1 Project Description

Write a program to implement the *bitonic sort* algorithm on a list of N integers. The program will print the unsorted list, sort it using the bitonic sort algorithm, and print the sorted list. Bitonic mergesort uses two recursive functions, *Red*Recursion and *Blue*Recursion, as well as two helper functions, *RedLoop* and *BlueLoop*, as described below.

2 Bitonic Sort Algorithm

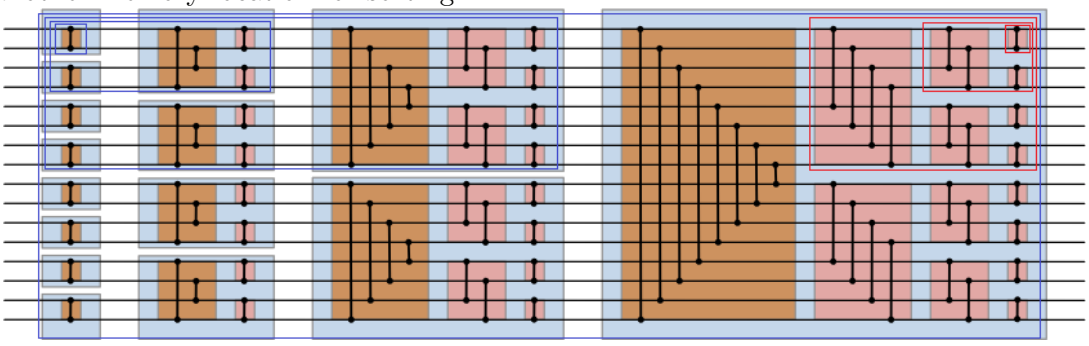
Bitonic sort is a divide-and-conquer algorithm for sorting a list by recursively sorting sub-lists. Let N be the size of the list to be sorted and M be the largest power of 2 less than N. The functions in bitonic sort are:

1. RedRecursion
   - find M
   - RedLoop: Sort all pairs of elements with indices separated by M
   - Perform red recursion on the first portion of the list (index 0 to $M-1$) and the second portion (index M to $N-1$) separately
2. BlueRecursion
   - find M
   - Perform blue recursion on the first portion of the list (index 0 to $M-1$) and the second portion (index M to $N-1$) separately
   - BlueLoop: Sort all pairs of elements with indices i and j such that $i+j=2M-1$
   - Perform red recursion on the first portion of the list (index 0 to $M-1$) and the second portion (index M to $N-1$) separately

Base case: If the length of the list is 1, then we do nothing (both red and blue recursion).

Bitonic sort: To perform bitonic sort, we just need to perform blue recursion on the full list.

In this project, we will be implementing an in-place bitonic sort, i.e., the list is not copied to another memory location for sorting.



Why does it work? To implement the algorithm, you do NOT need to understand why it works. But if you are interested, here is a brief description. A list is called bitonic if its values first increase (up to an arbitrary point) and then decrease, or vice versa. For example, the following are bitonic

1. 3 4 8 12 18 20 9 2
2. 20 9 2 3 4 8 12 18 20
3. 1 2 3 4 5 (note this special case)

However, this list is not bitonic:

1. 1 2 5 4 8

If we know a list is bitonic, we can sort it using a simple recursive strategy implemented in the RedRecursion function. First, using the RedLoop function, we swap elements in a way that produces two bitonic sublists: one on the first half and one on the second half. Importantly, the maximum of one list is smaller than the minimum of the other list. Thus, after this step, we can simply sort each half recursively using the red function, and the resulting list will be fully sorted.

We now define a function (BlueRecursion) to sort an arbitrary list recursively. First, we call the BlueRecursion function on the first half and the second half of the list, recursively. Now, the two halves are each sorted, but the full list is not. Then, the blue-merge step swaps elements around the center such that we obtain two bitonic sublists: one before the center and one after the center. The sublist before the center increases then decreases and the sublist after the center decreases then increases. Importantly, the largest value in the left sublist is smaller than the smallest value in the right sublist. Thus, if we sort each bitonic list separately, we are done. We sort each sublist by calling the RedRecursion function on each half separately.

