Computer organization and architecture

Lesson 8

Function calls

ARM uses the **branch and link instruction** (`BL`) to call a function and moves the link register to the PC (`MOV PC, LR`) to return from a function.

High-level

```
int main() {
  simple();
...
}

void simple() {
  return;
}
```

ARM assembly

```
MAIN ...
     ...
  BL SIMPLE ; call the simple
            ; function
  ...       ; next instruction

stop B stop

SIMPLE MOV PC, LR ; return
```

`BL` stores the return address of the next instruction (the instruction after `BL`) in the link register (`LR`), and branches to the target instruction.

PC and LR are alternative names for R15 and R14, respectively.

ARM is unusual in that PC is part of the register set.

So a function return can be done with a `MOV` instruction.

Many other instruction sets keep the PC in a special register and use a special return or jump instruction to return from functions.

Treating the PC as an ordinary register complicates the implementation of the processor.

Functions have inputs, called **arguments**, and an output, called the **return value**.

Functions should calculate the return value and cause no other **unintended side effects**.

When one function calls another, the calling function, the **caller**, and the called function, the **callee**, must agree on where to put the arguments and the return value.

In ARM, the caller conventionally places up to four arguments in registers R0–R3 before making the function call.

The callee places the return value in register R0 before finishing.

Function call with arguments and return values

```c
int main() {
  int y;
  y = diffofsums(2, 3, 4, 5); // -4
}

int diffofsums(int f, int g, int h, int i) {
  int result;
  result = (f + g) - (h + i);
  return result;
}
```

```
  MOV R0, #2        ; argument 0 = 2
  MOV R1, #3        ; argument 1 = 3
  MOV R2, #4        ; argument 2 = 4
  MOV R3, #5        ; argument 3 = 5
  BL DIFFOFSUMS     ; call function
  MOV R4, R0        ; y = returned value

stop B stop

DIFFOFSUMS
  ADD R8, R0, R1 ; R8 = f + g
  ADD R9, R2, R3 ; R9 = h + i
  SUB R4, R8, R9 ; result = (f + g) - (h + i)
  MOV R0, R4     ; put return value in R0
  MOV PC, LR     ; return to caller
```

According to ARM convention, the calling function, places the function arguments from left to right into the input registers, R0–R3.

The called function, `diffofsums` , stores the return value in the return register, R0.

When a function with more than four arguments is called, the additional input arguments are placed on the stack.

Stack – a portion of memory used for temporary variables

The stack **expands** (uses more memory) as the processor needs more scratch space.

The stack **contracts** (uses less memory) when the processor no longer needs the variables stored there.

The stack is a **last-in-first-out** (LIFO) queue.

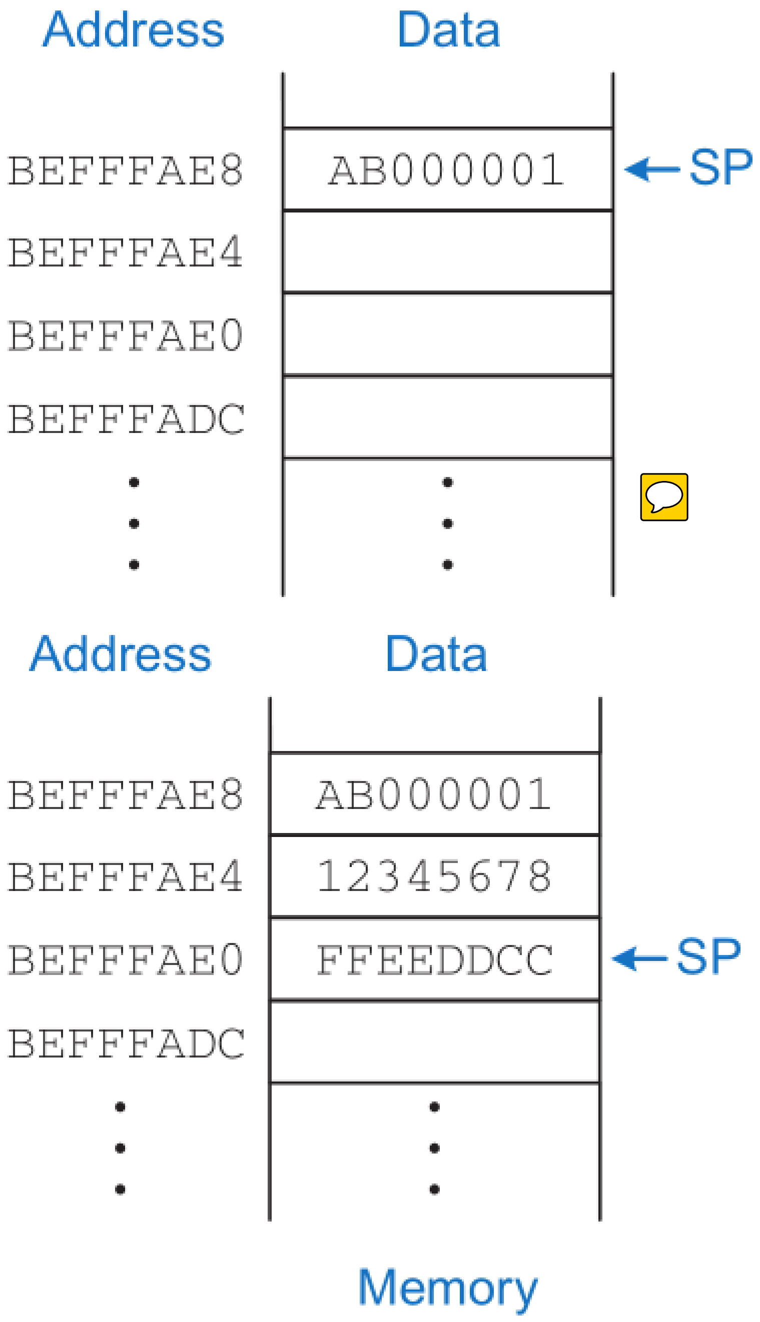
The last item **pushed** onto the stack is the first one that can be **popped** off.

