CprE 288 – Introduction to Embedded Systems
Course Review for Exam 3

Instructors:
Dr. Phillip Jones

Announcements

- Exam 3:
  - **See course website for day/time.**
- Exam 3 location: Our regular classroom
- Allowed Textbook, 1 page of notes, and a Calculator
  - Notes must be a paper copy (not electronic)

Topics

General architecture

Assembly programming

- Data Movement
  - Move constants to registers
  - Move data between registers
  - Move data between register and memory (Load/Store)
- Logic & Arithmetic
- Control Flow
  - Test/compare register(s), set status register flags
  - Choose the right branch
- Function call convention
  - Pass parameters and return values
  - Share registers between Caller and Callee
  - The Stack and its usage

General Purpose Register File

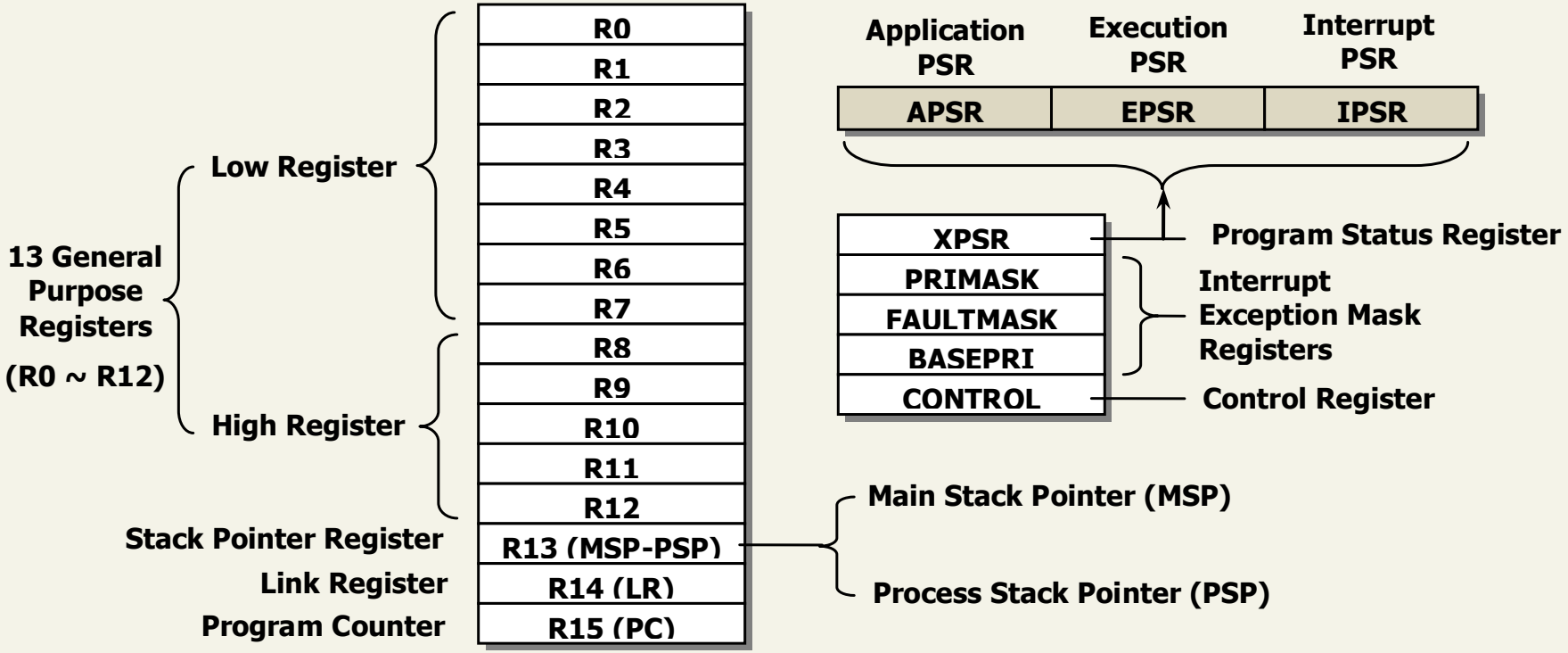


Figure 2.5 A structure block diagram of 21 registers in the Cortex®-M4 Core.

ARM: GP Registers (Cont.)

- 16 32-bit general purpose registers
  - Used for accessing SRAM
  - Used for storing function parameters
  - Used for instructions to execute operations on
- What is an 32-bit register.
  - Basically just 32 D-Flips connected together

Bit 31	Bit 30	Bit 5	…	…	Bit 2	Bit 1	Bit 0

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.

Figure 2.6 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

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*

Load/Store: Addressing modes

- Immediate offset
  - LDR Rt, [Rn, #K] Regular Imm Offset *Rt ← [Rn + K]*
  - LDR Rt, [Rn, #K]! Pre-Index Offset: *Rt ← [Rn + K], Rn ← Rn+K*
  - LDR Rt, [Rn], #K Post-Index Offset: *Rt ← [Rn], Rn ← Rn+K*
- Register offset
  - LDR Rt, [Rn, Rm, LSL #n] *Rt ← [Rn + (Rm << n)]*
- PC-Relative
  - LDR Rt, [PC, #K] *Rt ← [PC + K]*
- PUSH/POP Addressing mode
  - Loads/Stores a list of registers to the stack
- Multiple Register Addressing mode
  - Loads/Stores a list of registers
- Exclusive Addressing mode
  - Used to guarantee a single source is accessing a memory

