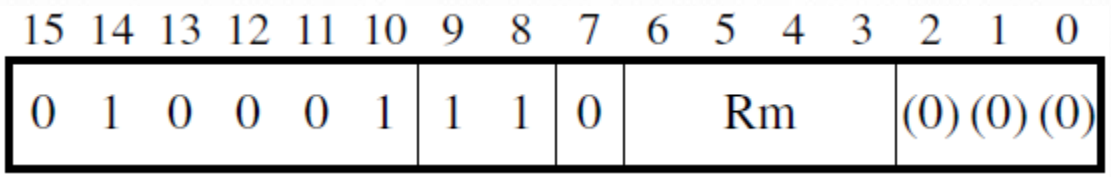
BX: Branch to address located in a register

Syntax: BX Rn

Operands: $0 \le n \le 15$

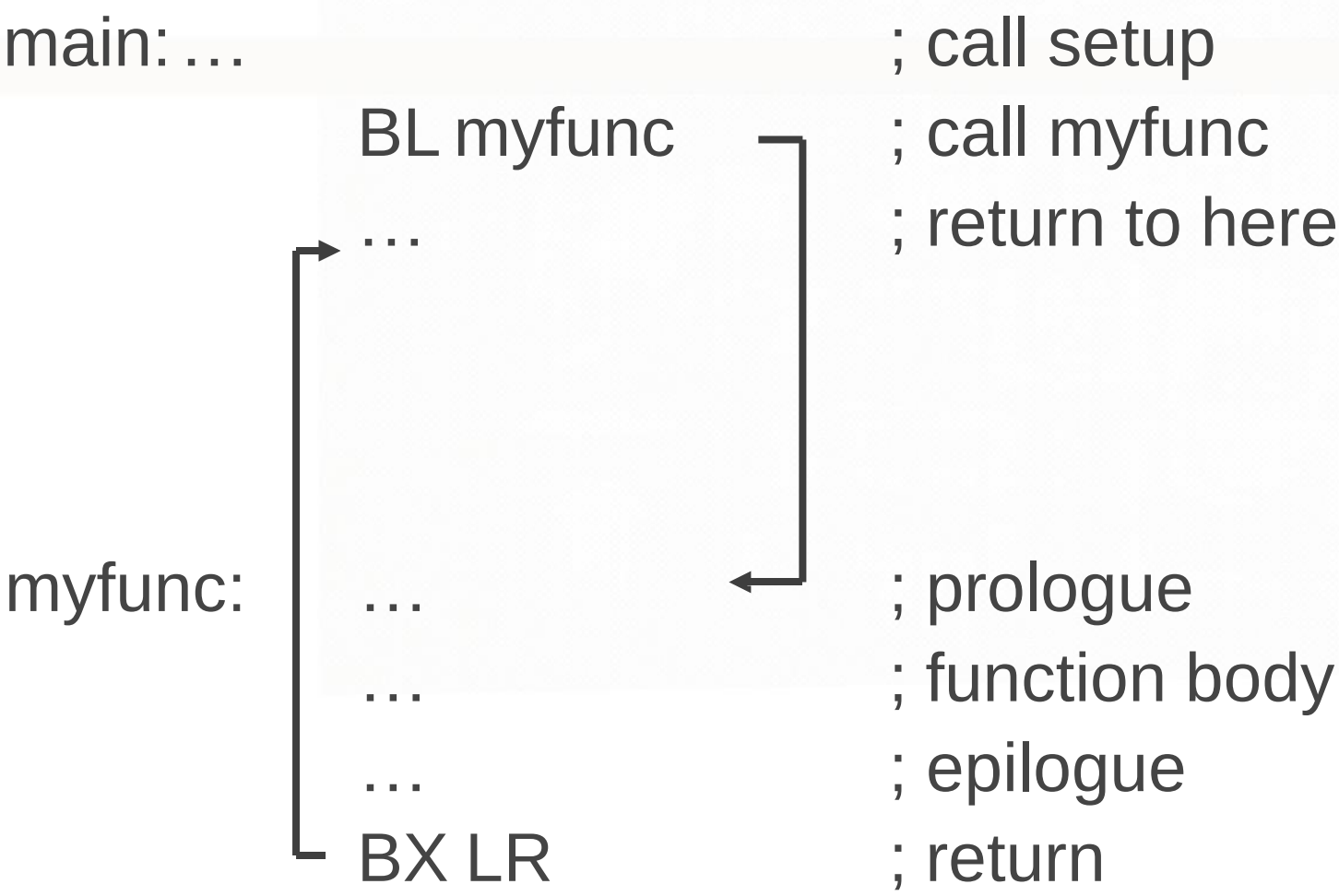
Operation: PC ← Rn

Binary Format:

