CprE 288 – Introduction to Embedded Systems
ARM Assembly Programming: Translating C Control Statements and Function Calls

Instructors:

Dr. Phillip Jones

Announcements

- **Exam 3: Friday 7/6 (last day of class)**
- **Final Projects**
  - Projects: Mandatory Demos Friday , 7/6 (11am – 3pm)
  - Reminder, Lab attendance is mandatory: -10 points from final project for each lab session you miss
- **Quiz 7** (Thuesday, 7/3): ARM Procedure Call Standard readings.
- Reading for the next few weeks
  - **Textbook:** Chapter 2.1-2.3, and 2.6.1-2.6.2
  - **Textbook:** Chapter 4.1 – 4.3
  - **Assemble ARM instruction set manual:**
    - Preface, Chapter 3, Chapter 4
  - **ARM Procedure Call Standard:**
    - Sections: 5, 7.1.1, 7.2.

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

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 execute 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 implementing 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 **<u>not met</u>**, 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.

<u>Example:</u>

```
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 ==, !=, >, $\geq$, <, $\leq$.
  - **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 MOVS that updates the Status Register

Compare (register)

CMP: Subtracts two registers. Updates Status register

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

Operands: $0 \leq n \leq 15$, $0 \leq m \leq 15$

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

Binary Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	1	0	1	1		Rn		

`shift` omitted

imm3:imm2= 0

type = 0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
(0)	imm3			1	1	1	1	imm2		type		Rm			

Note: if `shift` is not omitted, then Rm is shifted based on the encodings giving in section 4.3.1 in ARM Instruction Set

Compare (Immediate)

CMP: Subtract constant from reg. Update Status Register.

Syntax: CMP Rn, #const