Arithmetic Instruction

Brief overview of arithmetic instructions

Addition: ADD, ADC, ADDW

Subtraction: SUB, SBC

Logic: AND, ORR, EOR

Shift: LSL, LSR, ASR, ROR

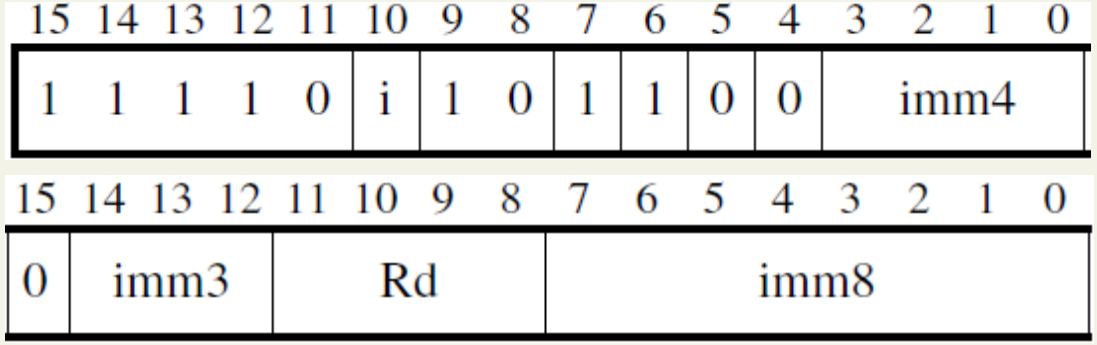
Compliments: NEG

Multiplication: MUL, SMULL, UMULL

Encoding Assembly Instructions

Given assembly instructions and there general encoding, provide their specific their binary encoding.

For example, given the general encoding of **`MOVT Rd, K`**.
Where K is encoded as: K = imm4:i:imm3:imm8



What is the specific binary encoding for **`MOVT R3,0xFF33`**

C to Assembly: Example Questions

```
char ch1 = 0x30;
char ch2 = 0x40;
int a = 0x1010;

ch1 = ch2;

a = a + a;
```

Example Questions

```
signed char ch1;
signed char ch2;
signed char flag;
int a;
int b;
signed char *pch;
int *pint;
```

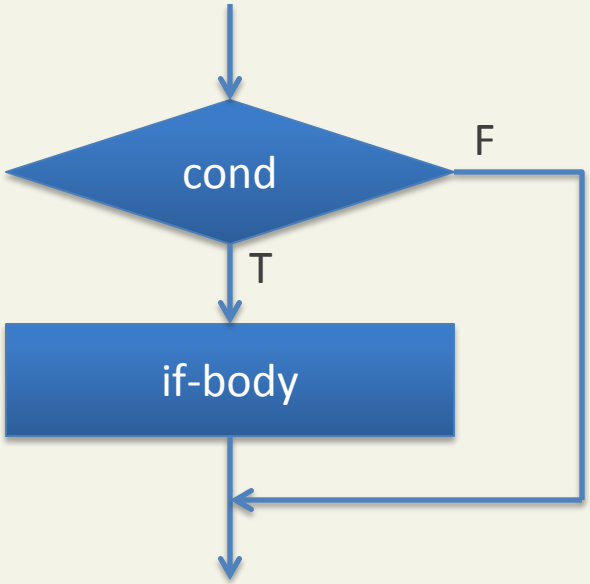
```
*pch = ch1;

a = *pint;

pint = &b;

a = a * b;

ch1 = ch1 & ch2;
```

Table of Branch Conditions

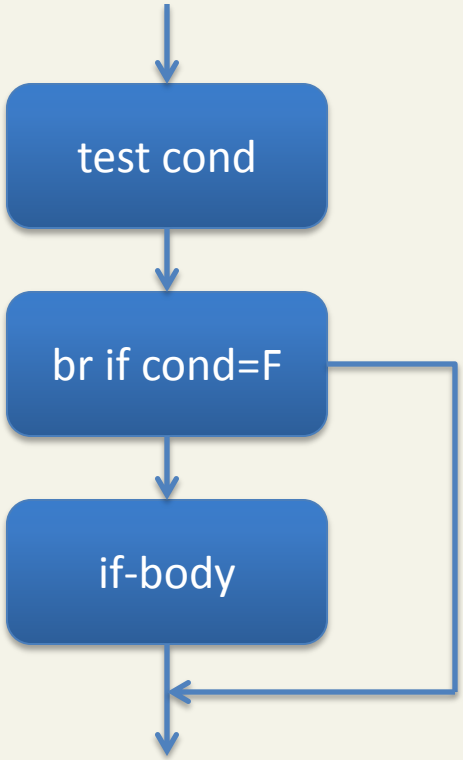
Condition Encoding (cond)	Branch Type	Meaning	Status Flag State
0000	BEQ	Equal	Z = 1
0001	BNE	Not equal	Z = 0
0010	BHS	Higher or Same (Unsigned)	C = 1
0011	BLO	Lower (Unsigned)	C = 0
0100	BMI	Negative	N = 1
0101	BPL	Positive	N=0
0110	BVS	Overflow	V=1
0111	BCV	No overflow	V=0
1000	BHI	Higher (Unsigned)	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

