Lancaster University

Topic 6: Functions

Overview

Lancaster University

- Functions can greatly simplify programming tasks
  - Abstraction & Modularity, Code reuse, Readability, Testability & validation, Maintainability
- Topic of the day: How do we implement functions in Assembly?
  - Function and register state
  - Stack memory
  - Register lr

Function

Lancaster University

A function is a stored subroutine that performs a specific task based on the parameters with which it is provided

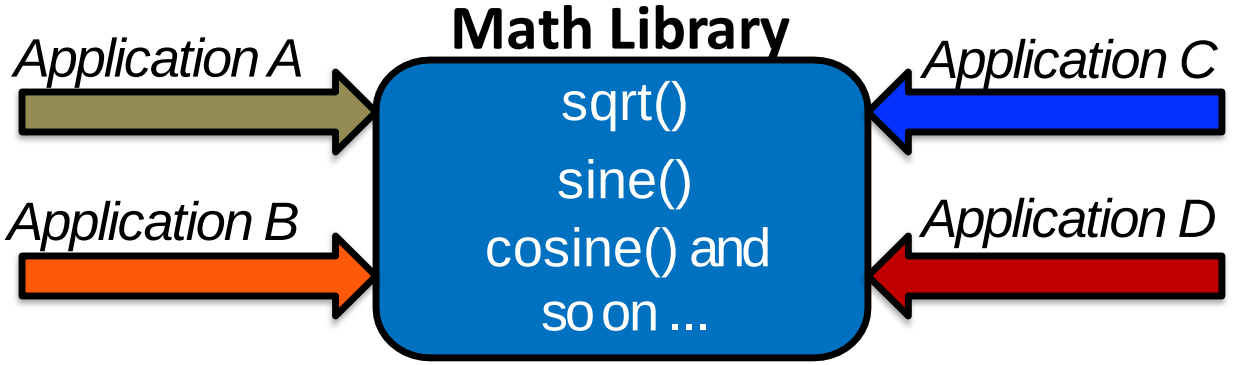
- Complex operations can be performed by calling a procedure.
- Make code easier to understand and manage.
- Program elements that can easily be re-used.
- Same procedure can be called many times within a program.

NOTE: branching to a label is not a function. You should follow specific rules in order to implement a function.

API

Lancaster University

- Application Programming Interface (API)
  - Defines the interfaces by which one software program communicates with another at the source code level



- API defines the interface only
  - The user of the API can ignore the implementation
  - Many implementations of the same API
  - C standard library hides many low-level details of the system

Functions as Detectives

Lancaster University

- Assigned a secret mission (function call)
- Acquires necessary resources (acquire parameters and memory - stack)
- Perform the mission (execute instructions)
- Leaves no trace (clean up memory)
- Returns safely to the point of origin (function return)

Breaking Down Function Execution

Lancaster University

1. Caller stores arguments in registers or memory
2. Function call: Caller transfers flow control to the callee
3. Callee acquires/allocates memory for doing work
4. Callee executes the function body
5. Callee stores the result in “some” register
6. Callee deallocates memory
7. Function return: Callee returns control the caller

Instructions for procedure calls

```
bl ProcedureAddress
```

"branch and link"

label to jump to

- `bl` stores the address of the next instruction in register `lr`
- ...and then jumps to `ProcedureAddress`
- to get back, we restore the current address to `pc`

```
mov pc, lr
```

Copy lr address to pc

```
bx lr
```

Branch to lr

Calling Convention

Lancaster University

- Assembly offers limited resources for computation, i.e. registers.
  - A function implementation should follow a set of calling conventions to ensure interoperability.
  - Many times, these conventions makes your code inefficient.
  - *Leave no trace.*
- With a convention in place:
  - Functions written by different programmers can interoperate
  - Functions compiled by two different compilers can interoperate
  - A library function by a third party can be used without corrupting state

Convention 1: registers for procedure calls

Lancaster University

`r0-r3:`

“argument” registers in which to pass parameters

`r0:`

return value registers

`lr:`

return address register (link register)

Convention 2: Preserving registers

Lancaster University

Preserved	Nonpreserved
Saved registers: r4 – r11	Temporary register: r12
Stack pointer: SP (r13)	Argument registers: r0 – r3
Return address: LR (r14)	Current Program Status Register
Stack above the stack pointer	Stack below the stack pointer