Operands: 0 ≤ n ≤ 15, 0 ≤ const ≤ 255

Operations: Rn − #const, PC ← PC+4

Binary Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	1	0	1	1			Rn	

0 ≤ const ≤ 255

const=
i:imm3:imm8

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0		imm3		1	1	1	1				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:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	0	0	1			Rn	

`shift` omitted

imm3:imm2= 0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
(0)		imm3		1	1	1	1		imm2		type			Rm	

type = 0

Note: if `shift` is not omitted, then Rm is shifted based on the encodings giving in section 4.3.1 in ARM Instruction Set

Test (Immediate)

TST: Bitwise AND reg with constant. Update Status Register

Syntax: TST Rn, #const

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

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

Binary Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	0	0	1			Rn	

$0 \leq const \leq 255$

const=
i:imm3:imm8

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0		imm3		1	1	1	1				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 <u>Signed</u> values
  - **BHI/BHS/BLO/BLS:** comparing <u>Unsigned</u> 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:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	cond			imm6						

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	J1	0	J2	imm11										

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

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

Exercises: Write a sequence of instructions

Branch to label if a < b, a and b are variables of "signed int" type

Branch to label if a >= b, a and b are vars of "unsigned int" type

Branch to label if a == b, a and b are "int" type variables

Exercises

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
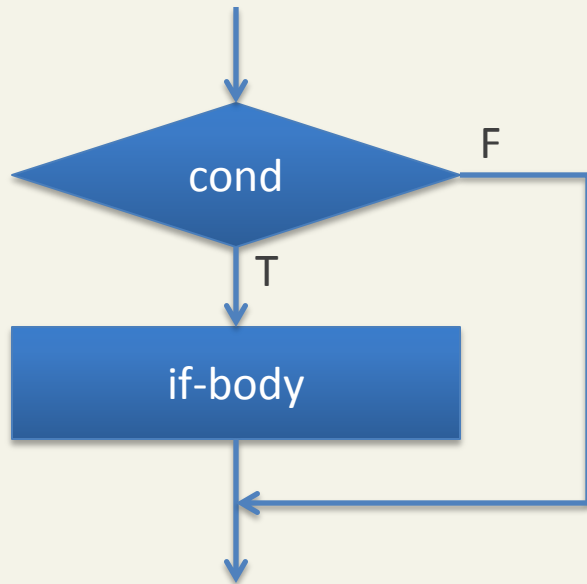
Linear Code Layout
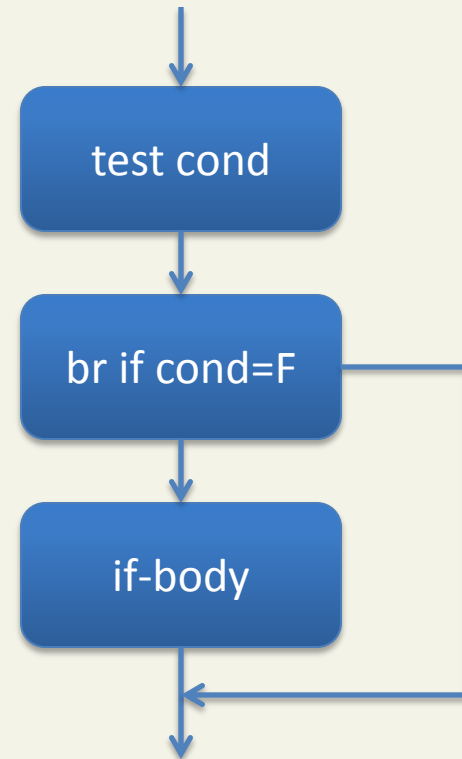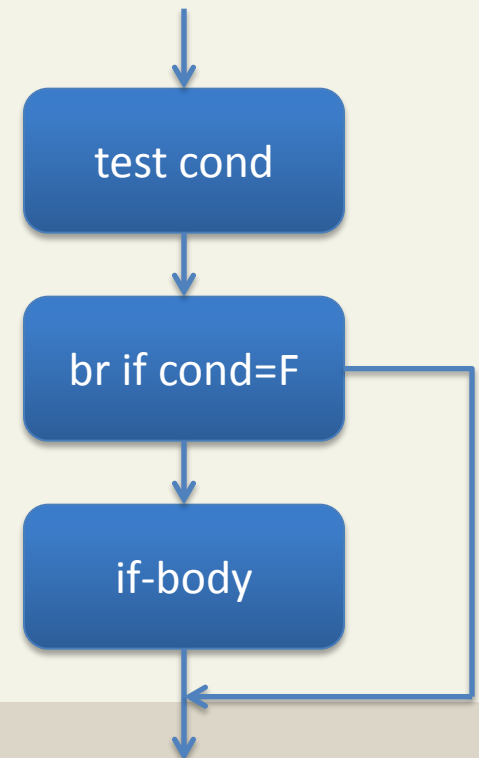
If-Statement: Structure