If-Statement: Structure

Control and Data Flow Graph

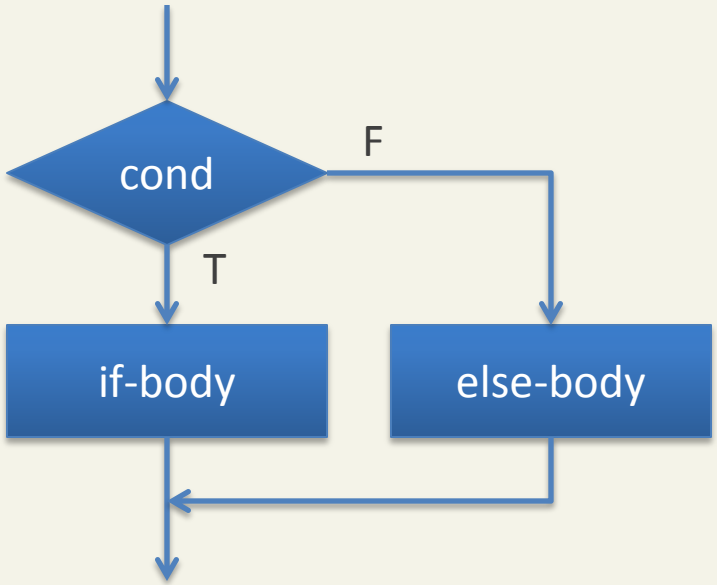


Linear Code Layout

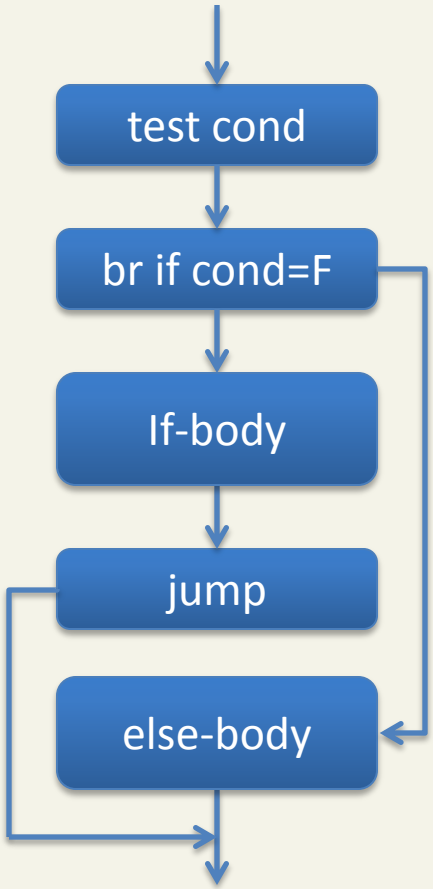


If-Else Statement: Structure

Control and Data Flow Graph



Linear Code Layout



Example Questions

```
int a, b, max;

if (a > b)
  max = a;
else
  max = b;
```

What if

```
  unsigned int a, b, max
```

Example Questions

```
int a;
signed char flag;

if (a <= 20)
  flag = 0;
else
  flag = 1;
```

Example Questions

```
int a, b;
signed char flag;

if (flag)
  a = b;
else
  b = a;
```

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

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)

; 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

Compound Condition (||)

if (a >= b || b< 10 || … || a > 20)

;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

ARM Function Call Standard: Sharing Registers

How to share registers between Caller and Callee?

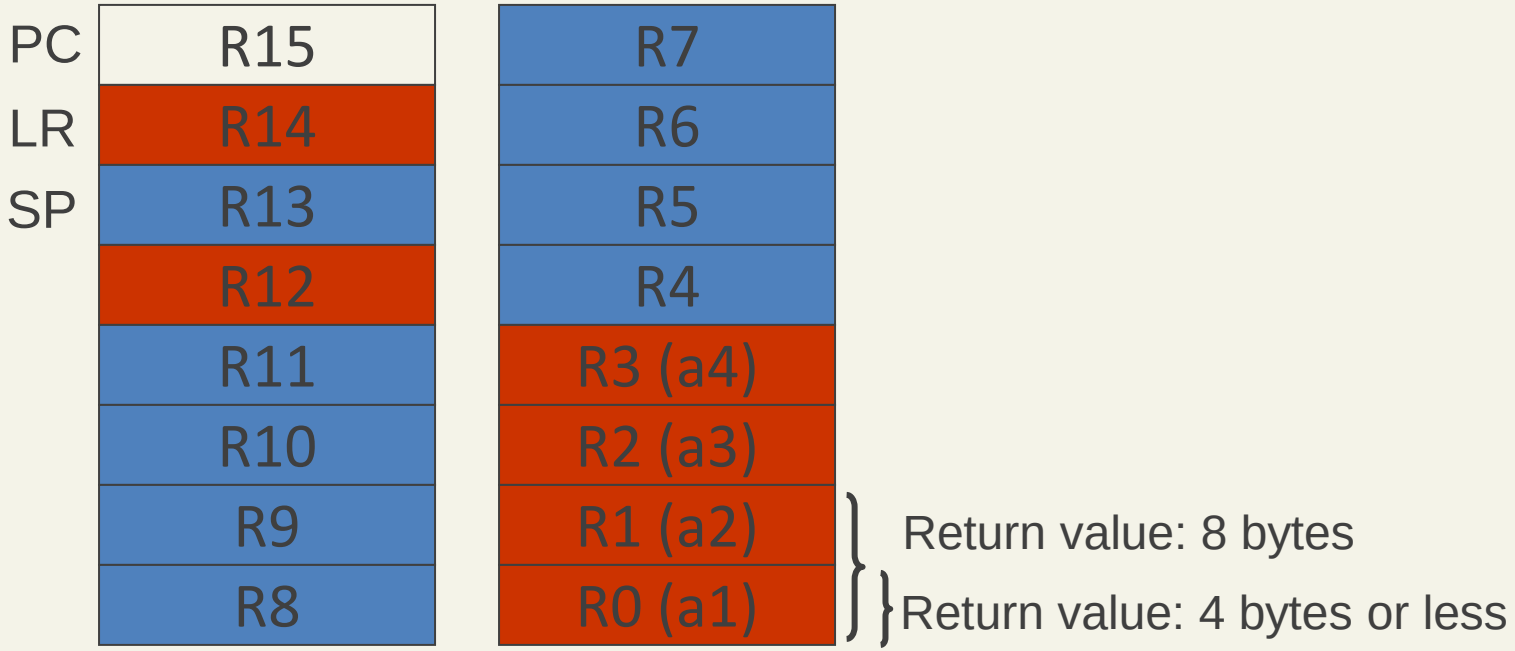
Non-volatile(Callee must preserve):R4-R11, SP