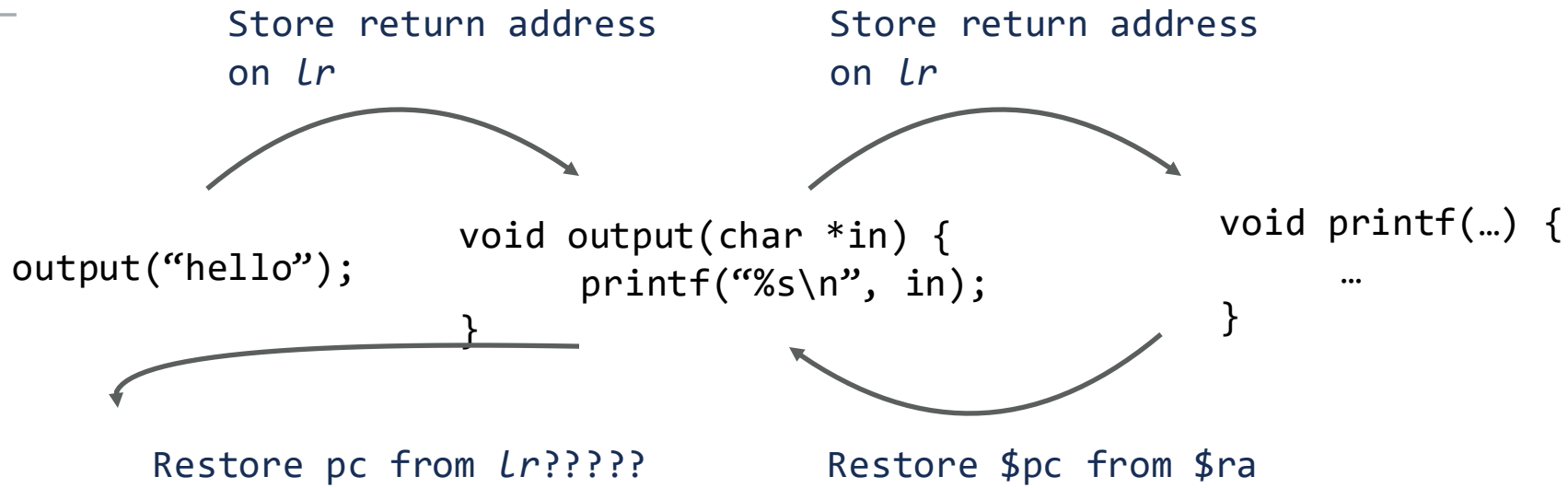
- Assembly convention
  - registers must be restored after procedure call.
  - If usage of these registers is avoided no spilling of registers on the stack is required.

Convention 3: Preserving registers

Lancaster University

- Sometimes a procedure needs to use more registers than just four arguments and two return values.
- Register content must be preserved during a procedure call
- Moving the contents of registers to the main memory is called **spilling registers**.
- Registers are stored to memory using a conceptual data structure known as a **stack**.
- The stack pointer register `sp` points to the contents of the register most recently pushed onto the stack.

Procedure nesting



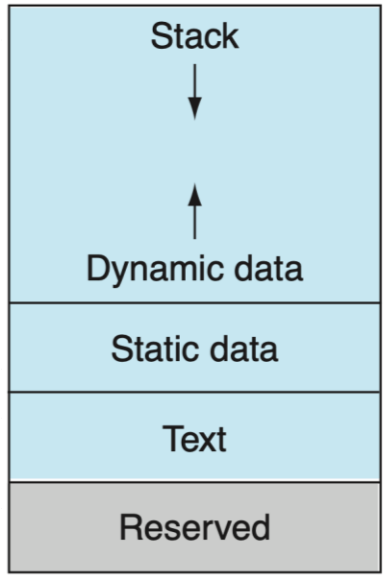
- `bl` save `pc` to `lr`.
- We must always save the lr register, if a nested procedure is called.
- `Lr` can be saved in stack.
  - If usage of these registers is avoided, no spilling of registers on the stack is required.

Stack memory

Lancaster University

- Stack is a memory region used to store local state for your functions.
- It is a dynamic memory region.
  - The stack pointer (sp) starts at address 0x20020000.
  - The current stack pointer bottom is pointed by register sp.
  - A stack is like a Last In First Out (LIFO) Queue.
  - You can use register sp to grow (decrease register sp) and shrink (increase register sp).

- The stack is used to store:
  - Preserved registers.
  - Local function variables
  - input arguments to the function.