```
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:
```
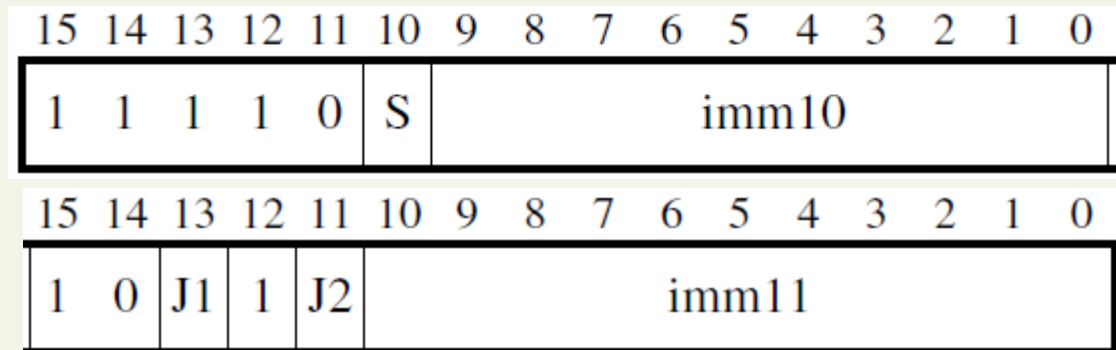
test cond

br if cond=F

if-body

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:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10									

I1 = NOT(J1 EOR S)

I2 = NOT(J2 EOR S)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	J1	1	J2	imm11										

IF-Else Statement

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

Example:
```
int max, a, b;
if (a < b)
  max = b;
else
  max = a;
```
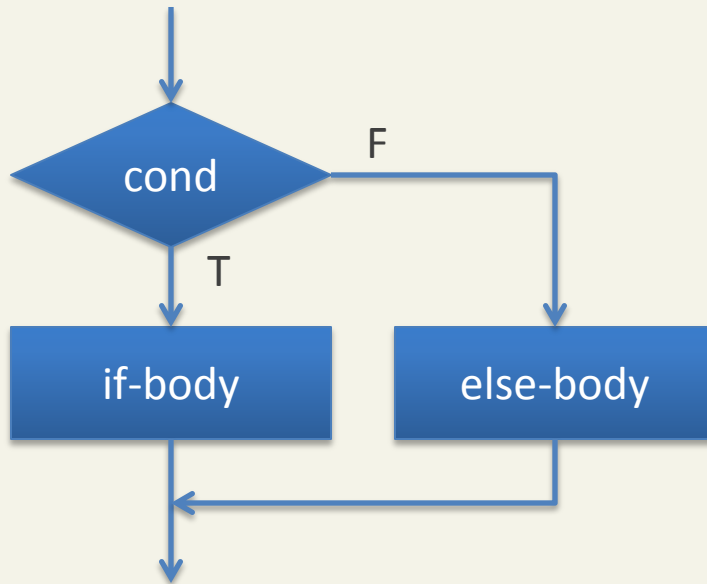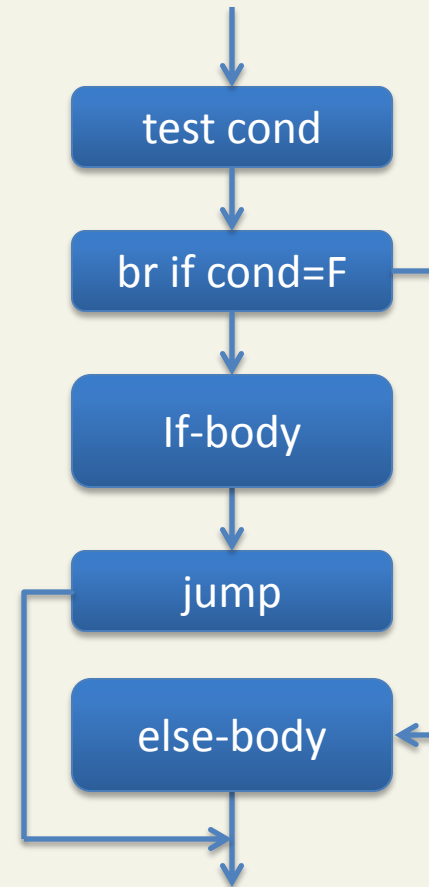
If-Else Statement: Structure

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 ↓ (all signed)
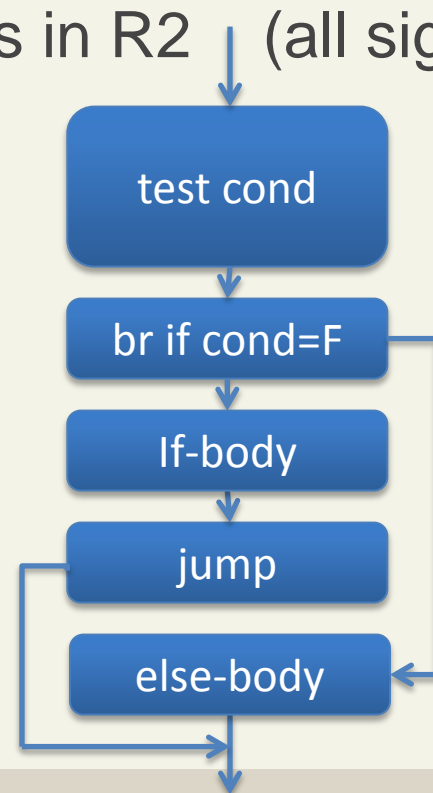
```
      CMP     R0, R1
      BGE     else