3 Implementation

In this project, you have to write five functions, namely ***FindM***, ***RedRecursion***, ***BlueRecursion***, ***RedLoop*** and ***BlueLoop*** to implement the bitonic sort algorithm in the LEGv8 assembly language. Some details to note:

- Use the procedures prototype as mentioned below and given to you in the template. Don't change the registers or the arguments passed to the procedures or the values returned.
- Follow the "Procedure Call Convention" for calling procedures, passing registers and managing the stack. The procedures should not make any assumptions about the implementation of other procedures, except for the name of input/output registers and the conventions mentioned in the course.
- We expect your code to be well-commented. Each instruction should be commented with a meaningful description of the operation. For example, this comment is bad as it tells you nothing:

```
// x1 gets x2 - 1
sub x1, x2, #1
```

A good comment should explain the meaning behind the instruction. A better example would be:

```
// Initialize loop counter x1 to n-1
sub x1, x2, #1
```

3.1 Function 1: FindM(N)

Find the greatest power of 2 strictly less than N.

3.1.1 Parameters

- x0: N

3.1.2 Return Value

- x0: M, the greatest power of 2 less than N.

3.1.3 Examples

1.
Input:

- N: 8

Output:

- 4

2.
Input:

- N: 13

Output:

- 8

3.2 Function 2: RedLoop(a, N, M)

This function sorts all pairs of elements separate by M apart.

3.2.1 Parameters

- x0: Starting address of a (sub)list (corresponding to `a`.)
- x1: The number of integers in the (sub)list (corresponding to `N`.)
- x2: The greatest power of 2 less than x1 (corresponding to `M`.)

3.2.2 Return Value

- This function does not return anything.

3.2.3 Pseudo-code

function RedLoop(a, N, M)
    for i $\in \{0, \ldots, N - M - 1\}$ **do**
        if a[i] > a[i+M] **then**
            swap a[i] with a[i+M]
        end if
    end for
end function

3.2.4 Examples

1.
Inputs:

- a: 2 4 3 1
- N: 4
- M: 2

When exiting:

- a: 2 1 3 4

2.
Inputs:

- a: 7 6 5 8
- N: 4
- M: 2

When exiting:

- a: 5 6 7 8

3.3 Function 3: RedRecursion(a, N)

This function sorts a bitonic (sub)list.

3.3.1 Parameters

- x0: Starting address of a (sub)list (corresponding to `a`.)
- x1: The number of integers in the (sub)list (corresponding to `N`.)

3.3.2 Return Value

- This function does not return anything.

3.3.3 Pseudo-code

```
function RedRecursion(a, N)
    if N > 1 then
        M = FindM(N)
        RedLoop(a, N, M)
        RedRecursion(a, M)
        RedRecursion(address of a[M], N-M)
    end if
end function
```

3.3.4 Examples

1.
Inputs:

- a: 2 4 3 1
- N: 4

When exiting:

- a: 1 2 3 4

2.
Inputs:

- a: 7 6 5 8
- N: 4

When exiting:

- a: 5 6 7 8

3.4 Function 4: BlueLoop(a, N, M)

This function sort all pairs of elements with indices i and and $2M - 1 - i$.

3.4.1 Parameters

- x0: Starting address of a (sub)list (corresponding to `a`.)
- x1: The number of integers in the (sub)list (corresponding to `N`.)
- x2: The greatest power of 2 less than x1 (corresponding to `M`.)

3.4.2 Return Value

- This function does not return anything.

3.4.3 Pseudo-code

```
function BlueLoop(a, N, M)
    for i ∈ {2M − N, . . . , M − 1} do
        if a[i] > a[2M-i-1] then
            swap a[i] with a[2M-i-1]
        end if
    end for
end function
```

3.4.4 Examples

1.
Inputs:

- a: 2 5 6 7
- N: 4
- M: 2

When exiting:

- a: 2 5 6 7

2.
Inputs:

- a: 2 5 6 7 1 3 4 8
- N: 8
- M: 4

When exiting:

- a: 2 4 3 1 7 6 5 8

3.5 Function 5: BlueRecursion(a, N)

This function sorts a (sub)list with a size that is a power of 2.

3.5.1 Parameters

- x0: Starting address of a (sub)list (corresponding to `a`.)
- x1: The number of integers in the (sub)list (corresponding to `N`.)

3.5.2 Return Value

- This function does not return anything.

3.5.3 Pseudo-code

```
function BlueRecursion(a, N)
    if N > 1 then
        M = FindM(N)
        BlueRecursion(a, M)
        BlueRecursion(address of a[M], N-M)
        BlueLoop(a, N, M)
        RedRecursion(a, M)
        RedRecursion(address of a[M], N-M)
    end if
end function
```

3.5.4 Examples

1.
Inputs:

- a: 2 5 7 6
- N: 4

When exiting:

- a: 2 5 6 7

2.
Inputs:

- a: 2 5 7 6 8 1 3 4
- N: 8

When exiting:

- a: 1 2 3 4 5 6 7 8