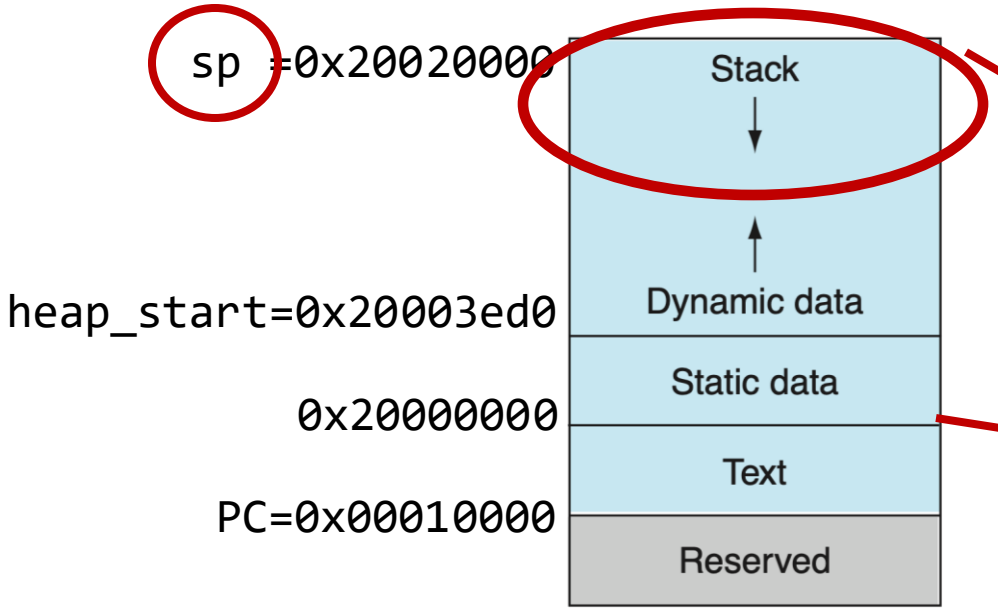


Stack Memory

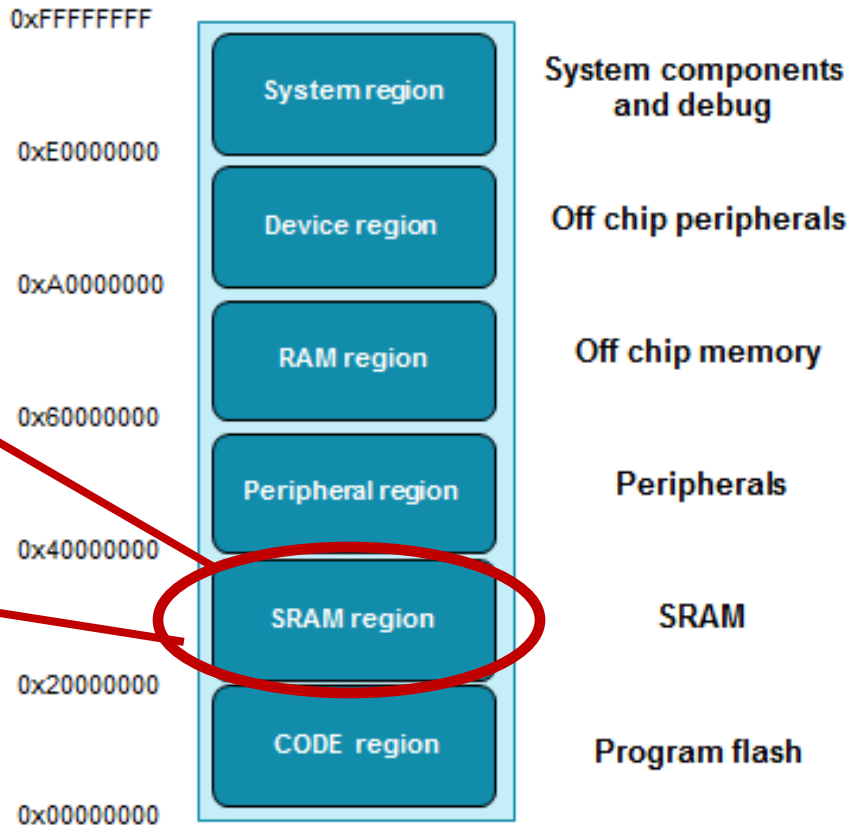
Lancaster University

the stack pointer value gets smaller.