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.

ARM Function Call Standard



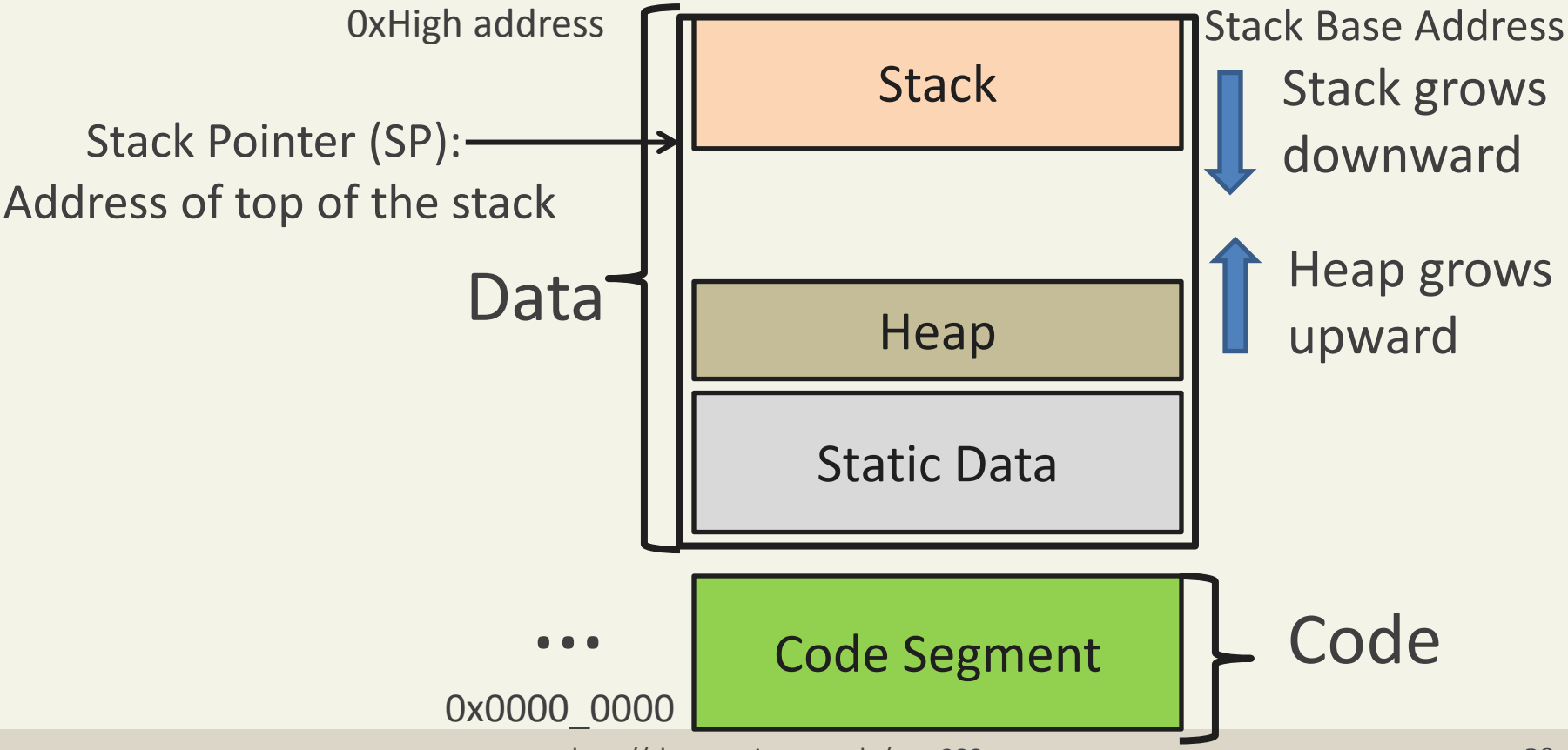
Non-volatile: **Callee** must preserve/restore if it uses

