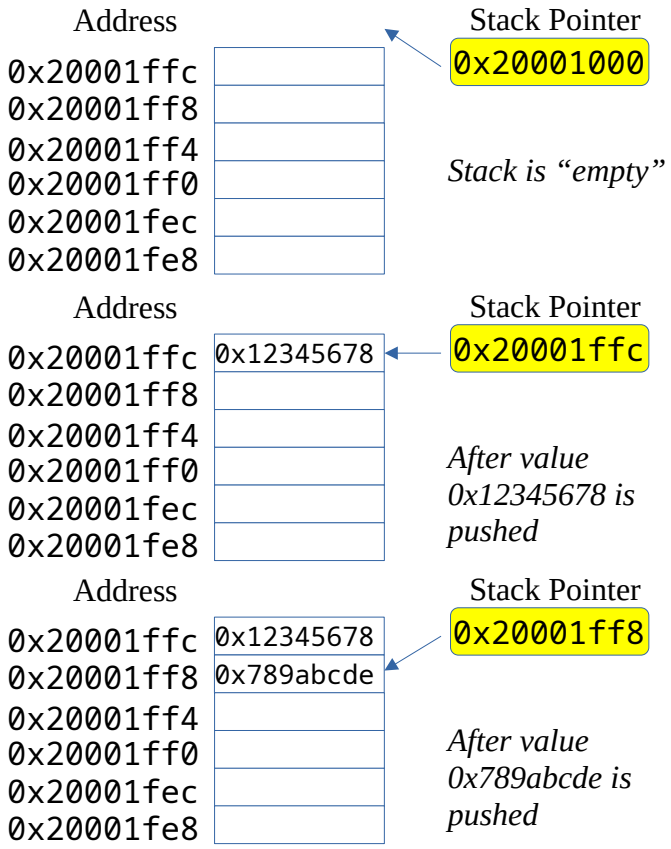
Assembler lab 3: Using the stack

Introduction.
The stack is used to store backup copies of registers during function calls and interrupt handlers, to pass some function parameters and to store local variables. In the context of this module, the stack is a block of RAM which is pointed to by the Stack Pointer register (SP).



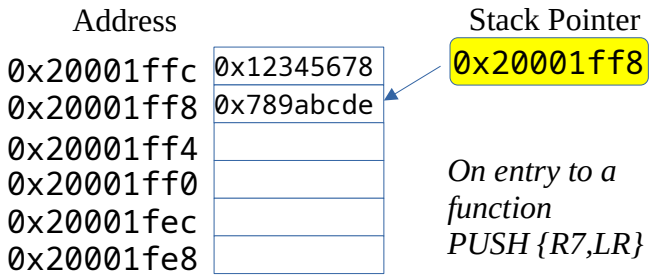
As you can see above, as values are pushed on to the stack it grows downwards from higher memory addresses to lower addresses. Each push moves the stack pointer by 4 bytes (this is a 32 bit CPU after all). Values can not be directly pushed on the stack. Only registers are pushed and popped. The "Push" algorithm is as follows:
1) Subtract 4 from SP
2) Store the register you wish to push where SP points.
Some examples of PUSHing registers to the stack:

```
PUSH {R7} // save a single register to the stack
PUSH {R4-R7,LR} // push multiple registers to the stack
```

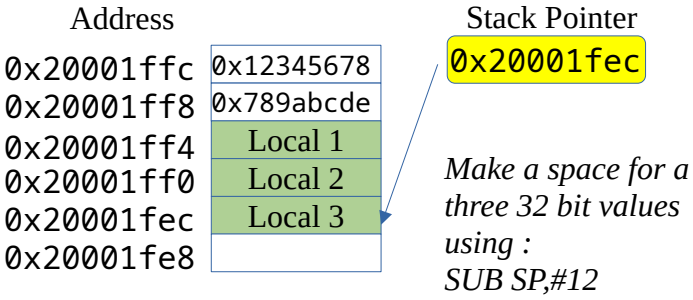
Values are retrieved from the stack using a POP instruction. This works are follows:
1) Copy the value from where SP points into the target register
2) Add 4 to SP.
Some examples of POPing :

```
POP {R7} // pop a single register
POP {R4-R7} // pop multiple registers
```

Local variables are stored on the stack as follows:
1) On entry into a function, registers R7 and LR are typically saved on the stack



2) A "hole" is made in the stack by subtracting a value (which is a multiple of 4 bytes) from SP. If there are future PUSHes then these will use memory below this hole. R7 is usually made point at this stack hole (R7 is often called a stack frame pointer or just a frame pointer). If the programmer does further "PUSH's" which will alter SP, R7 will remain unchanged and so will still point at the local variables.



