Basic Idea

- Large programs are hard to handle
  - We can break them to smaller programs
  - They are called subroutines
- Subroutines are called from the main program
- Writing subroutines
  - When should we jump? (use CALL)
  - Where do we return to? (use RETURN)
Subroutine

- A **subroutine** is a block of code that is **called** from different places from within a main program or other subroutines.

  - **Saves code space** in that the subroutine code does not have to be repeated in the program areas that need it;
    - Only the code for the subroutine call is repeated.

- A subroutine can have
  - **parameters** that control its operation
  - **local variables** for computation.

- A subroutine may pass a **return value** back to the caller.

- Space in data memory must be reserved for parameters, local variables, and the return value.
Subroutine

Without Subroutines

\[
\begin{align*}
&\text{instr}_1 \\
&\text{instr}_2 \\
&\text{instr}_a1 \\
&\text{instr}_a2 \\
&\ldots \\
&\text{instr}_a n \\
&\text{instr}_3 \\
&\text{instr}_4 \\
&\text{instr}_5 \\
&\text{instr}_a1 \\
&\text{instr}_a2 \\
&\ldots \\
&\text{instr}_a n \\
&\text{instr}_6 \\
&\text{instr}_7 \\
&\ldots
\end{align*}
\]

With Subroutines

\[
\begin{align*}
&\text{Caller} \\
&\text{instr}_1 \\
&\text{instr}_2 \\
&\text{call sub} \\
&\text{instr}_3 \\
&\text{instr}_4 \\
&\text{instr}_5 \\
&\text{call sub} \\
&\ldots \\
\end{align*}
\]

\[
\begin{align*}
&\text{Callee} \\
&\text{instr}_a1 \\
&\text{instr}_a2 \\
&\ldots \\
&\text{instr}_a n \\
&\text{return}
\end{align*}
\]

Replicated instruction sequence as a subroutine
Using Subroutines

• When using subroutines we need to know the following:
  • Where is the NEXT instruction’s address
  • How to remember the RETURN address

• Subroutines are based on MPU instructions and use STACK

Let’s Review Stacks and then we come back to subroutines!
Stack

- Temporary memory storage space used during the execution of a program
- Used by MPU
- Stack Pointer (SP)
  - The MPU uses a register called the stack pointer, similar to the program counter (PC), to keep track of available stack locations.
Data Storage via the Stack

• The word ‘stack’ is used because storage/retrieval of words in the stack memory area is the same as accessing items from a stack of items.

• Visualize a stack of boxes. To build a stack, you place box A, then box B, then box C
  • Notice that you only have access to the last item placed on the stack (the Top of Stack – TOS). You retrieve the boxes from the stack in reverse order (C then B then A). A stack is also called a LIFO (last-in-first-out) buffer (similar to a Queue)

GENERALLY set in SRAM 0x20000600 Area
Instructions to Store and Retrieve Information from the Stack

- **PUSH**
  - Increment the memory address in the stack pointer (by one) and stores the contents of the counter (PC+2) on the top of the stack

- **POP**
  - **Discards** the address of the top of the stack and decrement the stack pointer by one
Example

What is the value of PC (PC=PC+2), and STKPTR as you execute each line?

| nPC | TOS | STKPTR | REG |  | MOV R0, #2 |
|-----|-----|--------|-----| | MOV R0, #2 |
| 22  | 0   | 0      | 00  | 0001 | push |
| 24  | 0   | 0      | 20  | 0002 | push |
| 26  | 26  | 1      | 20  | 0003 | pop  |
| 28  | 28  | 2      | 20  | 0004 |     |
| 2A  | 26  | 1      | 20  | 0005 |     |
| 2C  | 0   | 0      | 20  | 0006 | pop  |
|     |     |        |     | 0007 |     |
Stack Growth Convention:
Ascending vs Descending