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

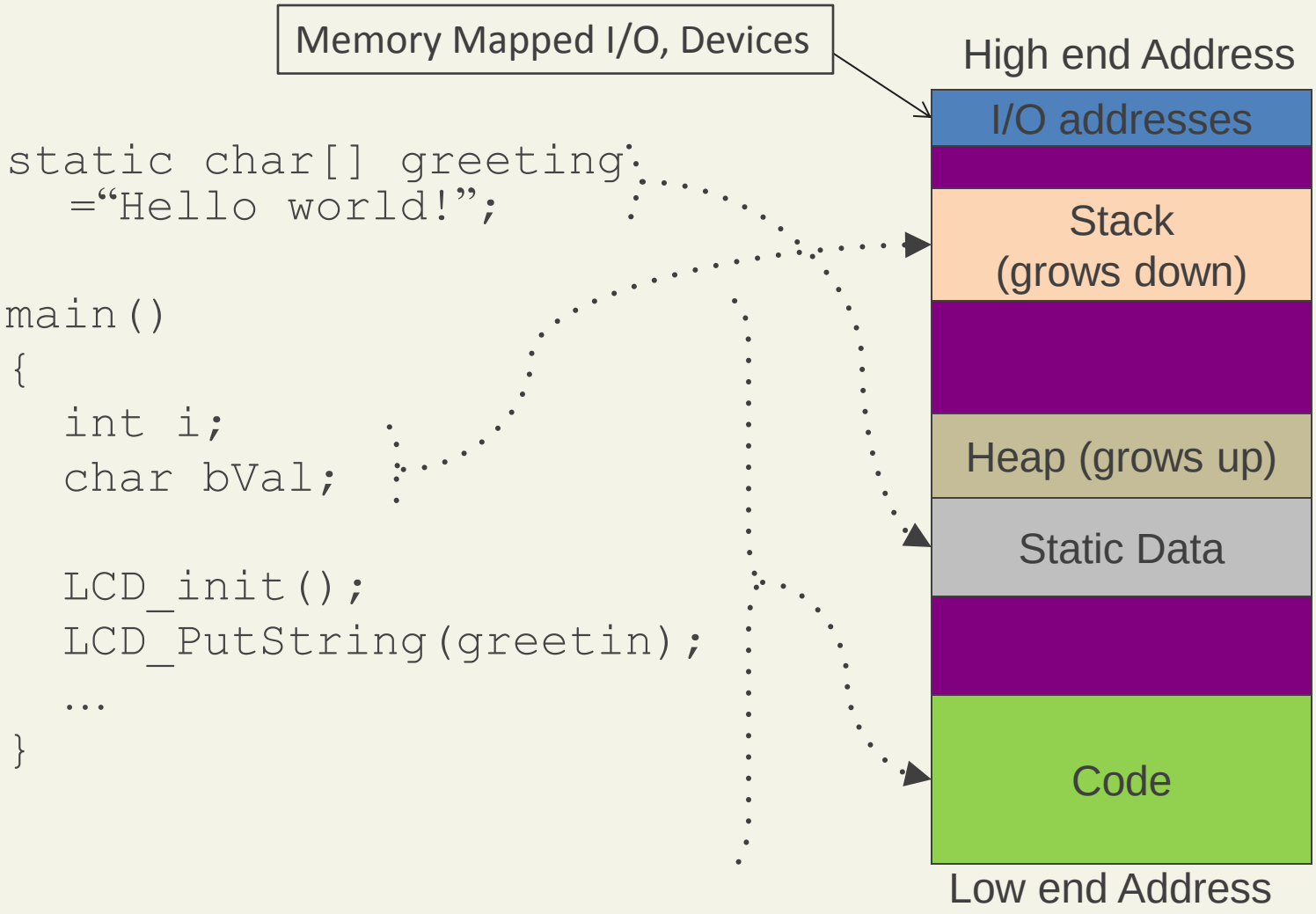
a: Function argument (Assuming each param is 4 bytes or less)

ARM: Typical Logical Memory Layout

- Static Data (e.g. Global vars)
- Stack (e.g. local vars, function args & return value, return address)
- Heap (e.g. dynamically allocated memory: malloc, new)
- Code (Code may be in the same or a separate memory device)