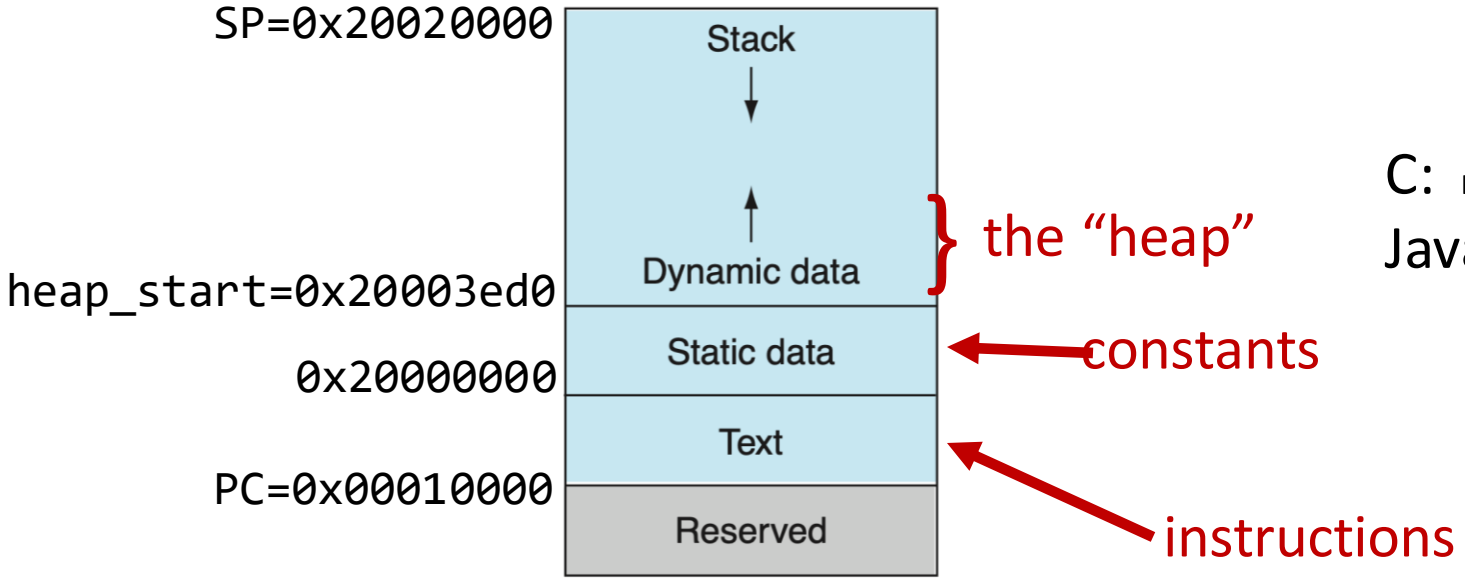
As words are pushed onto the stack, the stack "grows" down.



P&H fig. 2.17



Memory Layout



C: malloc()

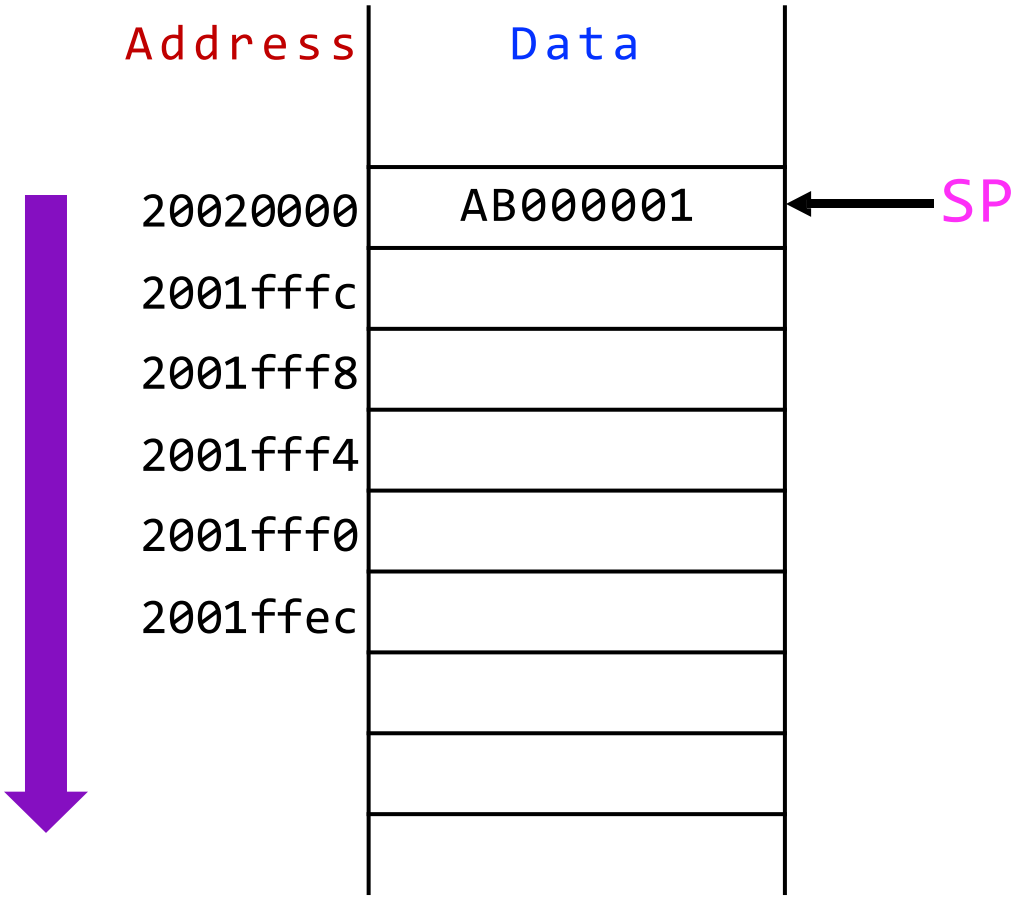
Java: new

P&H fig. 2.17

The Stack

Lancaster University

- ARM stack grows down in memory
- Stack Pointer (SP) points to the **top of the stack**
- SP register holds the address of (points to) the **top of the stack**