**Descending stack**: Stack grows towards low memory address

**Ascending stack**: Stack grows towards high memory address
Cortex-M Stack

- stack pointer (SP) = R13
- Cortex-M uses **full descending stack**
- stack pointer
  - decremented on **PUSH**
  - incremented on **POP**
  - SP starts at **0x20000600** for STM32F401xe devices
Stack

**PUSH** \{Rd\}

- \( SP = SP-4 \) → descending stack
- \((*SP) = Rd\) → full stack

**Push multiple registers**

```
<table>
<thead>
<tr>
<th>LM</th>
<th>HM</th>
<th>They are equivalent.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH {r6, r7, r8}</td>
<td>PUSH {r8, r7, r6}</td>
<td>PUSH {r8}</td>
</tr>
</tbody>
</table>
```

- The order in which registers listed in the register list does not matter.
- When pushing multiple registers, these registers are automatically sorted by name and the lowest-numbered register is stored to the lowest memory address, *i.e.* is stored last.
Example: swap R1 & R2

PUSH {R1}
PUSH {R2}
POP  {R1}
POP  {R2}
Example: swap R1 & R2

R1
0x11111111
R2
0x22222222
R13 (SP)
0x200001FC

memory
Address
0x20000200
0x200001FC
0x200001F8

PUSH {R1}
PUSH {R2}
POP {R1}
POP {R2}
Example: swap R1 & R2

```
PUSH {R1}
PUSH {R2}
POP  {R1}
POP  {R2}
```
Example: swap R1 & R2

PUSH {R1}
PUSH {R2}
POP  {R1}
POP  {R2}
Example: swap R1 & R2

PUSH {R1}
PUSH {R2}
POP  {R1}
POP  {R2}
Quiz

Are the values of R1 and R2 swapped?

PUSH {R1, R2}
POP  {R2, R1}

Descending SP

Highest Memory
  PUSH {r2}
  PUSH {r1}

Lowest Memory
Quiz

Are the values of R1 and R2 swapped?

PUSH {R1, R2} ; Push R2 then R1
POP   {R2, R1} ; POP R1 then R2

Answer: No.

But you can:
PUSH {R1, R2}
PUSH {R2}
PUSH {R2} ; POPing out of order!
POP   {R1}

or
PUSH {R1}
PUSH {R2}
PUSH {R2}
PUSH {R1, R2}
Back to Subroutine

• In general, a subroutines, also called a function or a procedure,
  • single-entry, single-exit
  • Return to caller after it exits

• In ARM:
  • When a subroutine is called, the Link Register (LR) holds the memory address of the next instruction to be executed after the subroutine exits.
Link Register

Link Register (LR) holds the return address of the current subroutine call.
Call a Subroutine

<table>
<thead>
<tr>
<th>Caller Program</th>
<th>Subroutine/Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV r4, #100</td>
<td>foo PROC</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>BL foo</td>
<td>MOV r4, #10 ; foo changes r4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>ADD r4, r4, #1</td>
<td>BX LR</td>
</tr>
<tr>
<td>; r4 = 101, not 11</td>
<td>ENDP</td>
</tr>
</tbody>
</table>
Calling a Subroutine

**BL label**

- Step 1: \( LR = PC + 4 \) (Return address)
- Step 2: \( PC = \text{label} \) (Next instruction)

**Notes:**
- \( \text{label} \) is name of subroutine
- Compiler translates label to memory address
- After call, LR holds return address (the instruction following the call)
Exiting a Subroutine