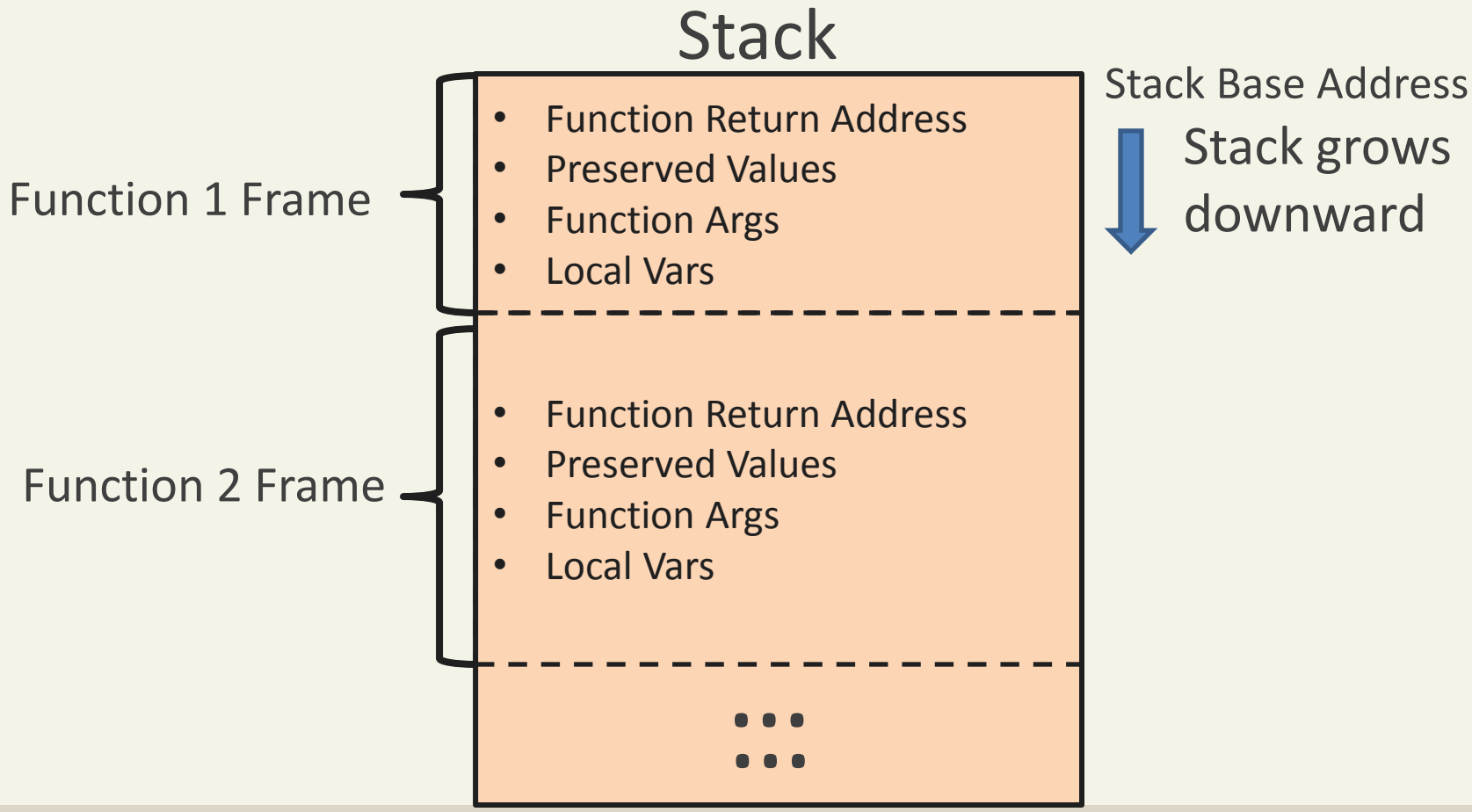


Example Memory Layout



ARM: Typical Function Stack-Frame

- Stack is typically divided into a number of function stack-frames.
- Stack-Frame: Contains information associated with a function call
  - Function args, local vars, return address, preserved values