Each function may allocate stack space to store local variables but must deallocate it before returning.

The **top of the stack** is the most recently allocated space.

The ARM stack grows down in memory.

The stack expands to lower memory addresses when a program needs more scratch space.



The **stack pointer**, SP (R13), is an ordinary ARM register that, by convention, **points** to the top of the stack.

A pointer is a memory address.

The SP starts at a high memory address and decrements to expand as needed.

The top of the stack is actually the lowest address.

This is called a **descending stack**.

ARM also allows for **ascending stacks** that grow up toward higher memory addresses.

One of the important uses of the stack is to save and restore registers that are used by a function.

A function should calculate a return value but have no other unintended side effects.

It should not modify any registers besides R0, the one containing the return value.

The `diffofsums` in the last example violates this rule because it modifies R4, R8, and R9.

To solve this problem, a function should save registers on the stack before it modifies them, then restores them from the stack before it returns.

A function should perform the following steps:

1. Make space on the stack to store the values of registers
2. Store the values of the registers on the stack
3. Execute the function using the registers
4. Restore the original values of the registers from the stack
5. Deallocate space on the stack

The stack space that a function allocates for itself is called its **stack frame**.

Each function should access only its own stack frame, not the frames belonging to other functions.

Function saving registers on the stack

```
DIFFOFSUMS
  SUB SP, SP, #12       ; make space on stack for 3 registers
  STR R9, [SP, #8]      ; save R9 on stack
  STR R8, [SP, #4]      ; save R8 on stack
  STR R4, [SP]          ; save R4 on stack
  ADD R8, R0, R1        ; R8 = f + g
  ADD R9, R2, R3        ; R9 = h + i
  SUB R4, R8, R9        ; result = (f + g) - (h + i)
  MOV R0, R4            ; put return value in R0
  LDR R4, [SP]          ; restore R4 from stack
  LDR R8, [SP, #4]      ; restore R8 from stack
  LDR R9, [SP, #8]      ; restore R9 from stack
  ADD SP, SP, #12       ; deallocate stack space
  MOV PC, LR            ; return to caller
```

When the function returns, R0 holds the result, but there are no other side effects.

R4, R8, R9, and SP have the same values as they did before the function call.

The stack pointer typically points to the topmost element on the stack.

This is called a **full stack**.

ARM also allows for **empty stacks** in which SP points one word beyond the top of the stack.

The ARM **Application Binary Interface** (**ABI**) defines a standard way in which functions pass variables and use the stack so that libraries developed by different compilers can interoperate.

LDM – load multiple registers

STM – store multiple registers

```
DIFFOFSUMS
  STMFD SP!, {R4, R8, R9} ; push R4/8/9
                          ; on full descending stack
  ADD R8, R0, R1          ; R8 = f + g
  ADD R9, R2, R3          ; R9 = h + i
  SUB R4, R8, R9          ; result = (f + g) - (h + i)
  MOV R0, R4              ; put return value in R0
  LDMFD SP!, {R4, R8, R9} ; pop R4/8/9
                          ; off full descending stack
  MOV PC, LR              ; return to caller
```

LDM and STM come in four flavors for full and empty descending and ascending stacks (FD , ED , FA , EA).

SP! indicates to store the data relative to the stack pointer and to update the stack pointer after the store or load.

`PUSH` is a synonym for `STMFD SP!`

`POP` is a synonym for `LDMFD SP!`

`PUSH` (and `POP`) save (and restore) registers on the stack in order of register number from low to high, with the lowest numbered register placed at the lowest memory address, regardless of the order listed in the assembly instruction.

```
PUSH {R8, R1, R3}
```

– stores R1 at the lowest memory address, then R3 and finally R8 at the next higher memory addresses on the stack.