**BX LR**

- PC = LR

**Caller Program**

```
MOV r4, #100
...
BL foo
...
```

**Subroutine/Callee**

```
foo PROC
...
MOV r4, #10
...
...
BX LR
ENDP
```
## ARM Procedure Call Standard

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
<th>Subroutine Preserved</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>Argument 1 and return value</td>
<td>No</td>
<td>If return has 64 bits, then r0:r1 hold it. If argument 1 has 64 bits, r0:r1 hold it.</td>
</tr>
<tr>
<td>r1</td>
<td>Argument 2</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td>Argument 3</td>
<td>No</td>
<td>If the return has 128 bits, r0-r3 hold it.</td>
</tr>
<tr>
<td>r3</td>
<td>Argument 4</td>
<td>No</td>
<td>If more than 4 arguments, use the stack</td>
</tr>
<tr>
<td>r4</td>
<td>General-purpose V1</td>
<td>Yes</td>
<td>Variable register 1 holds a local variable.</td>
</tr>
<tr>
<td>r5</td>
<td>General-purpose V2</td>
<td>Yes</td>
<td>Variable register 2 holds a local variable.</td>
</tr>
<tr>
<td>r6</td>
<td>General-purpose V3</td>
<td>Yes</td>
<td>Variable register 3 holds a local variable.</td>
</tr>
<tr>
<td>r7</td>
<td>General-purpose V4</td>
<td>Yes</td>
<td>Variable register 4 holds a local variable.</td>
</tr>
<tr>
<td>r8</td>
<td>General-purpose V5</td>
<td>YES</td>
<td>Variable register 5 holds a local variable.</td>
</tr>
<tr>
<td>r9</td>
<td>Platform specific/V6</td>
<td>No</td>
<td>Usage is platform-dependent.</td>
</tr>
<tr>
<td>r10</td>
<td>General-purpose V7</td>
<td>Yes</td>
<td>Variable register 7 holds a local variable.</td>
</tr>
<tr>
<td>r11</td>
<td>General-purpose V8</td>
<td>Yes</td>
<td>Variable register 8 holds a local variable.</td>
</tr>
<tr>
<td>r12 (IP)</td>
<td>Intra-procedure-call register</td>
<td>No</td>
<td>It holds intermediate values between a procedure and the sub-procedure it calls.</td>
</tr>
<tr>
<td>r13 (SP)</td>
<td>Stack pointer</td>
<td>Yes</td>
<td>SP has to be the same after a subroutine has completed.</td>
</tr>
<tr>
<td>r14 (LR)</td>
<td>Link register</td>
<td>No</td>
<td>LR does not have to contain the same value after a subroutine has completed.</td>
</tr>
<tr>
<td>r15 (PC)</td>
<td>Program counter</td>
<td>N/A</td>
<td>Do not directly change PC</td>
</tr>
</tbody>
</table>
Scratch registers, not saved by subroutine. Hold arguments/result

Callee must save them. Caller expects these values are retained.
### Preserve Runtime Environment via Stack

<table>
<thead>
<tr>
<th>Caller Program</th>
<th>Subroutine/Callee</th>
</tr>
</thead>
</table>
| ```                   | foo PROC
| MOV r4, #100            | PUSH {r4} ; preserve r4 |
| ...                    | ...                |
| BL foo                 | MOV r4, #10 ; foo changes r4 |
| ...                    | ...                |
| ADD r4, r4, #1 ; r4 = 101, not 11 | POP {r4} ; Recover r4 |
| ```                    | BX LR              |
| ENDP                   |                    |

**Callee must save R4 and beyond.**  
**Caller expects these values to be retained**
Stacks and Subroutines

Where is the ADD INSTRUCTION
Return Address saved?
Stacks and Subroutines

ADD INSTRUCTION

Saving PC1+4: ADD INSTRUCTION
Subroutine Calling Another Subroutine

Function MAIN

```
MAIN
  MOV R0, #2
  BL QUAD
ENDL ...
```

Function QUAD

```
QUAD
  PUSH {LR}
  BL SQ
  BL SQ
  POP {LR}
  BX LR
```

Function SQ

```
SQ
  MUL R0, R0
  BX LR
```
Subroutine Calling Another Subroutine

Function MAIN

```assembly
MAIN PROC
    MOV R0,#2
    BL QUAD
ENDL
ENDP
```

Function QUAD

```assembly
QUAD PROC
    PUSH {LR}
    BL SQ
    BL SQ
    POP {LR}
    BX LR
ENDP
```