contents of stack pointer

SP `0x20020000`

Growing the Stack

Lancaster University

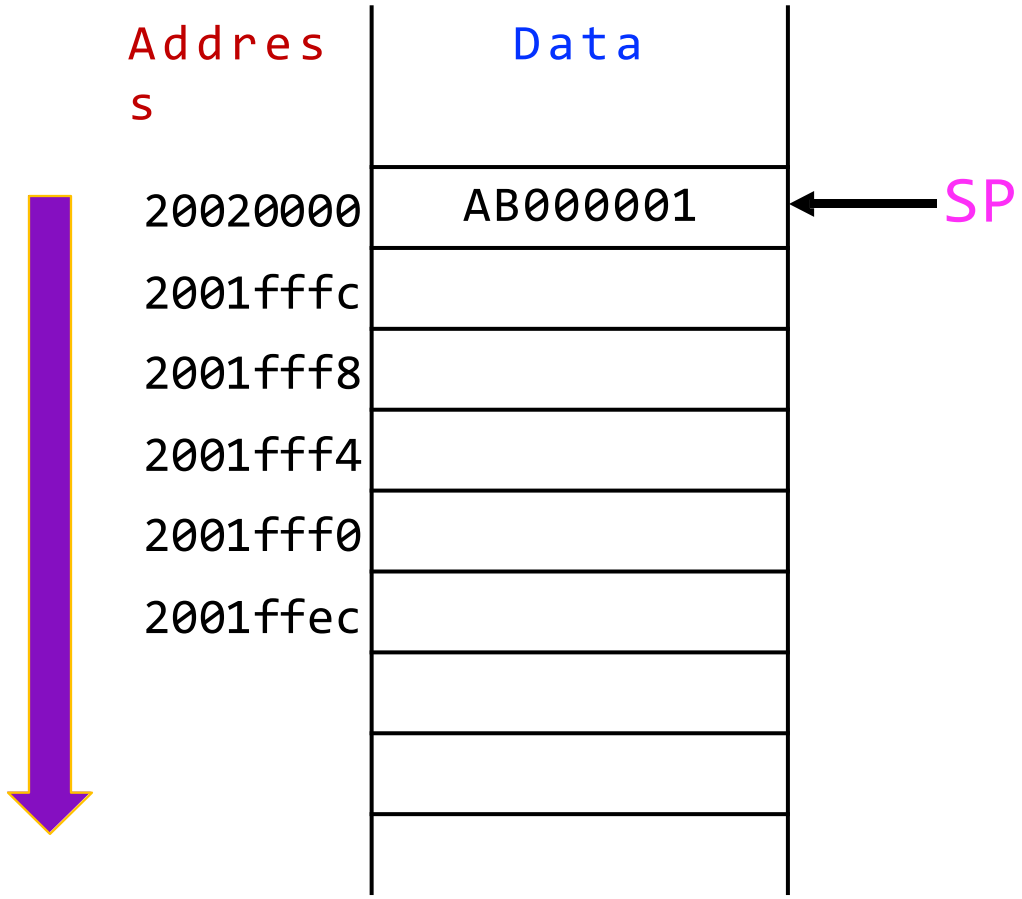
- Let us push two items on the stack

  0x12345678

  0xFFFFDDCC

Where does the SP points to now?

- How does the stack look?