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

test cond

br if cond=F

If-body

jump

else-body

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 **<u>AND's</u>**, 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

Compound Condition (||)

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

;If only Boolean **OR's**, then complement only 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

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

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]`

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
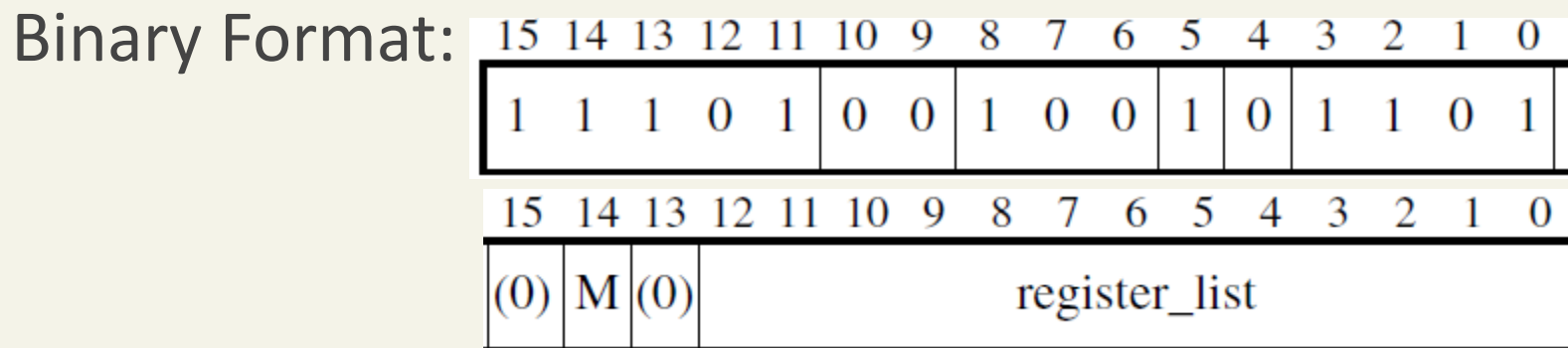 - E.g. Function args, local vars, return address, preserved values

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:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	0	0	1	0	1	1	0	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
(0)	M	(0)	register_list												

POP

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

Syntax: POP Rn

Operands: $0 \le n \le 12$, 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:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	0	1	1	1	1	0	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P	M	(0)	register_list												

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 got back to the Caller

- Assembly Instructions used
  - `BL label`: Branch with link, places PC + 4 into the LR (R14), and `PC ← label`. Where label is the name of the function being called.
  - `BX LR`: Branch eXchange, places the vale of LR (R14) into the PC. i.e. `PC ← LR`. Assuming LR (R14) contains the return address.
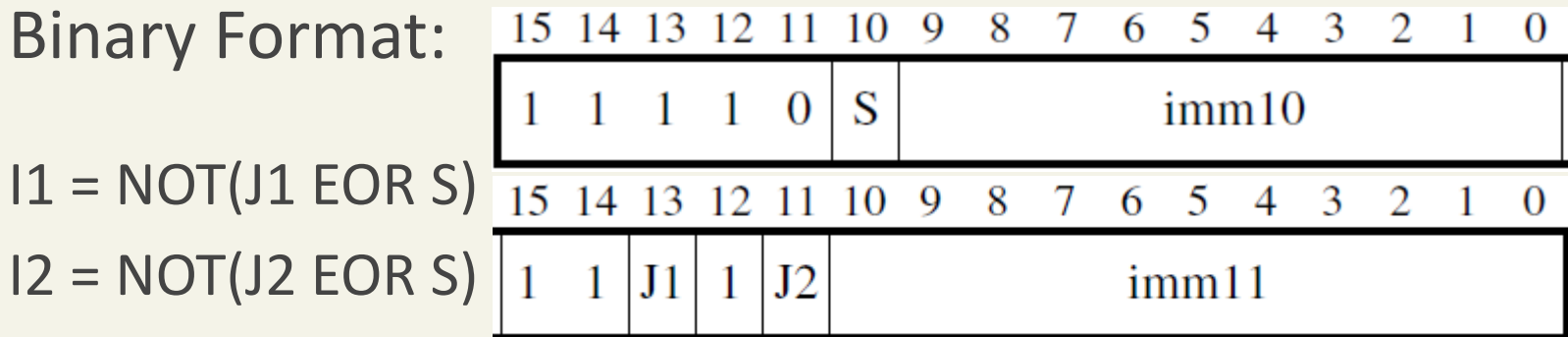
Branch with Link

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:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10									

I1 = NOT(J1 EOR S)

I2 = NOT(J2 EOR S)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	J1	1	J2	imm11										