ARM divides registers into **preserved** and **nonpreserved**

Preserved registers: R4 – R11, SP(R13), LR (R14)

Nonpreserved registers: R0 – R3, R12

A function must save and restore any of the preserved registers that it wishes to use.

A function can change the nonpreserved registers freely.

A further improved version of `diffofsums` that saves only R4 on the stack:

```
DIFFOFSUMS
  PUSH {R4}         ; save R4 on stack
  ADD R1, R0, R1    ; R1 = f + g
  ADD R3, R2, R3    ; R3 = h + i
  SUB R4, R1, R3    ; result = (f + g) - (h + i)
  MOV R0, R4        ; put return value in R0
  POP {R4}          ; pop R4 off stack
  MOV PC, LR        ; return to caller
```

The code reuses the nonpreserved argument registers R1 and R3 to hold the intermediate sums when those arguments are no longer necessary.

When one function calls another, the callee must save and restore any preserved registers that it wishes to use.

The callee may change any of the nonpreserved registers.

Hence, if the caller is holding active data in a nonpreserved register, the caller needs to save that nonpreserved register before making the function call and then needs to restore it afterward.

For these reasons, preserved registers are also called **callee-save**,

nonpreserved registers are called **caller-save**.

R4–R11 are generally used to hold local variables within a function, so they must be saved.

LR must also be saved, so that the function knows where to return.

R0–R3 and R12 are used to hold temporary results.

These calculations typically complete before a function call is made, so they are not preserved.

R0–R3 are often overwritten in the process of calling a function.

They must be saved by the caller if the caller depends on any of its own arguments after a called function returns.

R0 should not be preserved, because the callee returns its result in this register.

The Current Program Status Register (CPSR) holds the condition flags, so it is not preserved across function calls.

The convention of which registers are preserved or not preserved is NOT a part of the the ARM architecture.

It is a part of the **Procedure Call Standard** for the ARM Architecture.

Alternate procedure call standards exist.

Optimized diffofsums function call:

```
DIFFOFSUMS
  ADD R1, R0, R1    ; R1 = f + g
  ADD R3, R2, R3    ; R3 = h + i
  SUB R0, R1, R3    ; return (f + g) - (h + i)
  MOV PC, LR        ; return to caller
```

We stored the result in the return register R0, eliminating the need to push and pop R4 and to move result from R4 to R0.

Additional arguments

Functions may have more than four input arguments and may have too many local variables to keep in preserved registers.

The stack is used to store this information.

By ARM convention, if a function has more than four arguments, the first four are passed in the argument registers as usual.

Additional arguments are passed on the stack,
just above SP.

The caller must expand its stack to make room for the additional arguments.

Local variables

A function can also declare local variables or arrays.

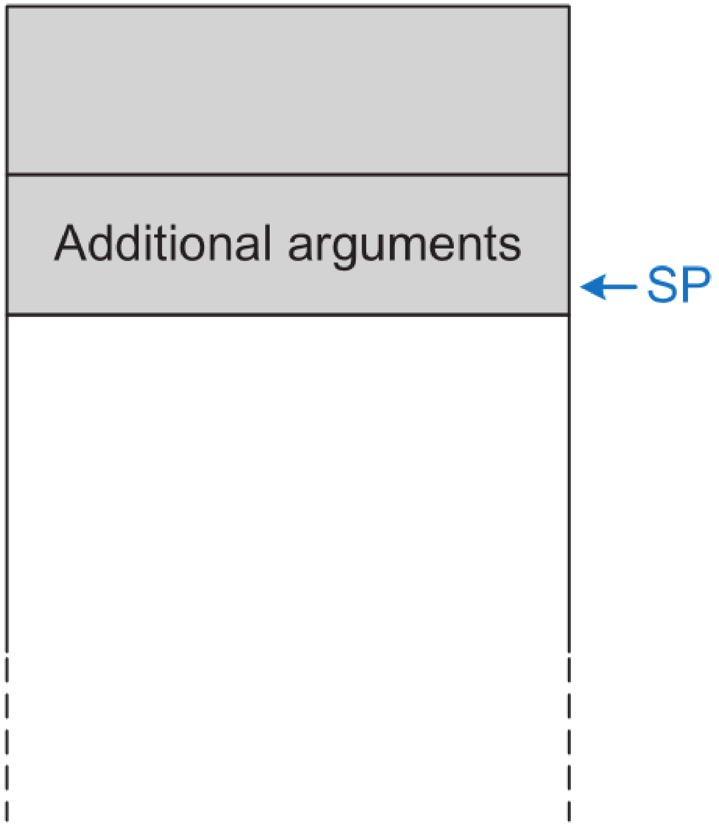
Local variables are declared within a function and can be accessed only within that function.