contents of stack pointer

SP `0x20020000`

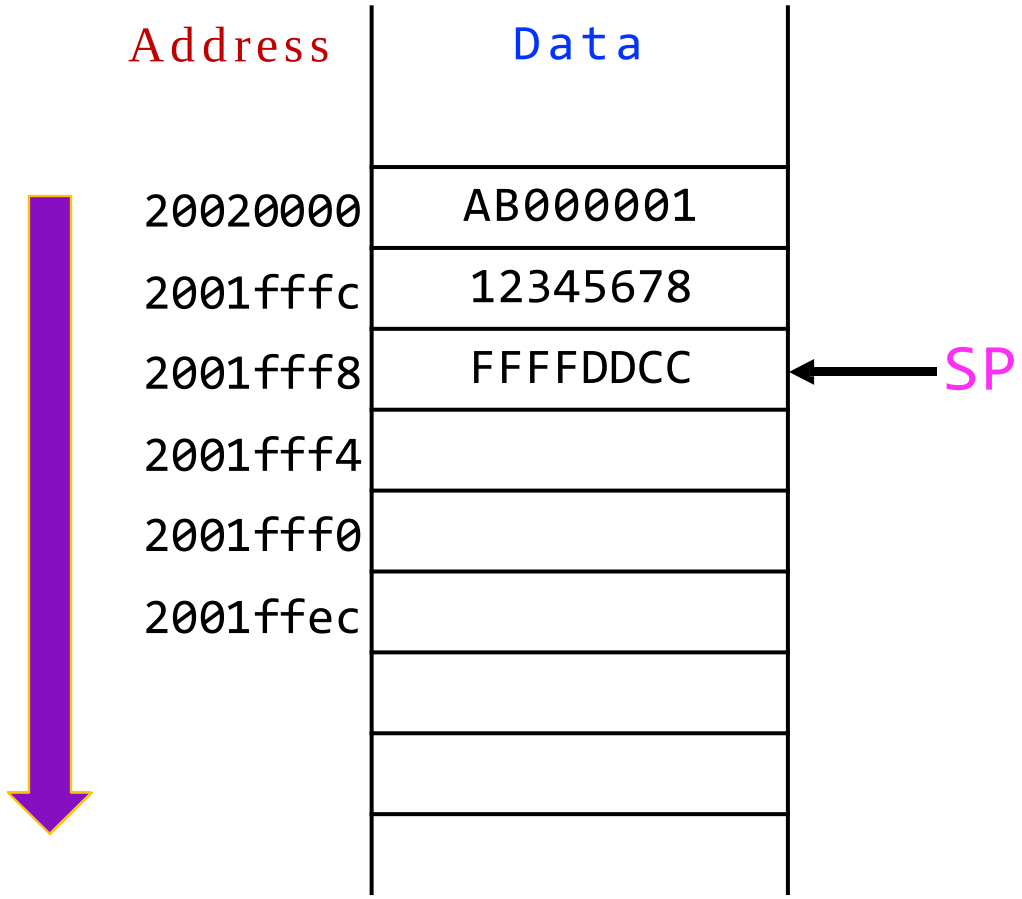
Growing the Stack

Lancaster University

- SP points to the most recently pushed item on the stack
- SP decrements by 8 to make space for two words

contents of stack pointer

SP 0x2001ff8



Instructions for stack

Lancaster University

```
push {r4, r5, lr}
```

"push registers onto stack."

List of registers

- Push modifies sp register to make space

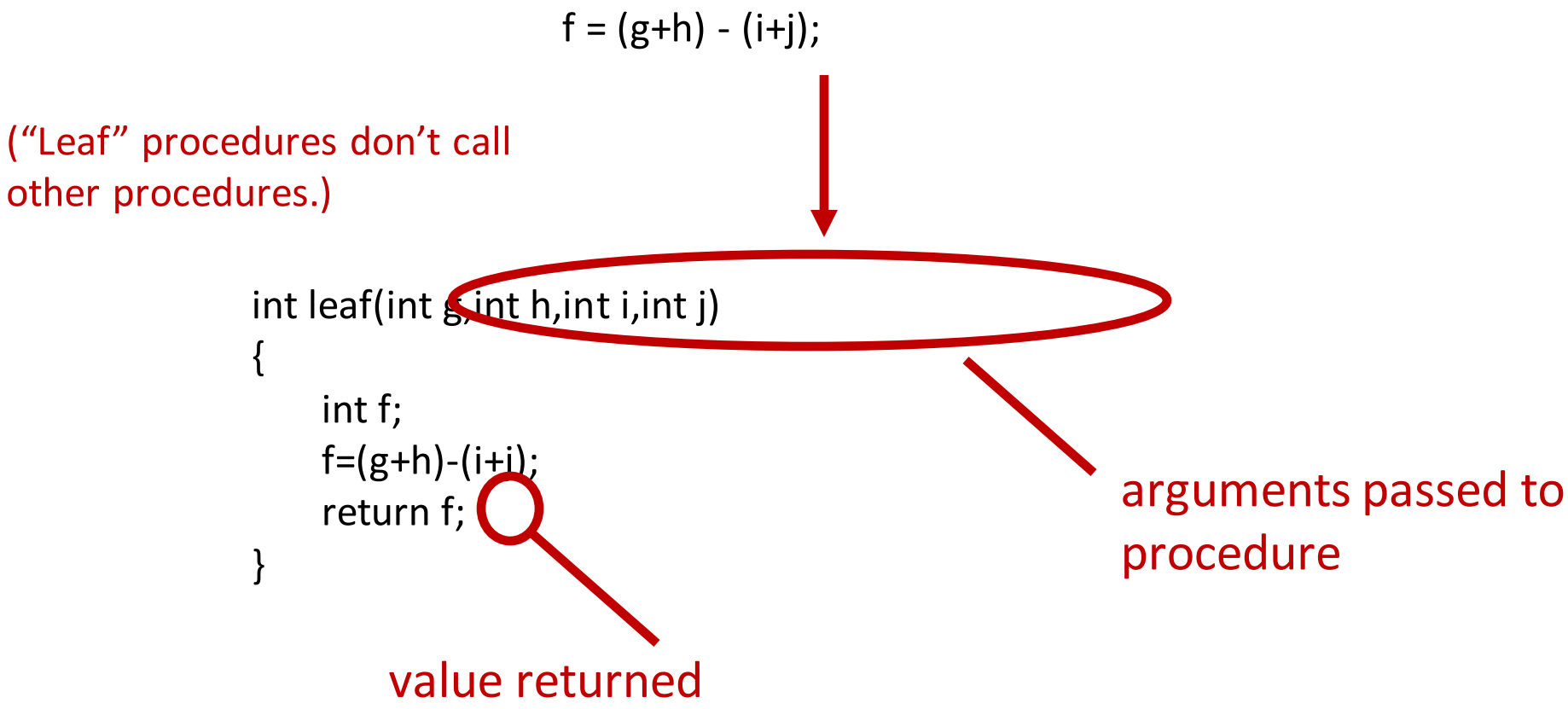
  …and saves registers in memory.

