Computer organization and architecture

Lecture 9

Nonleaf function calls

A function that does not call other functions is called a **leaf function**

A function that does call other functions is called a **nonleaf function**

Caller save rule:

Before a function call, the caller must save any nonpreserved registers (R0–R3 and R12) that it needs after the call.

After the call, it must restore these registers before using them.

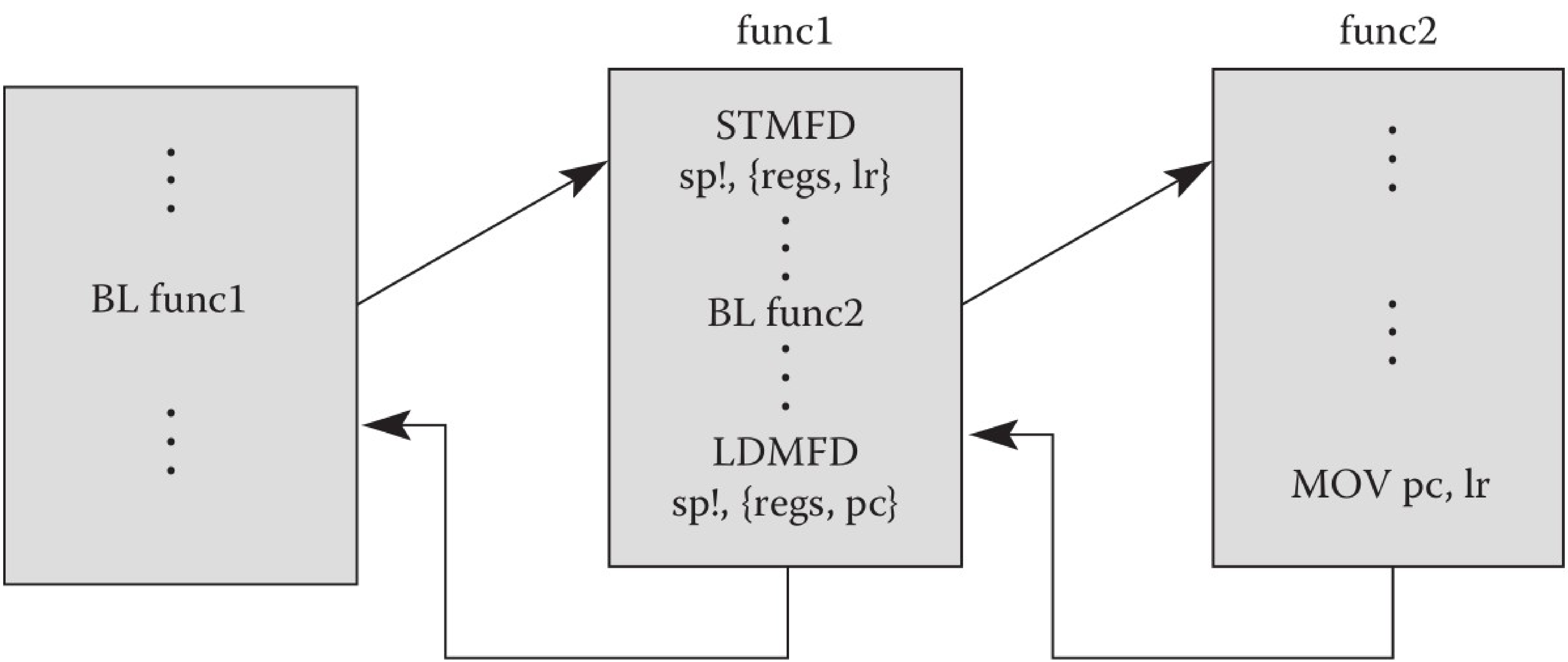
Callee save rule:

Before a callee disturbs any of the preserved registers (R4–R11 and LR), it must save the registers.

Before it returns, it must restore these registers.