Local variables are stored in R4–R11.

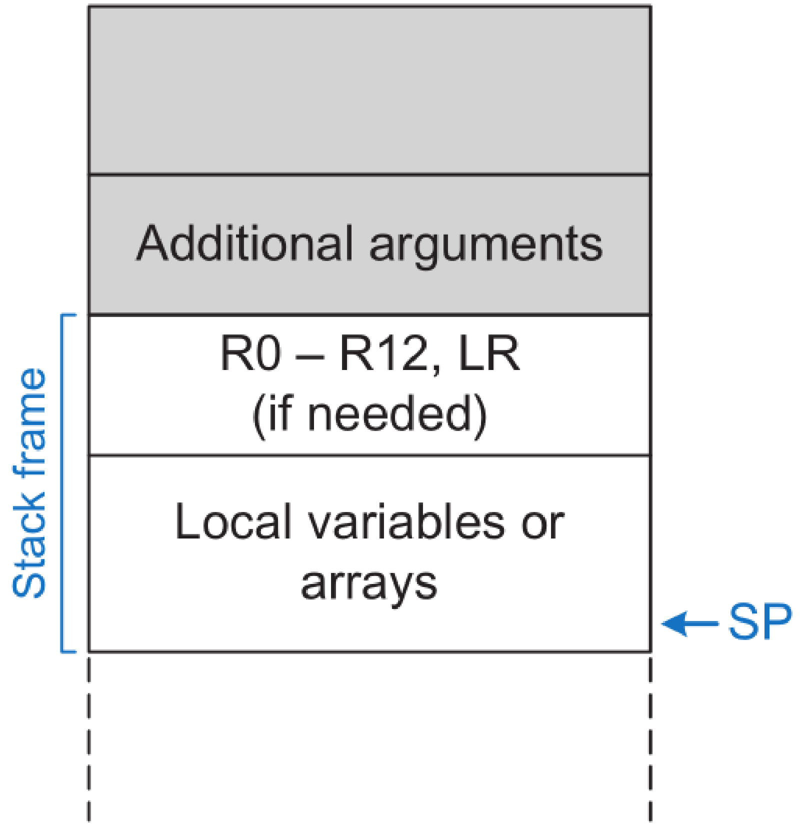
If there are too many local variables, they can also be stored in the function’s stack frame.

In particular, local arrays are stored on the stack.

The caller's stack for calling a function with more than 4 arguments



The callee's stack frame



If the callee has more than four arguments, it finds them in the caller's stack frame.

Exercise 8.1 Play with the stack

Push 5 values to the stack:

Find these values in memory

Observe the stack pointer (R13) and the disassembly window

Note how the stack expands toward lower memory addresses.

```
MOV R0, #1
PUSH {R0}
MOV R0, #2
PUSH {R0}
MOV R0, #3
PUSH {R0}
MOV R0, #4
PUSH {R0}
MOV R0, #5
PUSH {R0}
```

Then pop the values from the stack:

In which order they are popped?

The values are still in the stack

```
POP {R1}
POP {R2}
POP {R3}
POP {R4}
POP {R5}
```

Can you now get the value 3 and pop it into R6?

Do it as follows: put the base address 0xFFFFFFFC to the SP, and then use LDR and the offset

Exercise 8.2

```
int main() {
  int y;
  y = diffofsums(2, 3, 4, 5); // -4
}

int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;
}
```

Write an assembly version of this code without passing the arguments through the registers R0, R1, R2, R3.

The main function should push the arguments to the stack.

```
PUSH {R0, R1, R2, R3}
```

Notice how the values are stored in the stack.

The diffofsums should pop the arguments from the stack.

```
POP {R0, R1, R2, R3}
```

Exercise 8.3

These days, ARM compilers do a function return using

```
BX LR
```

The `BX` branch and exchange instruction is like a branch, but it also can transition between the standard ARM instruction set and the Thumb instruction set

Replace

```
MOV PC, LR
```

with

```
BX LR
```

The warning should disappear

Exercise 8.4

Translate the following high-level function into ARM assembly.

```
int f(int n, int k) {
  int b;
  b = k + 2;
  if (n == 0) b = 10;
  else b = b + (n * n);
  return b * k;
}
```

Pay particular attention to properly saving and restoring registers across function calls and using the ARM preserved register conventions.

Keep local variable b in R4

Exercise 8.5

Step through your function from the previous exercise for the cases of f(2, 3) and f(0,3).

Put some number into R4 in the main program (e.g. 0xFFFFFFAB), and see how it is pushed to the stack, and then popped from the stack.

Keep track of the values of the stack pointer (SP), link register (LR), and program counter (PC).

You might also find it useful to keep track of the values in R0, R1, and R4 throughout execution.