3) Access the local variable using the frame pointer as follows:

```
LDR R0,[R7] // read local variable 3 into R0
STR R0,[R7] // write local variable 3 with the contents of R0
```

If there is more then one local variable then R7 is combined with an offset to access each of them.

```
LDR R0,[R7,#4]      // read local variable 2 into R0
STR R1,[R7,#8]      // write local variable 1 with the contents of R1
```

4) Just prior to exiting the function, the space used by local variables must be reclaimed. This is done by adding the same value to SP as was taken away in the beginning. For example, in the case above we would do this:

```
ADD SP,#12   // reclaim stack space
POP {R7,PC} // restore R7 and return to the caller
```

Lab work.
Shown below is a C function and its assembly language equivalent. This function takes 2 pointers to integers as parameters. It's job is to swap the memory contents that each of these pointers point at. A local variable called ***temp*** is used in the swapping process.

```
void swap(uint32_t *x, uint32_t *y)
{
        uint32_t temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
```

The assembler version creates a hole in the stack big enough to hold temp. This space is accessed using R7 as a pointer. The comments show the equivalent line of C code.

```
// On entry: R0 points at x, R1 points at y
swap:
    push {R7,LR} // back up R7 and SP as they will change
    sub SP,#4    // make space for the local variable (uint32_t temp)
    mov R7,SP    // will use R7 as a stack frame pointer in case SP moves around
    ldr R2,[R0]  // read what x points at
    str R2,[R7]  // write it to temp (temp=*x)
    ldr R2,[R1]  // read what y points at
    str R2,[R0]  // write it to where x points (*x=*y)
    LDR R2,[R7]  // read temp
    str R2,[R1]  // write it to where y points (*y=temp)
    add SP,#4    // free space used by local variables
    pop {R7,PC}  // retore R7 and return to caller
```

To call on this version of swap you could do the following:

Assembler version

```
main:
// Insert your code here.

    ldr R2,=0x12345678  // data value 1
    ldr R3,=0x9abcdef0  // data value 2
    ldr R0,=0x20000000  // pointer 1 (x)
    ldr R1,=0x20000010  // pointer 2 (y)
    str R2,[R0] // write data value 1 to where x points
    str R3,[R1] // write data value 2 to where y points
    bl swap     // call swap function
    nop
    nop

exit:
    B .
```

C version

```
uint32_t v1,v2;
uint32_t *x,*y;
int main()
{
        x = &v1;
        y = &v2;
        v1 = 0x12345678;
        v2 = 0x9abcdef0;
        swap(x,y);
}
```

Task 1: Copy/paste or type the above code into the assembly language template and check to make sure it operates correctly

Task 2: A C language version of strlen is shown below along with a partially assembly language version. ***Complete the assembly language version and test it.*** You can pass it the address of **source_string** to test your code.

```c
uint32_t a_strlen(char *s)
{
        uint32_t len;
        char c;
        len = 0;
        while(c = *s)
        {
                s++;
                len++;
        }
        return len;
}
```

```
a_strlen:
    push {R7,LR} // back up R7 and SP as they will change
    sub SP,#8    // make space for the local variables (must be multiples of 4 bytes)
    mov R7,SP    // will use R7 as a stack frame pointer in case SP moves around
                 // 'len' is created first so is further up the stack: at R7+4
                 // 'c' is created last so it is where R7 points
    movs R1,#0   //
    str R1,[R7,#4] // len = 0;
a_strlen_loop:
    ldrb R2,[R0] // read a byte from where s points
    strb R2,[R7]  // write it to c (c = *s)
    cmp R2,#0
    beq a_strlen_exit
    adds R0,R0,#1 // increment the pointer (s++)

                // Insert assembler code for len++

    b a_strlen_loop
a_strlen_exit:
    add SP,#8    // free space used by local variables
    pop {R7,PC}  // retore R7 and return to caller

    .align 4

// put your constant data here

source_string:  .asciz "Hola Mundo"
```

Task 3: If time permits, implement and test an assembler version of **strcpy**