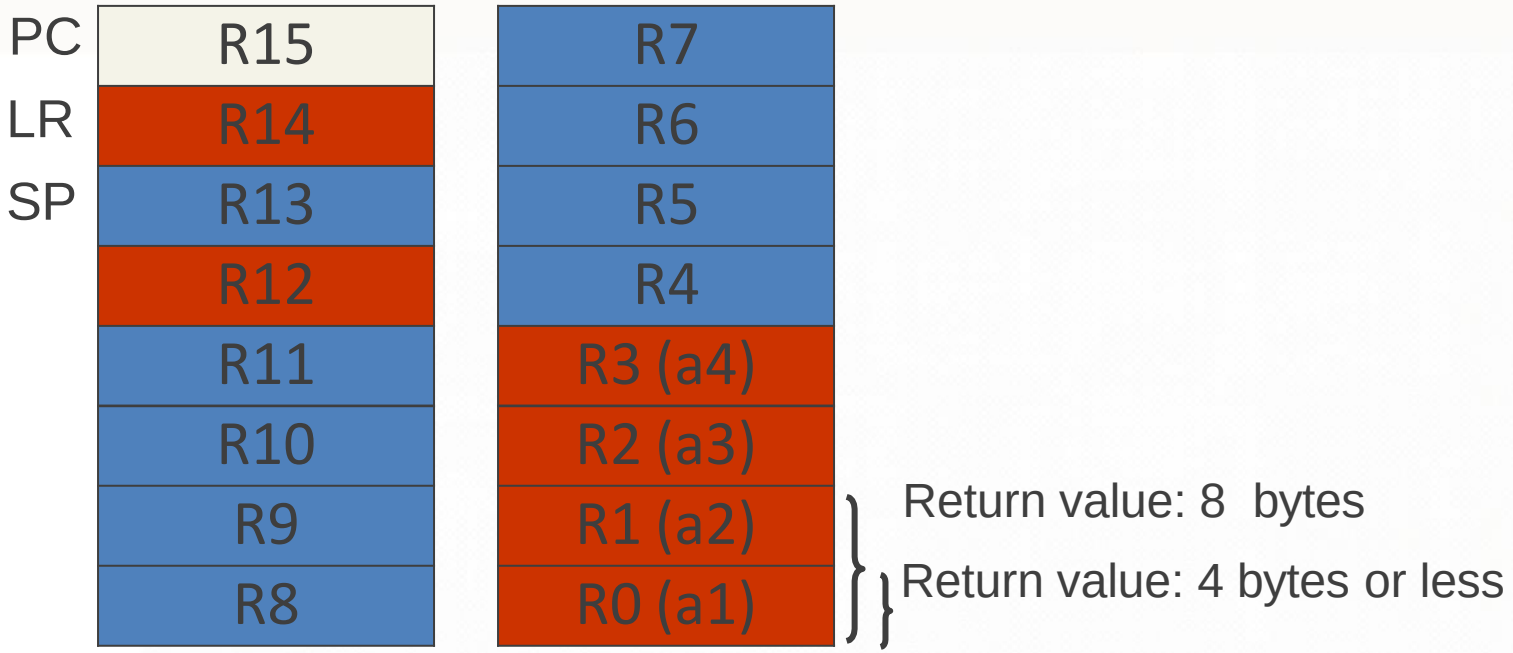


Note: Bit[0] of Rn specifies if the Thumb or ARM Instruction set should be used.

Function Call: Example



ARM Function Call Standard



Non-volatile: **Callee** must preserve/restore if it uses

Volatile: **Caller** must preserve/restore if it wants the register to maintain its value across a function call

a: Function argument (Assuming each param is 4 bytes or less)

Exercise 2(Function Call, Push & Pop)

```
int x, y, z;  // x @ 0x1000_A000, y @ 0x1000_B000, z @ 0x1000_C000

void my_func()
{
   …
   z = max(x, y);
   …
}

int max(int a, int b)
{
    if(a<b)
        return b;
    else
        return a;
}
```