Function SQ

```assembly
SQ PROC
    MUL R0,R0
    BX LR
ENDP
```
Example: $R0 = R0^4$

```assembly
MOV R0, #2
BL QUAD
B ENDL

SQ
MUL R0, R0
BX LR

QUAD
PUSH {LR}
BL SQ
BL SQ
POP {LR}
BX LR
ENDL
```

Descending Stack

- $R0 = 0x20000200$
- $R0 = 0x200001FC$
- $R0 = 0x200001F8$
- $R0 = 0x0800014C$
- $R0 = 0x08000150$
- $R0 = 0x08000140$
- $R0 = 0x08000148$
- $R0 = 0x08000154$
- $R0 = 0x08000158$
- $R0 = 0x0800015C$

PC
0x08000138

SP
0x20000200

LR

```
Example: \( R0 = R0^4 \)

```
MOV R0,#2
BL QUAD
B ENDL

MUL R0,R0
BX LR

PUSH {LR}
BL SQ
BL SQ
POP {LR}
BX LR
```

```
MOV R0,#2
BL QUAD
B ENDL

MUL R0,R0
BX LR
QUAD
PUSH {LR}
BL SQ
BL SQ
POP {LR}
BX LR
```
Example: \( R0 = R0^4 \)

- MOV R0, #2
- BL QUAD
- B ENDL

- SQ
  - MUL R0, R0
  - BX LR

- QUAD
  - PUSH {LR}
  - BL SQ
  - BL SQ
  - POP {LR}
  - BX LR
  - ENDL

---

Preserve Link Register (LR)
Example: \( R0 = R0^4 \)

```
MOV R0, #2
BL QUAD
B ENDL

SQ
MUL R0, R0
BX LR

QUAD
PUSH {LR}
BL SQ
BL SQ
POP {LR}
BX LR
ENDL
```

Note:
The address is saved

<table>
<thead>
<tr>
<th>Address 1</th>
<th>Address 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0800140</td>
<td>0x0200200</td>
</tr>
<tr>
<td>0x20001FC</td>
<td>0x20001F8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address 3</th>
<th>Address 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x200014C</td>
<td>0x0800144</td>
</tr>
<tr>
<td>0x0800148</td>
<td>0x080014C</td>
</tr>
<tr>
<td>0x0800150</td>
<td>0x0800154</td>
</tr>
<tr>
<td>0x0800158</td>
<td>0x080015C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address 5</th>
<th>Address 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x02001FC</td>
<td>0x0800138</td>
</tr>
<tr>
<td>0x080013C</td>
<td>0x0800140</td>
</tr>
<tr>
<td>0x0800144</td>
<td>0x0800148</td>
</tr>
<tr>
<td>0x0800150</td>
<td>0x0800154</td>
</tr>
<tr>
<td>0x0800158</td>
<td>0x080015C</td>
</tr>
</tbody>
</table>
Example: $R0 = R0^4$

MOV R0, #2
BL QUAD
B ENDL

SQ

MUL R0, R0
BX LR

QUAD
PUSH {LR}
BL SQ
BL SQ
POP {LR}
BX LR

ENDL

...
Example: $R0 = R0^4$