Function and Stack

Function Frame: Local storage for a function

Example: 1. A is called; 2. A calls B; 3. B calls C; 4. C returns

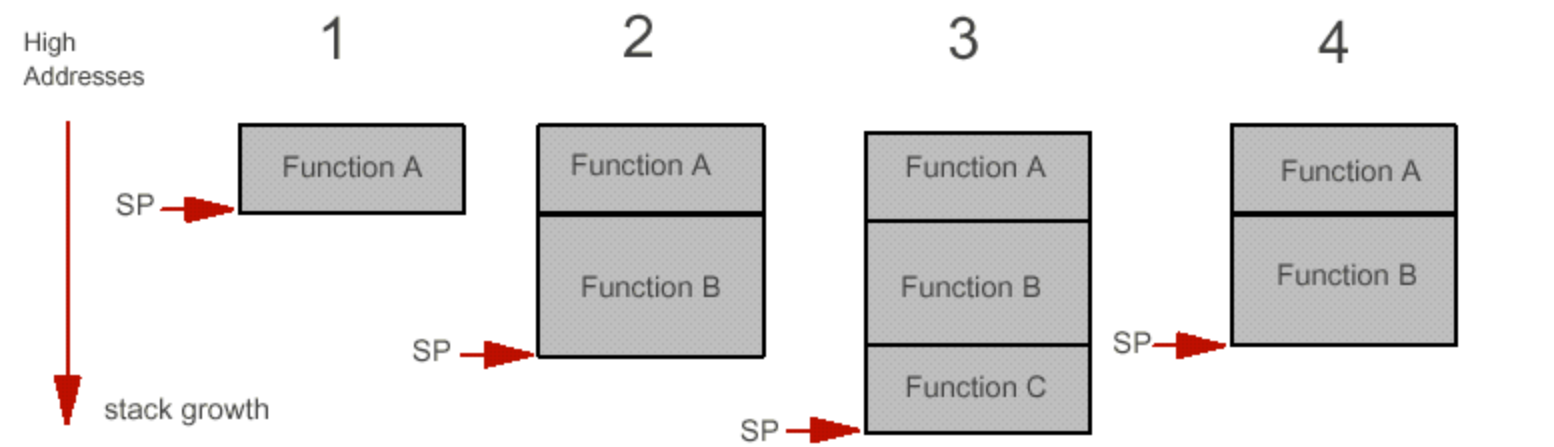
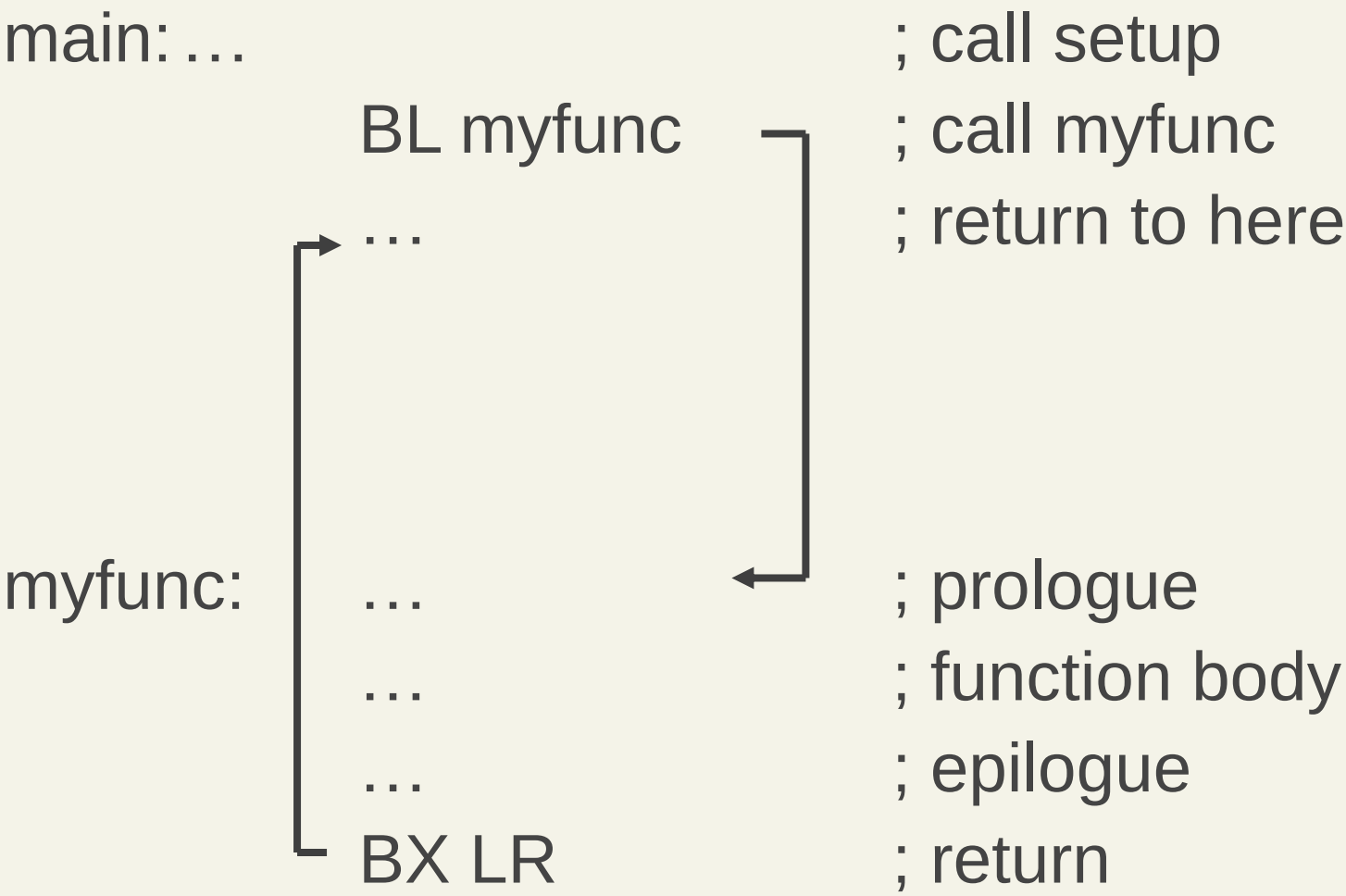


Figure 1 - Stack Frame creation and destruction

Function Call: Example



Calling a Func call vs. Implementing a Func

Know the difference between how to make a function call, and how to implement a function.

Example Question

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

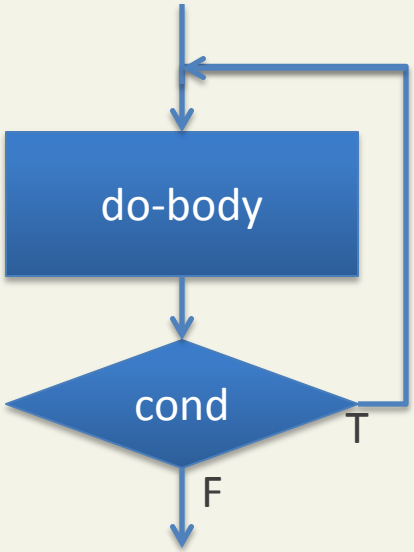
main:

```
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
```

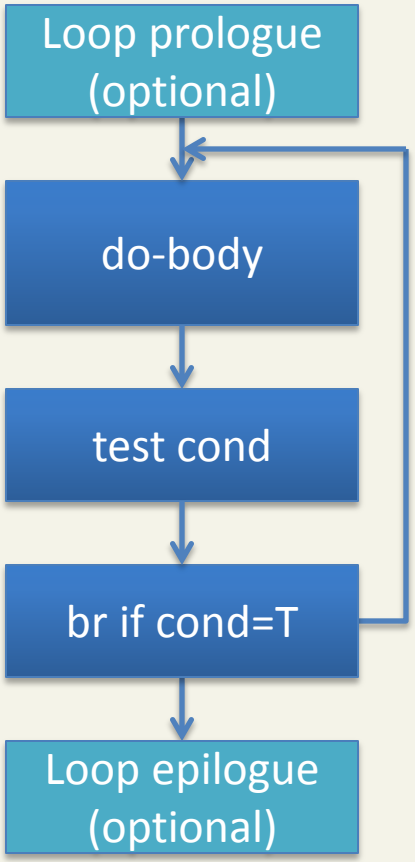
Question: Is it not necessary to push/pop R5?