  to restore sp and registers, use pop

```
pop {r4, r5, lr}
```

"pop registers onto stack"

A simple procedure in C



Complex Calculations

Lancaster University

```
add r0,r1

add r1,r2,r3

sub r0,r1
```

equivalent C statement

```
f = (g + h) - (i + j);
```

register mapping

f: r0
g: r0
h: r1
i: r2
j: r3

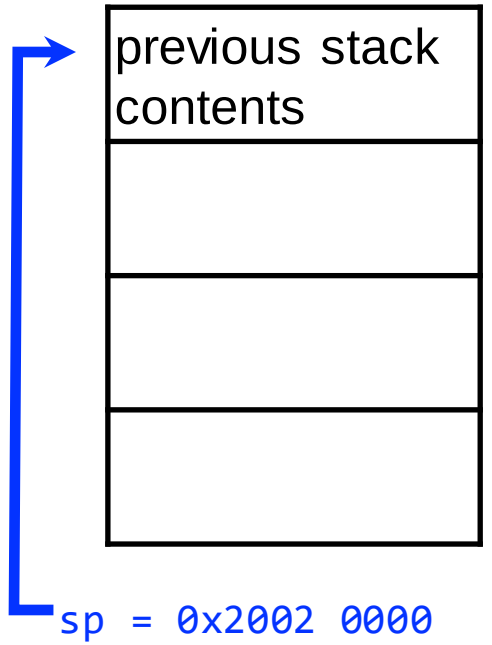
Complex calculations must be broken down in smaller steps —> compiler !

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

leaf:

procedure label
(used by caller's
"branch and link")

main memory



register mapping

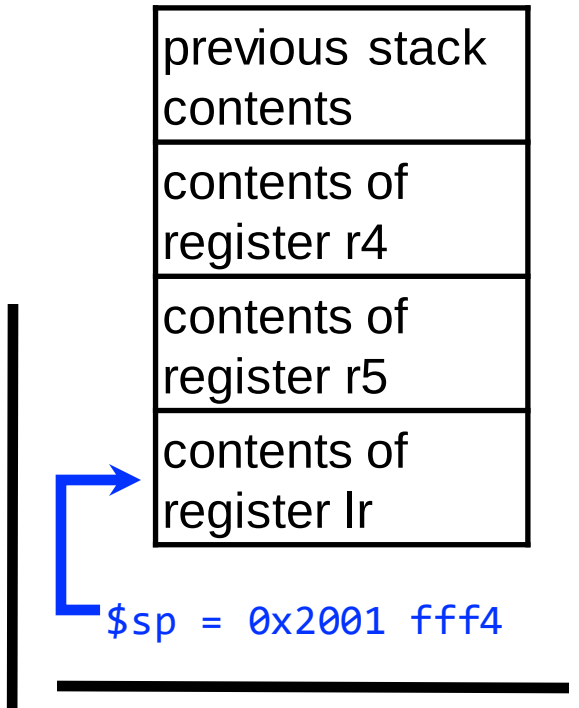
```
f: r0
g: r0
h: r1
i: r2
j: r3
temp: r4, r5
```