```
MOV R0, #2
BL QUAD
B ENDL

SQ
MUL R0, R0
BX LR

QUAD
PUSH {LR}
BL SQ
BL SQ
POP {LR}
BX LR

ENDL...
```
Example: \( R_0 = R_0^4 \)

MOV R0,#2
BL QUAD
B ENDL

MUL R0,R0
BX LR

PUSH {LR}
BL SQ
BL SQ
POP {LR}
BX LR
Example: \( R0 = R0^4 \)

- `MOV R0, #2`
- `BL QUAD`
- `B ENDL`
- `MUL R0, R0`
- `BX LR`
- `PUSH {LR}`
- `BL SQ`
- `BL SQ`
- `POP {LR}`
- `BX LR`
- `BL SQ`
- `BL SQ`
- `POP {LR}`
- `BX LR`
Example: \( R0 = R0^4 \)

<table>
<thead>
<tr>
<th>MOV R0, #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL QUAD</td>
</tr>
<tr>
<td>B ENDL</td>
</tr>
<tr>
<td>SQ</td>
</tr>
<tr>
<td>MUL R0, R0</td>
</tr>
<tr>
<td>BX LR</td>
</tr>
<tr>
<td>QUAD</td>
</tr>
<tr>
<td>PUSH {LR}</td>
</tr>
<tr>
<td>BL SQ</td>
</tr>
<tr>
<td>BL SQ</td>
</tr>
<tr>
<td>POP {LR}</td>
</tr>
<tr>
<td>BX LR</td>
</tr>
<tr>
<td>ENDL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RO 0x10</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQ</td>
</tr>
<tr>
<td>MUL R0, R0</td>
</tr>
<tr>
<td>BX LR</td>
</tr>
<tr>
<td>SP 0x200001FC</td>
</tr>
<tr>
<td>SR 0x08000158</td>
</tr>
<tr>
<td>LC 0x08000148</td>
</tr>
<tr>
<td>PC 0x08000140</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>xxxxxxxx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08000140</td>
</tr>
<tr>
<td>0x20000200</td>
</tr>
<tr>
<td>0x200001FC</td>
</tr>
<tr>
<td>0x200001F8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MOV R0, #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL QUAD</td>
</tr>
<tr>
<td>B ENDL</td>
</tr>
<tr>
<td>SQ</td>
</tr>
<tr>
<td>MUL R0, R0</td>
</tr>
<tr>
<td>BX LR</td>
</tr>
<tr>
<td>QUAD</td>
</tr>
<tr>
<td>PUSH {LR}</td>
</tr>
<tr>
<td>BL SQ</td>
</tr>
<tr>
<td>BL SQ</td>
</tr>
<tr>
<td>POP {LR}</td>
</tr>
<tr>
<td>BX LR</td>
</tr>
<tr>
<td>ENDL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RO 0x10</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQ</td>
</tr>
<tr>
<td>MUL R0, R0</td>
</tr>
<tr>
<td>BX LR</td>
</tr>
<tr>
<td>SP 0x200001FC</td>
</tr>
<tr>
<td>SR 0x08000158</td>
</tr>
<tr>
<td>LC 0x08000148</td>
</tr>
<tr>
<td>PC 0x08000140</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MOV R0, #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL QUAD</td>
</tr>
<tr>
<td>B ENDL</td>
</tr>
<tr>
<td>SQ</td>
</tr>
<tr>
<td>MUL R0, R0</td>
</tr>
<tr>
<td>BX LR</td>
</tr>
<tr>
<td>QUAD</td>
</tr>
<tr>
<td>PUSH {LR}</td>
</tr>
<tr>
<td>BL SQ</td>
</tr>
<tr>
<td>BL SQ</td>
</tr>
<tr>
<td>POP {LR}</td>
</tr>
<tr>
<td>BX LR</td>
</tr>
<tr>
<td>ENDL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RO 0x10</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQ</td>
</tr>
<tr>
<td>MUL R0, R0</td>
</tr>
<tr>
<td>BX LR</td>
</tr>
<tr>
<td>SP 0x200001FC</td>
</tr>
<tr>
<td>SR 0x08000158</td>
</tr>
<tr>
<td>LC 0x08000148</td>
</tr>
<tr>
<td>PC 0x08000140</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MOV R0, #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL QUAD</td>
</tr>
<tr>
<td>B ENDL</td>
</tr>
<tr>
<td>SQ</td>
</tr>
<tr>
<td>MUL R0, R0</td>
</tr>
<tr>
<td>BX LR</td>
</tr>
<tr>
<td>QUAD</td>
</tr>
<tr>
<td>PUSH {LR}</td>
</tr>
<tr>
<td>BL SQ</td>
</tr>
<tr>
<td>BL SQ</td>
</tr>
<tr>
<td>POP {LR}</td>
</tr>
<tr>
<td>BX LR</td>
</tr>
<tr>
<td>ENDL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RO 0x10</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQ</td>
</tr>
<tr>
<td>MUL R0, R0</td>
</tr>
<tr>
<td>BX LR</td>
</tr>
<tr>
<td>SP 0x200001FC</td>
</tr>
<tr>
<td>SR 0x08000158</td>
</tr>
<tr>
<td>LC 0x08000148</td>
</tr>
<tr>
<td>PC 0x08000140</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MOV R0, #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL QUAD</td>
</tr>
<tr>
<td>B ENDL</td>
</tr>
<tr>
<td>SQ</td>
</tr>
<tr>
<td>MUL R0, R0</td>
</tr>
<tr>
<td>BX LR</td>
</tr>
<tr>
<td>QUAD</td>
</tr>
<tr>
<td>PUSH {LR}</td>
</tr>
<tr>
<td>BL SQ</td>
</tr>
<tr>
<td>BL SQ</td>
</tr>
<tr>
<td>POP {LR}</td>
</tr>
<tr>
<td>BX LR</td>
</tr>
<tr>
<td>ENDL</td>
</tr>
</tbody>
</table>
Example: R0 = R0^4

MOV R0, #2
BL QUAD
B ENDL

MUL R0, R0
BX LR

PUSH {LR}
BL SQ
BL SQ
POP {LR}
BX LR

...
Example: \( R_0 = R_0^4 \)