Branch eXchange

BX: Branch to address located in a register

Syntax: BX Rn

Operands: 0 ≤ n ≤ 15

Operation: PC ← Rn

Binary Format:

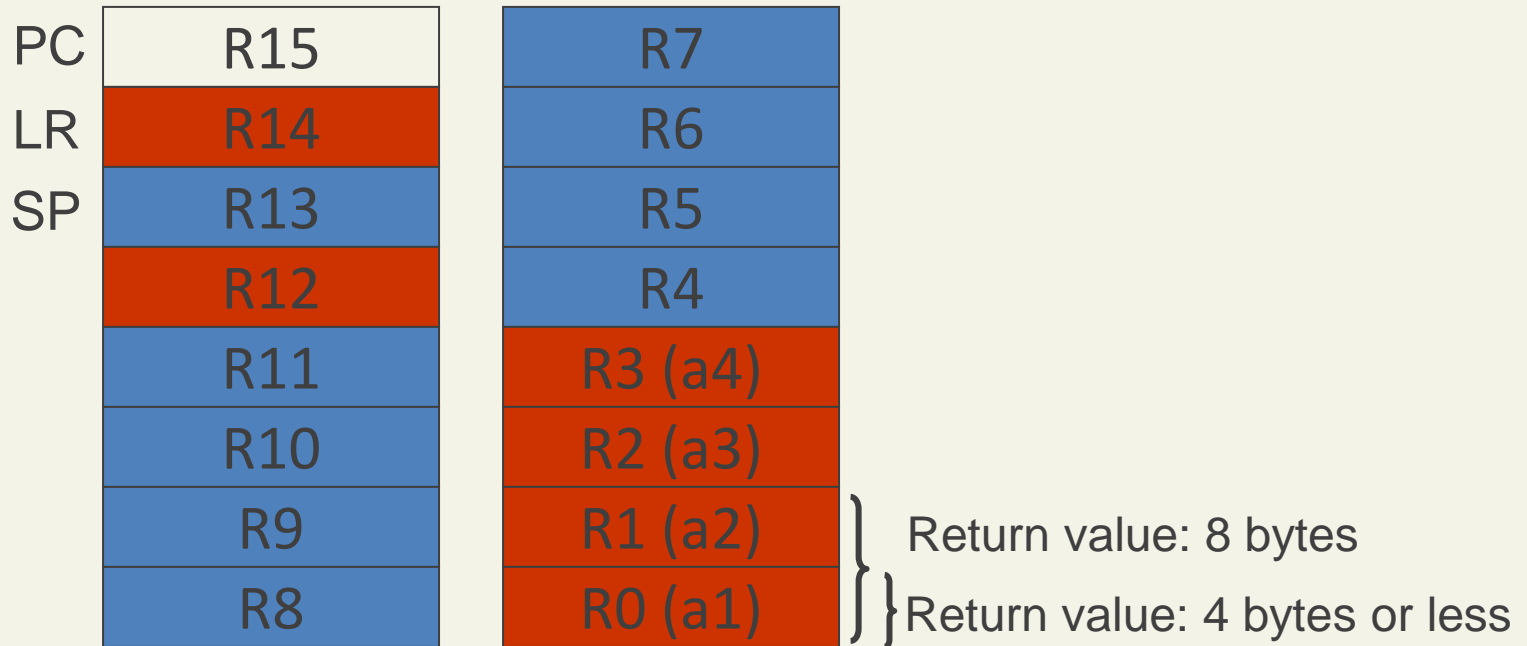| 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	0			Rm		(0)	(0)	(0)