A nonleaf function overwrites `LR` when it calls another function using `BL`

Thus, a nonleaf function must always save `LR` on its stack and restore it before returning.



```c
int f1(int a, int b)
{
  int i, x;
  x = (a+b)*(a-b);
  for(i=0; i<a; i++)
    x = x+f2(b+i);
  return x;
}

int f2(int p) {
  int r;
  r = p + 5;
  return r + p;
}
```

```
F1
  PUSH {R4, R5, LR} ; save preserved regs used by f1
  ADD R5, R0, R1    ; x = (a+b)
  SUB R12, R0, R1   ; temp = (a-b)
  MUL R5, R5, R12   ; x = x * temp = (a+b) * (a-b)
  MOV R4, #0        ; i = 0
FOR
  CMP R4, R0        ; i < a?
  BGE RETURN        ; no: exit loop
  PUSH {R0, R1}     ; save nonpreserved regs
  ADD R0, R1, R4    ; argument is b + i
  BL F2             ; call f2(b+i)
  ADD R5, R5, R0    ; x = x + f2(b+i)
  POP {R0, R1}      ; restore nonpreserved regs
  ADD R4, R4, #1    ; i++
  B FOR             ; continue for loop
RETURN
  MOV R0, R5        ; return value is x
  POP {R4, R5, LR}  ; restore preserved registers
  MOV PC, LR        ; return from f1
F2
  PUSH {R4}         ; save preserved regs used by f2
  ADD R4, R0, 5     ; r = p+5
  ADD R0, R4, R0    ; return value is r+p
  POP {R4}          ; restore preserved regs
  MOV PC, LR        ; return from f2
```

Recursive function calls

A **recursive function** is a nonleaf function that calls itself

Recursive functions behave as both caller and callee and must save both preserved and nonpreserved registers.

Example: the factorial function

$$n! = 1 \times 2 \times \ldots \times (n-1) \times n$$

can be written as a recursive function

$$n! = n \times (n-1)!$$

```c
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n - 1));
}
```

```
FACTORIAL PUSH {R0, LR}    ; push n and LR on stack
          CMP R0, #1       ; R0 <= 1?
          BGT ELSE         ; no: branch to else
          MOV R0, #1       ; otherwise, return 1
          ADD SP, SP, #8   ; restore SP
          MOV PC, LR       ; return
ELSE      SUB R0, R0, #1   ; n = n - 1
          BL FACTORIAL     ; recursive call
          POP {R1, LR}     ; pop n (into R1) and LR
          MUL R0, R1, R0   ; R0 = n * factorial(n - 1)
          MOV PC, LR       ; return
```

The formula $n! = n \times (n-1)!$ is not tail recursive

The stack size grows with n

If a function is recursive but not tail recursive, it is possible to run out of stack for extremely deep recursions.

A function is **tail recursive** if it calls itself as its last action.

In this case the function's stack frame can be reused.

In the formula $n! = n \times (n-1)!$

the factorial is not the last action

The multiplication function is in the tail position.

A tail recursive version of factorial

```
facttail(n) = fact-iter(1, n);

fact-iter(p, n) =
    if (n < 2) return p;
    else
    return fact-iter(p*n, n-1);
```

The `fact-iter` calls itself last in the control flow.

The stack does not grow with n.

```
FACTTAIL PUSH {LR}
         MOV R1, #1
         BL FACTITER
         POP {LR}
         MOV PC, LR

FACTITER PUSH {LR}
         ADD SP, SP, #4
         CMP R0, #2
         BGE ELSE
         MOV R0, R1
         MOV PC, LR
ELSE     MUL R1, R0, R1
         SUB R0, R0, #1
         BL FACTITER
         POP {LR}
         MOV PC, LR
```

If a function calls itself as its last action, the function’s stack frame can be reused.

In general, if the last action of a function consists of calling a function (which may be the same), one stack frame would be sufficient for both functions.

Such calls are called **tail-calls**.

https://en.wikipedia.org/wiki/Tail_call

Exercise 9.1

Implement the following code snippet in ARM assembly

```
void setArray(int num) {
  int i;
  int array[10];
  for (i = 0; i < 10; i = i + 1)
    array[i] = compare(num, i);
  }

int compare(int a, int b) {
  if (a >= b)
    return 1;
  else
    return 0;
}
```

Use R4 to hold the variable i

The array should be stored on the stack of the setArray function

Exercise 9.2

Calculate 5! using the recursive function from this lesson.

5! = 0x78

Observe the size and the contents of the stack.

Understand every line in the code.

Calculate 10! using the same recursive function.

10! = 0x375F00

Observe the size and the contents of the stack in this case, and compare it with the previous case.

Try to run both cases using the tail recursive version of factorial.

Observe the size and the contents of the stack.

Exercise 9.3

Write an assembly code for the recursive version of a function that calculates the sum of natural numbers.

```
int sum(int num)
{
    if (num!=0)
        return num + sum(num-1);
    else
        return num;
}
```

Test it for $n = 0$, $n = 1$, and $n = 10$.

Observe the size and the contents of the stack.

Can you write a tail recursive version of the sum?