Function Call: Ans of the Exercise 2

```
int max(int a, int b)
{
    if(a<b)
        return b;
    else
        return a;
}
```

```
max:
        ; a=>R0, b=>R1, return value in R0
        CMP R0, R1      ; compare a, b
        BRGE endif      ; branch if a>=b
        MOV R0, R1      ; move b to R0
endif:

        BX LR           ; Return to Caller
```

Example 2_1

```
int add2(int a, int b) {
  return a+b;
}

int add3(int a, int b, int c) {
  return add2(add2(a, b), c);
}

int sum, x, y, z; // x @ 0x1000_0000, y @ 0x1000_0004,
                  // z @ 0x1000_0008, sum @ 0x1000_000C
int main() {
  …
  sum = add3(x, y, z);
  …
}
```

Example 2_2

```
int add2(int a, int b) {
  return a+b;
}
```

```
add2:
        ; a=>R0, b=>R1, return value in R0
        ADDS R0, R1   ; a + b
        BX LR         ; Return to Caller
```

Example 2_3

```
int add3(int a, int b, int c) {
  return add2(add2(a, b), c);
}
```

```
add3:
        ; a=>R0,  b=>R1,  c=>R2,  return value in R0
        PUSH LR     ;Save return address for add3
        PUSH R2     ;Save c

        BL add2     ;call add2(a,b)

        POP R1      ;Restore c to R1
        BL add2     ;call add2(add2(a,b), c)

        POP LR      ;Restore return address for add3
        BX LR       ; Return to Caller
```

Question: Why save c?

Example 2_4

How main() calls add3: Assume for some reason R3 and R5

- must be preserved across the function call
- // sum = add3(x, y, z);
- // x @ 0x1000_0000, y @ 0x1000_0004, z @ 0x1000_0008
- // sum @ 0x1000_000C
- main:

```
PUSH R3   ;Save R3

MOVW R4, 0x0000   ; Get base address
MOVT R4, 0x1000
LDR R0, [R4, #0] ;load x ; 1st parameter of add3
LDR R1, [R4, #4] ;load y ; 2nd parameter of add3
LDR R2, [R4, #8] ;load z ; 3rd parameter of add3

BL add3       ;call add3(x,y, z), return value in R0

STR R0, [R4, #C] ; ; store result to sum
POP R3       ;Restore R3
```

Question: Is it not necessary to push/pop R5?

Translating C Control Statements and Function Calls, Loops, Interrupt Processing

C Control Statements

Loop statements:

While statement:

```
while (cond)
{
   loop-body;
}
```

Do-While statement:

```
do
{
   loop-body
} while (cond);
```

For statement:

```
for (init-expr; cond-expr; incr-expr)
{
   loop-body;
}
```

DO-WHILE Loop

Do-While statement:

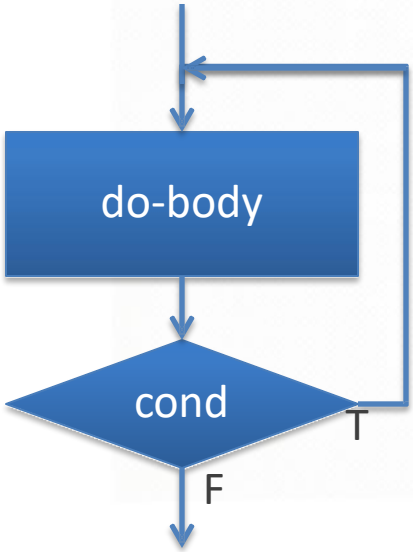
```
do
{
   loop-body
} while (cond);
```

Example:

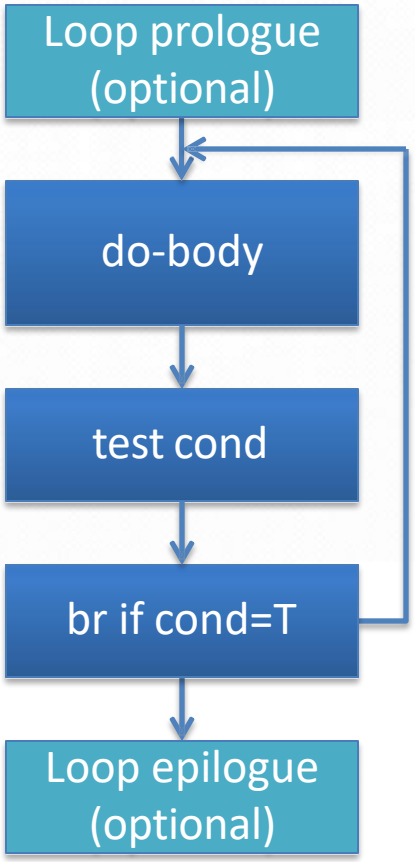
```
void strcpy (char *dst,
             char *src)
{
      char ch;
      do {
           ch = *src++;
           *dst++ = ch;
      } while (ch);
}
```