```assembly
MOV R0, #2
BL QUAD
B ENDL

MUL R0, R0
BX LR

PUSH {LR}
BL SQ
BL SQ
POP {LR}
BX LR

...```

Recover Link Register (LR)
Example: \( R_0 = R_0^4 \)

- **MOV R0, #2**
- **BL QUAD**
- **B ENDL**

**SQUAD**
- **MUL R0, R0**
- **BX LR**

**QUAD**
- **PUSH {LR}**
- **BL SQ**
- **BL SQ**
- **POP {LR}**
- **BX LR**
- **ENDL**

- **MOV R0, #2**
- **BL QUAD**
- **B ENDL**

**RO**
- 0x10

**SP**
- 0x20000200

**LR**
- 0x08000140

**PC**
- 0x08000140

**QUAD**
- xxxxxxxx
  - 0x20000200
  - 0x08000140
  - 0x200001FC
  - 0x200001F8

- **MOV R0, #2**
  - 0x08000138
- **BL QUAD**
  - 0x0800013C
- **B ENDL**
  - 0x08000140
- **MUL R0, R0**
  - 0x08000144
- **BX LR**
  - 0x08000148
- **PUSH {LR}**
  - 0x0800014C
- **BL SQ**
  - 0x08000150
- **BL SQ**
  - 0x08000154
- **POP {LR}**
  - 0x08000158
- **BX LR**
  - 0x0800015C
Initializing the stack pointer (SP)

- Before using the stack, software has to define stack space and initialize the stack pointer (SP).
- The assembly file `startup.s` defines stack space and initialize SP.
Example: \( R0 = R0^4 \)

Find PC, LR, and SP values as you execute each line of code. What is happening?
Example: R0 = R0^4

Find PC, LR, and SP values as you execute each line of code. What is happening?
Let’s practice a little more....
Using Push & Pop
Calling a Subroutine

• Note that R0 and R1 can be used as the arguments
• PUSH and POP can be used to store the values
Calling a Subroutine from a different source file (sum3.s) from the __main
Practice

• Find PC, LR, and SP values as you execute each line of code. What is happening?