# Week 13-18 practical solution

# Week 14: Intro to Assembly

# Task 2

- execute an instruction the performs the operation 5 + 5 and store the result in register r2.

```
mov r0, #5
mov r1, r0
add r2, r0, r1
```

# Task 3

- Set r1 to hold a value of 229.

```
mov r1, #229
```

- Set r2 to hold a value of $2^{29} + 40$

```
mov r2, #40
movt r2, #0x1000
```

- Compute and store the sum in r0

```
add r0, r1, r2
```

# Task 4

- Shifting logically left by three bits.

lsl r0, 3

- Let's now reset register r0 to the 0x800000ff and to explore the impact of the different right shift operations
  - lsr -> 0x80000868
  - asr -> 0xE000003f
  - ror -> 0xE000003f

# Hacker Edition

```
mvn r0, r0
add r0, #1
```

- This is a 2's complement conversion.
  - First instruction bit flips values.
  - Second instruction increments by 1.

5 —> 0000 0101

flip —> 1111 1010
add 1 —> 1111 1011
-5 —> 1111 1011

# Question Code

What is the content of r2?

```
mov r0, #8
mov r1, #3
and r2, r0, #1
```

# Week 16: Assembly Development

# Number Addition

```
arr_sum:
 @ Add code to compute sum
 mov r1, 0
 mov r2, 0
add_loop:
 ldr r3, [r0, r1, lsl 2] @read address arr + 4*i (bytes)
 add r2, r3
 add r1, r1, 1
 cmp r1, #10 @ loop condition
 blt add_loop
 mov r0, r2 @ reutrn via r0 the result
 bx lr
```

# Mean Estimation

```
arr_mean:
 @ Add code to compute sum
 mov r1, 0
 mov r2, 0
mean_loop:
 ldr r3, [r0, r1, lsl 2] @read address arr + 4*i (bytes)
 add r2, r3
 lsr r2, 1      @ half the value
 add r1, r1, 1
 cmp r1, #10    @ loop condition
 blt mean_loop
 mov r0, r2     @ return via r0 the result
 bx lr
```

# Bubble Sort in C

```
void bubbleSort(int arr[], int size) {
  int swapped = 1;
  while(swapped) {
    swapped = 0;
  for (int j = 0; j < size - i - 1; j++) {
    if (arr[j] > arr[j + 1]) { // Swap arr[j] and arr[j + 1]
      int temp = arr[j];
      arr[j] = arr[j + 1];
      arr[j + 1] = temp;
      swapped = 1; // Mark that a swap occurred
    }
   }
  }
}
```

# Sorting Values

```
arr_sort:
  push {r4}
  @ r0 arr addr, r1 swapped, r2 counter
  @ r3 tmp
  add r0, #4 @ increament point to arr
+ 4
  mov r1, #0 @ initalize swap flag
  mov r2, #1 @ initialize counter
sort_loop:
  ldr r3, [r0, -4] @ r3 = *(r0 - 4)
(prev item)
  ldr r4, [r0] @ r4 = *r0
  @ If r3 > r4, swap elements
  cmp r3, r4
  ble skip_swap
  str r3, [r0] @ swap elements
  str r4, [r0, -4]
  mov r1, #1 @ record swap in flag
skip_swap:
```

```
  add r2, #1 @ r2++
  add r0, #4 @ r0++
  cmp r2, #10 @ check loop condition
  blt sort_loop @ If no swap was made,
                @ array is sorted
                @ complete function
  cmp r1, #0
  beq sort_finish @ Otherwise, reset
loop
  ldr r0, =arr
  add r0, 4
  mov r1, #0
  mov r2, #1
  b sort_loop
sort_finish:
  pop {r4}
  bx lr
```

# Binary Search

```
int binary_search(int arr[], int len, int target, int start, int end) {
 while (start <= end) {
  int mid = start + (end - start) / 2; // Prevent potential overflow
  if (arr[mid] == target) {
   return mid;
  } else if (arr[mid] < target) {
    start = mid + 1;
  } else {
    end = mid - 1;
  }

 }
 return -1; // Target not found
}
```

# Binary search

```
@ Binary Search Function
 @ Inputs: r0 = base address of array, r1 = size, r2 =
target
 @ Outputs: r0 = index if found, -1 if not found

binary_search:
 PUSH {r4, r5, r6, lr} @ Save return address
 mov r3, #0 @ r3 = left index, r4 = right index
 mov r4, #9
binary_loop:
 CMP r3, r4 @ Check if left <= right
 BGT not_found @ If left > right, element not found
 ADD r5, r3, r4 @ mid = (left + right) / 2
 LSR r5, r5, #1 @ Logical shift right (divide by 2)
 LDR r6, [r0, r5, LSL #2] @ Load arr[mid] CMP r6, r2 @
Compare arr[mid] with target
 BEQ found @ If arr[mid] == target, found
 BLT search_right @ If arr[mid] < target, search right
 BGT search_left @ If arr[mid] > target, search left
search_right:
 ADD r3, r5, #1 @ left = mid + 1
 B binary_loop
search_left:
 SUB r4, r5, #1 @ right = mid – 1
 B binary_loop
found:
 MOV r0, r5 @ Store index in r0
 POP {r4, r5, r6, lr} @ Restore return address
mov pc, lr @ Return
not_found:
 MOV r0, #-1 @ Return -1 if not found
 POP {r4, r5, r6, lr} @ Restore return address
 mov pc, lr @ Return
```

# Question Code

```
mov r0, #8 ; Initialize R0 with 8
mov r1, #1 ; Initialize R1 with 1
 LOOP:
cmp r0, #1
ble DONE
lsl r1, #1
sub r0, r0, #1
b LOOP
DONE:
mov r2, r1
```

# Week 17-18: Multiplication Game

# Introduction

- Multiplication Game with Assembly and the micro:bit
  - Press Button A to generate a random number between 1-100
  - Press Button B to generate a random number between 1-100
  - Press Buttons A + B to compute and display A * B
- Implement Ethiopian multiplication function in Assembly to realize multiplication.