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

Function Call: Example
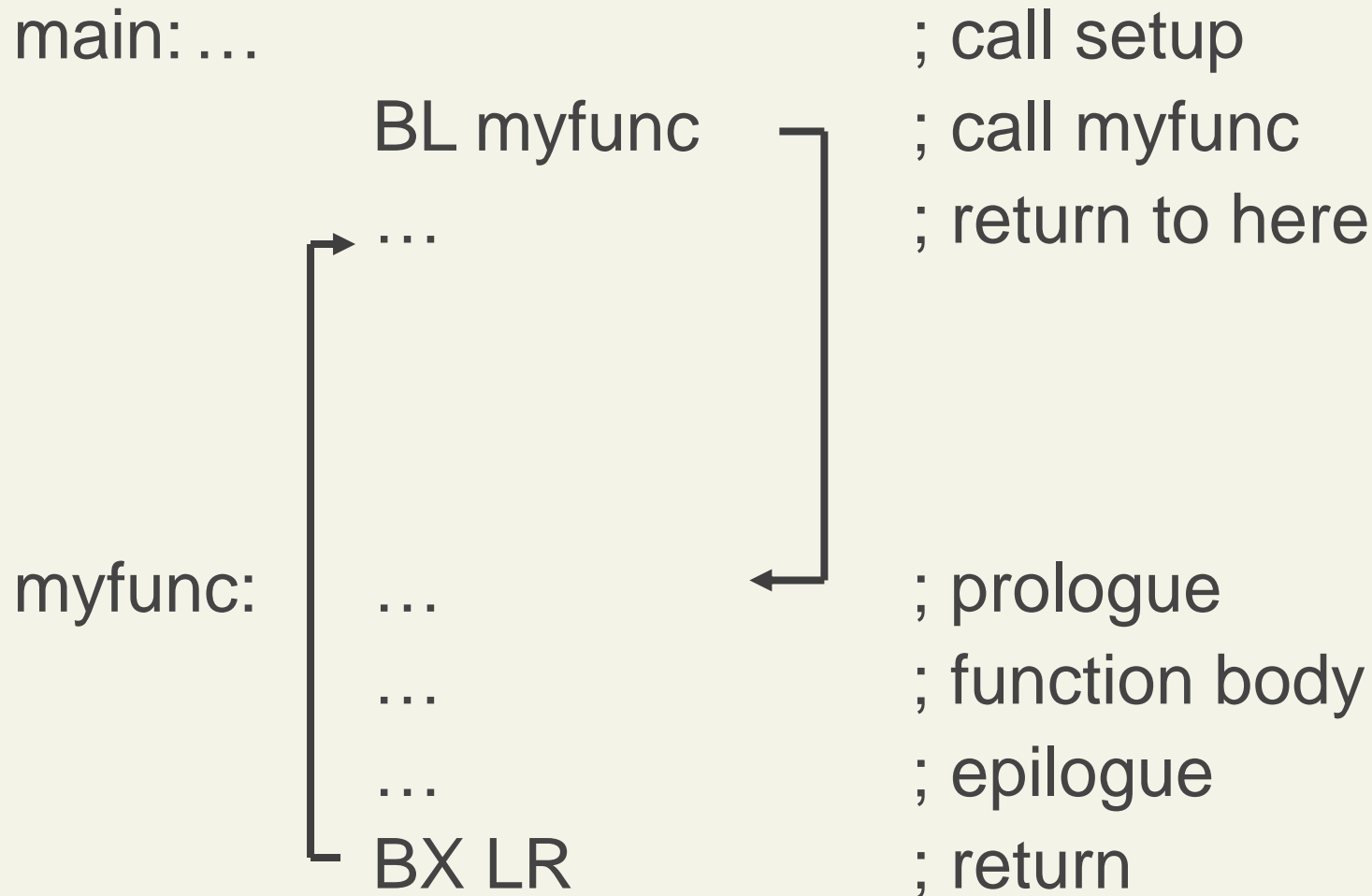
main: …   ; call setup

      BL myfunc   ; call myfunc

      …   ; return to here

myfunc: …   ; prologue

           …   ; function body

           …   ; epilogue

           BX LR   ; return

ARM Function Call Standard

PC	R15		R7
LR	R14		R6
SP	R13		R5
	R12		R4
	R11		R3 (a4)
	R10		R2 (a3)
	R9		R1 (a2) }Return value: 8 bytes
	R8		R0 (a1) }Return value: 4 bytes or less

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

```
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: Example

```
; z = max(x, y);
;  // x @ 0x1000_A000, y @ 0x1000_B000, z @ 0x1000_C000
my_func:
        ...  ; more instructions
  MOVW R2, 0xA000 ;get x address (global variable)
  MOVT R2, 0x1000
  MOVW R3, 0xB000 ;get y Address (global variable)
  MOVT R3, 0x1000 ;
  LDR R0, [R2, #0] ;load x ; 1st parameter of max
  LDR R1, [R3, #0] ;load y ; 2nd parameter of max