DO-WHILE Loop

Control and Data Flow Graph



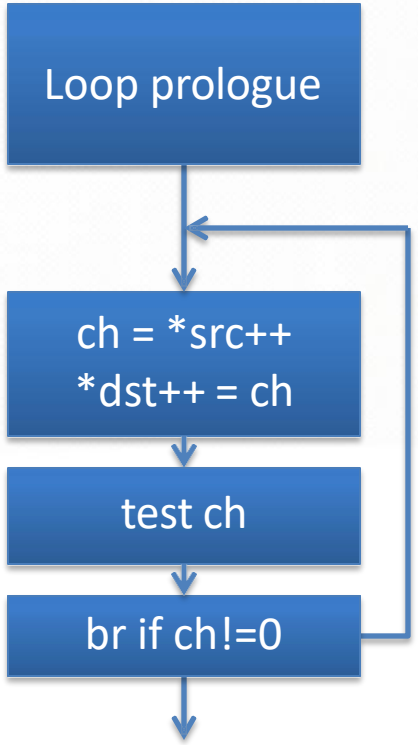
Linear Code Layout



DO-WHILE Loop

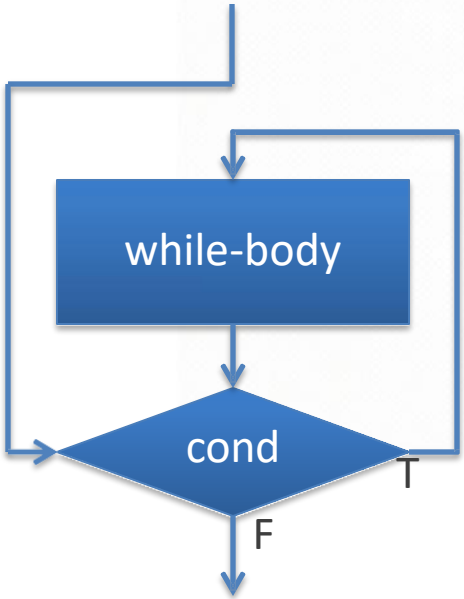
```
; parameter: dst=>R0, src=>R1
; reg use: ch=>R2
strcpy:
    ;e.g. initialize local vars
loop:
    LDRB R2, [R1], #1      ; Load 1 byte from [R1] into R2 and increment R1
    STRB R2, [R0], #1      ; Store 1 byte from R2 into [R0], then increment R0
    CMP R2, #0             ; Compare ch (R2) with 0 (null terminator '\0')
    BEQ end_copy           ; If ch == 0, end loop (branch if equal/zero flag set)
    B loop                 ; Otherwise, repeat loop
end_copy:
    BX LR                  ; Return to caller
```

```
do {
   ch = *src++;
   *dst++ = ch;
} while (ch != 0);
```

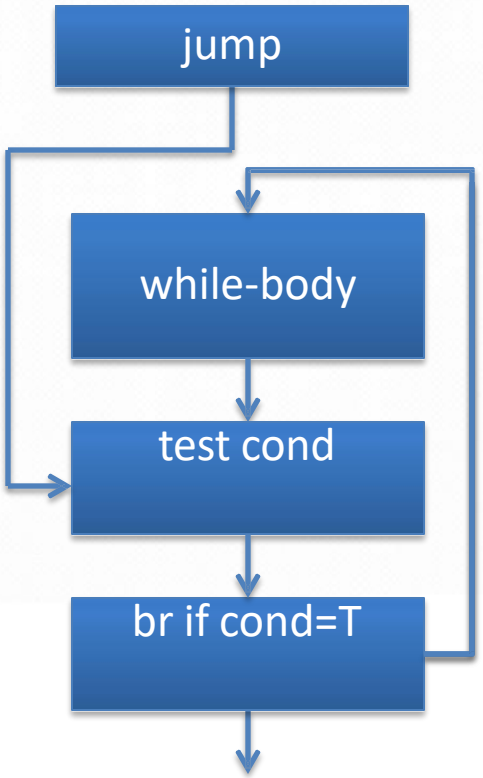


WHILE Loop

Control and Data Flow Graph



Linear Code Layout



(optional prologue and epilogue not shown)