# The template – setup code

```
// initialize the random number generator
uBit.seedRandom();

// Ensure that different levels of brightness can be displayed
uBit.display.setDisplayMode(DISPLAY_MODE_GREYSCALE);

// Set up listeners for button A, B and the combination A and B.
uBit.messageBus.listen(MICROBIT_ID_BUTTON_A, MICROBIT_BUTTON_EVT_CLICK, onButtonA);
uBit.messageBus.listen(MICROBIT_ID_BUTTON_B, MICROBIT_BUTTON_EVT_CLICK, onButtonB);
uBit.messageBus.listen(MICROBIT_ID_BUTTON_AB, MICROBIT_BUTTON_EVT_CLICK, onButtonAB);
```

# The template – event handlers

```
// Event handler for buttons A and B pressed together
void onButtonAB(MicroBitEvent e)
{
// DEVELOP CODE HERE
    int ret = eth_mult(val);
    uBit.display.print(ret);

}

// Event handler for button A
void onButtonA(MicroBitEvent e)
{
    val[0] = 1 + microbit_random(99);
    uBit.display.print(val[0]);
}

// Event handler for button B
void onButtonB(MicroBitEvent e)
{
    val[1] = 1 + microbit_random(99);
    uBit.display.print(val[1]);
}
```

```
extern "C"
{
    int eth_mult(int val[]);
}
```

# 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
```

```
bx lr
```

Copy lr address to pc

Branch to lr

# Convention 1: registers for procedure calls

`r0-r3:`

“argument” registers in which to pass parameters

`r0:`

return value registers

`lr:`

return address register (link register)

# Convention 2: Preserving registers

| 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.

# Passing Pointer Arguments

```
ldr r0, [pc, #36] @ (30 <_Z10onButtonABN5codal5EventE+0x30>)
bl 0 <eth_mult>
```

- `int eth_mult(int val[]);`
  - val is a pointer (32-bit value) to the first element of the array.
  - Address copied to register r0.
  - You need ldr/str to access the value.

```
bl 0 <eth_mult>
mov r1, r0
```

- Use register r0 to return the result.
  - ASM has no data types, just 32-bit values.
  - Variables are a C concept.
  - The compiler generates the right code for you, *as long as you follow conventions*.

# Ethiopian Multiplication

| A | B |
|---|---|
| 17 | 34 |
| 8 | 68 |
| 4 | 136 |
| 2 | 272 |
| 1 | 544 |

```
// Function to perform Ethiopian Multiplication
int ethiopian_mult(int a, int b) {
   int result = 0; // Initialize result to 0
   // the while loop stops when a==0.
   while (a >= 1) {
      if (a % 2 != 0) {
         // If 'a' is odd, add 'b' to the result
         result += b;
      }
      a = a >> 1; // Halve 'a'
      b = b << 1; // Double 'b'
   }
   return result; // Return the multiplication result
}
```

# Assembly Function

```
eth_mult:
@ r0 -> tmp, r1:a, r2:b, r3:result
  ldr r1, [r0] @ load a, b from mem
  ldr r2, [r0, 4]
  mov r3, #0 @ Set r3 = 0

loop:
  ands r0, r1, #0x1 @ Test even number
  beq skip_add @ If odd, skip (ands sets
  add r3, r2 @ N flag on PSR to 1 if 0)
skip_add:
  lsl r2, #1 @ Divide and multiply
  lsrs r1, #1 @ lsr/lsl update N, Z flags
  bne loop  @ If r1 is not zero, repeat.
  mov r0, r3 @ return result via r0.
  bx lr
```

# Hacker Edition – Negative Numbers

- Due to 2’complement arithmetics, Ethiopian multiplication does not work for negative numbers.
- Workaround:
  - Convert numbers to positive.
  - Note of exactly one is negative.
  - Run multiplication.
  - Convert result to negative if needed.

# Negative number Changes

```
push {r4}
@ r0:tmp, r1:a, r2:b, r3:result, r4: sign
  mov r4, #0     @ reset r4
  ldr r1, [r0]   @ load a & b from mem
  lsrs r3, r1, 31 @ Test signedness bit
  beq skip_abs   @ 2'complement abs
  add r4, 1      @ Note a is negative
  mvn r1, r1
  add r1, 1
skip_abs:
  ldr r2, [r0, 4]
  lsr r3, r2, 31 @ Test signedness bit
  beq skip_abs2
  add r4, 1      @ Note b is negative
  mvn r2, r2
  add r2, 1
skip_abs2:
  mov r3, #0     @ Reset result register
```

```
  mov r0, r3 @ Save return val

  @ Check for conversion
  ands r4, 0x1
  beq return @ if r4 is even, switch sign
  mvn r0, r0
  add r0, #1
return:
  pop {r4}
  bx lr
```

# Conclusion

- Week 14-17 task
  - Integration example of C++ and Assembly.
  - Optimize critical operations in assembly.
- Function conventions reminder.
- Hacker edition: negative numbers.

- Next time: Processing sound