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

```
leaf:

    push {r4, r5, lr}
```

main memory



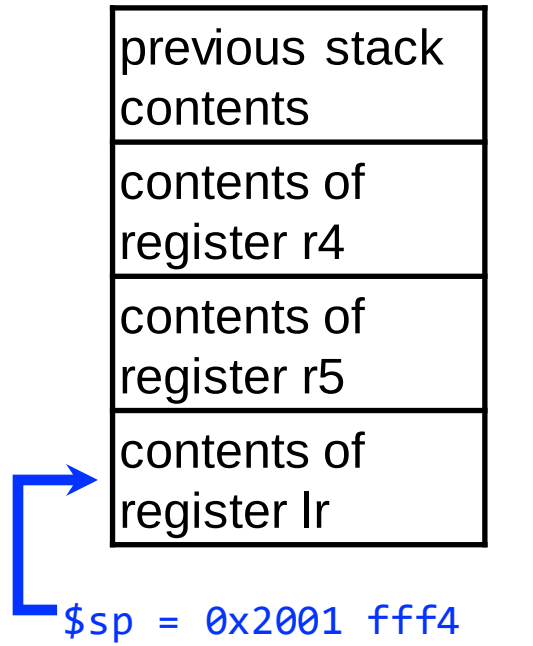
register mapping

```
f: r0
g: r0
h: r1
i: r2
j: r3
temp: r4, r5
```

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

```
leaf:
    push {r4, r5, lr}
    add r4,r0,r1     @ r4 contains g+h
    add r5,r2,r3     @ r5 contains i+j
    sub r0,r4,r5     @ f = r4 - r5
    pop {r4, r5, lr}@restore r4, r5, lr
    mov pc, lr       @ return back
```

main memory



register mapping

```
f: r0
g: r0
h: r1
i: r2
j: r3
temp: r4, r5
```

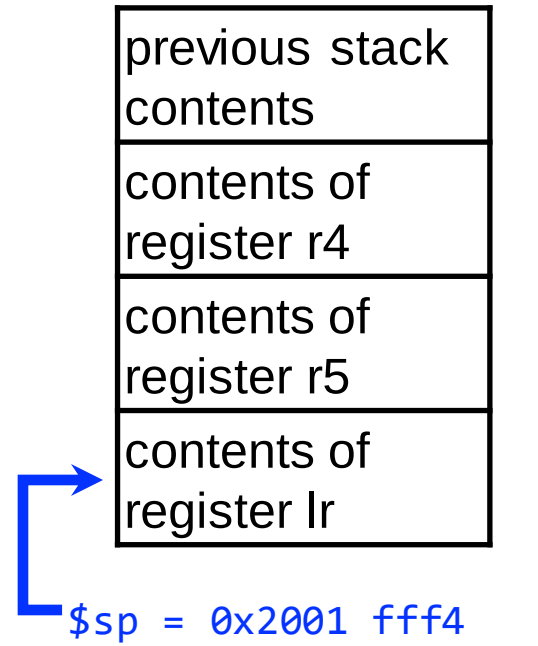
```
int leaf_example(int g, int h, int i, int j)
{
   int f;
   f=(g+h)-(i+j);
   return f;
}
```

```
leaf:

   add r0,r1   @ r0 contains g+h
   add r2,r3   @ r2 contains i+j
   sub r0,r2   @ f = r0 – r2
   mov pc, lr  @ return back
```

use non preserved registers for local variables

main memory



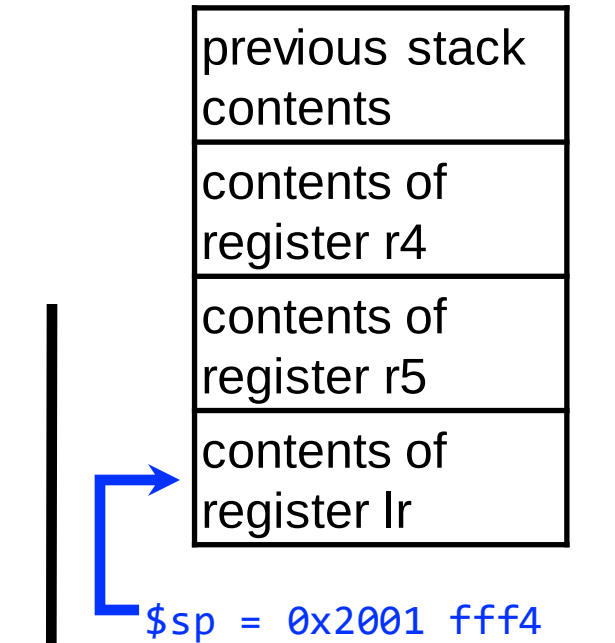
register mapping

```
f: r0
g: r0
h: r1
i: r2
j: r3
temp: r0, r2
```

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

```
leaf:
   add r0,r1   @ r0 contains g+h
   add r2,r3   @ r2 contains i+j
   sub r0,r2   @ f = r0 - r2
   mov pc, lr    @ return back
main:
    mov r0, #5
    mov r1, #6
    mov r2, #4
    mov r3, #3
    bl leaf
```

main memory



register mapping

```
f: r0
g: r0
h: r1
i: r2
j: r3
temp: r0, r2
```

Summary

Lancaster University

Procedures/Function calls

An example

Next

- Inline Assembly in C