WHILE Loop Example

strlen(): return the length of a C string

```
int strlen(char *str)
{
   int len = 0;
   while (*str++)
   {
      len++;
   }
   return len;
}
```

While statement:

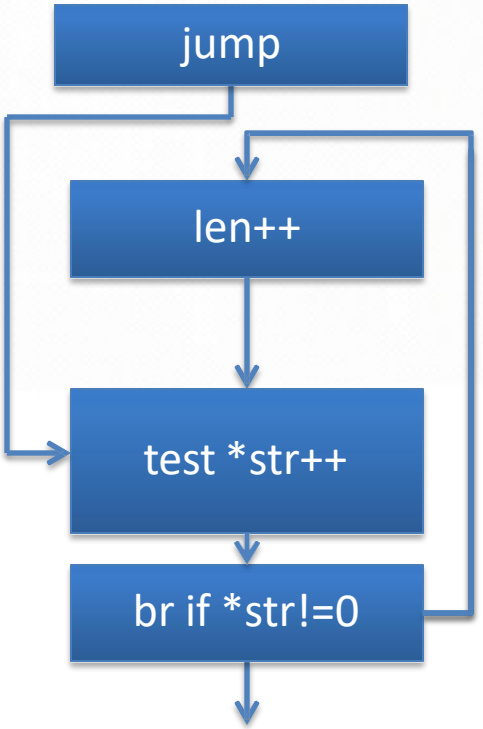
```
while (cond)
{
   loop-body;
}
```

WHILE Loop

```
; R0 = pointer to input string (str)
; R3 = length counter (len)
; R2 = temporary register to hold character (tmp_char)

ANDS R3, R3, #0        ; Set R3 = 0 → initialize length to zero (prologue)
B test                 ; Jump to test condition before entering loop (pre-check)
loop:
   ADDS R3, R3, #1     ; Increment length counter
(len++)
 test:
    LDRB R2, [R0], #1   ; Load byte from [R0] into R2
 and increment R0 (tmp_char = *str++)
    CMP  R2, #0         ; Check if tmp_char is null
 terminator '\0'
    BNE  loop           ; If tmp_char != 0, repeat loop
  MOV  R0, R3           ; Move result (len) into R0 (as
 return value)
    BX   LR             ; Return to caller
```

```
int strlen(char* str) {
   int len = 0;
   while (*str++ != 0)
      len++;
   return len;
}
```



Exercise 3

Translate the below C code for counting the 1 bits in R0 into ARM assembly code, using the registers indicated by the variable names.

```
r3 = 1;
r1 = 0;
while (r3 != 0) {
   if ((r0 & r3) != 0) {
      r1 = r1 + 1;
   }
   r3 = r3 + r3;
}
```

Exercise 4

```
if (a > b) {
    do {
        a = a - b;
    } while (a >= 55);
    --b;
} else {
    b = b + 44;
}
```

- Your Task:

Translate the above C code into ARM assembly, following these constraints:

a. You may use only R0–R3
b. Instruction Timings:
   MOV, SUB, ADD, CMP: 1 cycle
   Conditional branch (BNE, BLE, BGT, etc.): 3 cycles
   Unconditional branch (B, BX): 2 cycles

Optimization Goals:

a. Minimize total cycle cost
b. Avoid stack usage
c. Avoid unnecessary instructions or comparisons
d. Compact loop (as few lines as possible)

Final values of a and b must remain in R0 and R1 respectively.

Exercise 4- Part Two (Optional)

```
if (a > b) {
    do {
        a = a - b;
    } while (a >= 55);
    --b;
} else {
    b = b + 44;
}
```

- Your Task:

Translate the above C code into ARM assembly, following these constraints:

a. You may use only R0–R3

b. Instruction Timings:

MOV, SUB, ADD, CMP: 1 cycle

Conditional branch (BNE, BLE, BGT, etc.): 3 cycles

Unconditional branch (B, BX): 2 cycles

Optimization Goals:

a. Minimize total cycle cost

b. Avoid stack usage

c. Avoid unnecessary instructions or comparisons

d. Compact loop (as few lines as possible)

Final values of a and b must remain in R0 and R1 respectively.

Part Two-

Assume: Initial a = 200; Initial b = 30; Answer the following:

1. How many iterations of the do-while loop will execute?

2. What is the total number of machine cycles executed (based on the instruction costs)?