DO-WHILE Loop

Control and Data Flow Graph

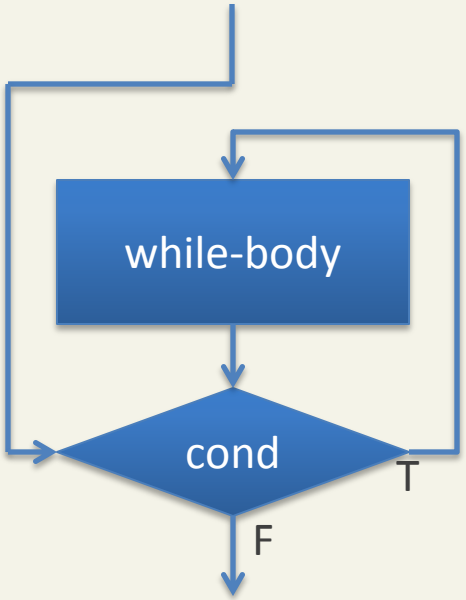


Linear Code Layout

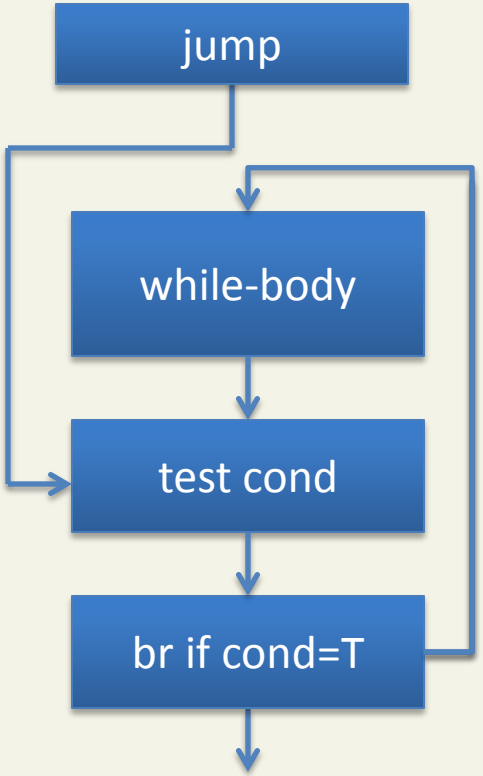


WHILE Loop

Control and Data Flow Graph



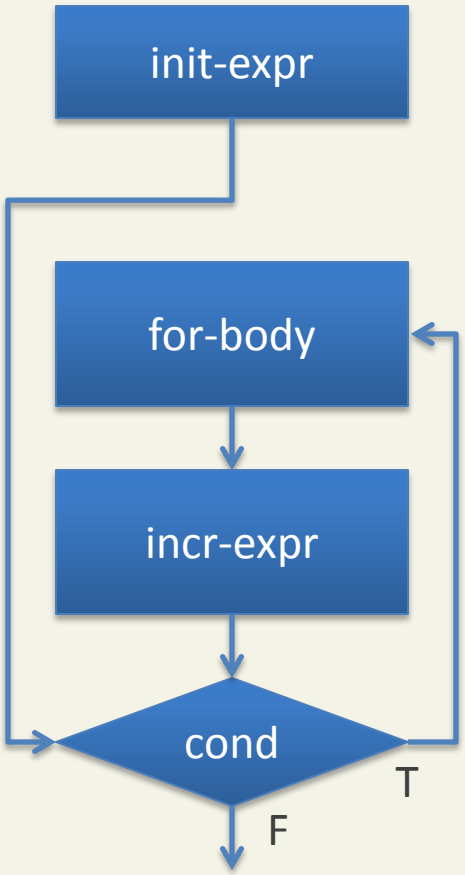
Linear Code Layout



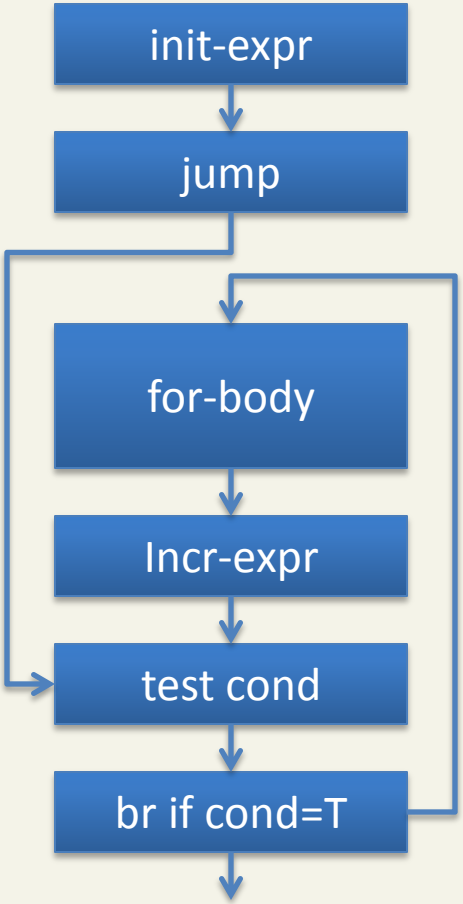
(optional prologue and epilogue not shown)

FOR Loop

Control and Data Flow Graph



Linear Code Layout



(optional prologue and epilogue not shown)

Example Questions

```
// Copy the contents of array X into array Y.
// Both arrays have N elements.
void copyArray(int X[ ], int Y[ ], int N);
```

Example Questions

```
// Find out the maximum value of an
// array, return the value. The array
// has N elements
int maxOfArray(int X[ ], int N);
```