  PUSH LR    ;Save return address for my_func
  BL  max    ;call max(x,y)
  POP LR     ;Restore return address for my_func

  MOVW R2, 0xC000 ;get z address (global variable)
  MOVT R2, 0x1000
  STR R0 [R2, #0] ;store z ; save returned results
        ...    ; more instructions
  BX LR            ; Return to Caller
```

```
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
```

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

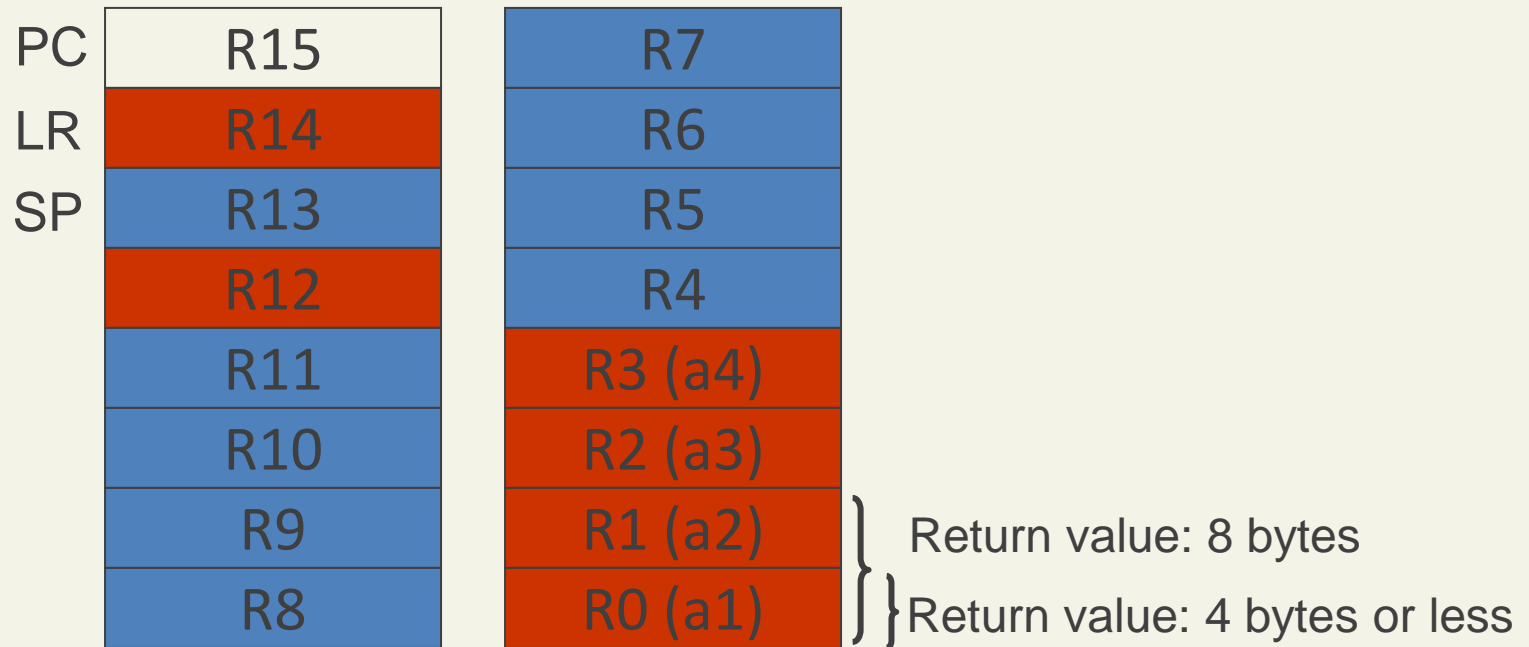How to share registers between Caller and Callee?

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.

ARM Function Call Standard

PC	R15		R7
LR	R14		R6
SP	R13		R5
	R12		R4
	R11		R3 (a4)
	R10		R2 (a3)
	R9		R1 (a2)
	R8		R0 (a1)

Return value: 8 bytes
Return value: 4 bytes or less

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)

Example

```
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

```
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
```

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?

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)

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

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