1 Introduction

The prefix sum is a useful primitive with many applications in engineering and computer science. It is computed very simply: Given an input sequence \( a = \{a_1, a_2, \ldots, a_n\} \), the prefix sum of \( a \) is the sequence \( b = \{b_1, b_2, \ldots, b_n\} \) where

\[
b_i = \begin{cases} 
  a_i & \text{if } i = 1 \\
  a_i + b_{i-1} & \text{if } 1 < i \leq n
\end{cases}
\]

This makes the prefix sum a sort of "running total" algorithm; as we scan \( a \), we accumulate its sum in \( b \).

In this assignment, you will design hardware that computes the prefix sum of an input sequence. There are three main parts in this design:

1. The Moore state machine design, which is responsible for control signals.
2. The synchronous data path design, which is responsible for computing parts of the output sequence.
3. The top level design, which incorporates the design and serves as I/O to the test benches.

The state machine module is contained in `controller.vhd`, the datapath is contained in `datapath.vhd`, and the top level module is `pa3.vhd`. Additionally, `array.vhd` contains the definition for the `array_32` type, which is used as the input/output of `pa3.vhd` (you should not modify this file). There are three test benches: `prefix_tb1.vhd`, `prefix_tb2.vhd`, and `prefix_tb3.vhd`. More details will be provided later on each module. Goals of this assignment are to implement the design using VHDL:

- Implement the datapath design.
- Implement a Moore state machine to provide control to the datapath.
- Evaluate design correctness and performance using test benches.

2 Background

Calculating the prefix sum sequentially is a trivial task. Let us assume that \( a \) and \( b \) are implemented as \( n \) length arrays for our algorithm. We could use the following algorithm to compute the prefix sum sequentially:

```
Algorithm 1 Sequential Prefix Sum

procedure PREFIXSEQ(a[1 : n], n)
  b[1] ← a[1]
  for i = 2 to n do
    b[i] ← b[i - 1] + a[i]
  end for
  return b[1 : n]
end procedure
```
Since we are iterating through the sequence once, this algorithm is \(O(n)\) in terms of the sequence length \(n\). However, assuming that addition and position shifting can be done in parallel, we can increase the runtime efficiency (we assume that \(n\) is a power of 2 for simplicity):

**Algorithm 2** Parallel Prefix Sum

```plaintext
procedure PrefixParallel(a[1 : n], n)
    b[1 : n] ← a[1 : n]
    for i = 0 to \log n - 1 do
        b[1 : n] ← b[1 : n] + Rshift(b[1 : n], 2^i, n)
    end for
    return b[1 : n]
end procedure

procedure Rshift(x[1 : n], k, n)
    y[1 : k] ← 0
    for i = k + 1 to n do
        y[i] ← x[i - k]
    end for
    return y[1 : n]
end procedure
```

As you can see, this parallel algorithm is only \(O(\log n)\) time, but requires \(n\) adders, shifters, and registers rather than 1, as well as hardware that can "shift over" register values in constant time. This is clearly difficult to scale as \(n\) increases.

It may appear that we must choose between either hardware area and speed of computation, but in our design, we will strike a balance between the two: we will create a sequential implementation that uses elements of the parallel adder to give a speedup over the classic sequential implementation. If we have access to \(n\) adder/shifter/register elements, and the total array size is now \(m\), we can compute the output with a total of \(m \cdot n \cdot (\log n + 1)\) operations (accounting for loading input values into registers).

### 3 Design Overview

#### 3.1 Algorithm and Examples

In our design, you will implement the partially-parallel algorithm to compute the prefix sum. You will use a total of \(n\) adder/shifter/register elements in the datapath. The pseudocode for the design is given in Algorithm 3 (you may assume that the input array size \(m\) is a multiple of \(n\)):

**Algorithm 3** Partially-Parallel Prefix Sum

```plaintext
1: procedure PartialPrefix(a[1 : m], n, m)
2:    r[1 : n] ← zeros  \(\triangleright\) Initialize an empty accumulator array length \(n\)
3:    for i = 1 to i = m - n + 1 in increments of \(n\) do  \(\triangleright\) Parallel load
4:        r[1 : n] ← a[i : i + n - 1] + [r[n], 0, \ldots, 0]  \(\triangleright\) Compute \(n\) elements in parallel in \(\log n\) cycles
5:    end for
6:    if i > 1 then  \(\triangleright\) Parallel store of previous iteration output
7:        b[i - n : i - 1] ← r[1 : n]
8:    end if
9:    b[m - n + 1 : m] ← r[1 : n]  \(\triangleright\) Parallel store of final iteration
10:   return b[1 : m]
11: end procedure
```

Note that in this implementation, you will be always using 4 parallel adders/shifters/registers (i.e. \(n = 4\),
and the input size will always be a multiple of 4. **Note that when** $m > n$, **lines 4 and 7 are performed in parallel after the first iteration.** This enables the design to perform loads and stores to the input and output arrays in parallel. For a better understanding of the design, two examples are shown below.

### 3.1.1 Example 1: $m = 4$

Let’s say our input array is $a = [1, 2, 3, 4]$. We can use the above algorithm to compute the output $b$. Note that $n = 4$, $m = 4$, so the FOR loop will only execute once.

- **Cycle 1: Parallel load**
  
  $$r[1:4] \leftarrow a[1:4] + [r[4], 0, 0, 0] = [1, 2, 3, 4] + [0, 0, 0, 0] = [1, 2, 3, 4]$$

- **Cycle 2: Shift and add 1**
  
  $$r[1:4] \leftarrow r[1:4] + [0, r[1], r[2], r[3]] = [1, 2, 3, 4] + [1, 2, 3] = [1, 2, 3, 5, 7]$$

- **Cycle 3: Shift and add 2**
  
  $$r[1:4] \leftarrow [r[1:4] + [0, 0, r[1], r[2]] = [1, 3, 5, 7] + [0, 0, 1, 3] = [1, 3, 6, 10]$$

- **Cycle 3+: Parallel Store and Termination**
  
  $$b[1:4] \leftarrow r[1:4] = [1, 3, 6, 10]$$

After the third cycle, $b$ will contain the correct result of $b = [1, 3, 6, 10]$

### 3.1.2 Example 2: $m = 8$

Now let’s increase our input size and say $a = [1, 2, 3, 4, 5, 6, 7, 8]$. Note that $n = 4$, $m = 8$, so now the FOR loop will execute twice!

- **Cycle 1: Parallel load**
  
  $$r[1:4] \leftarrow a[1:4] + [r[4], 0, 0, 0] = [1, 2, 3, 4] + [0, 0, 0, 0] = [1, 2, 3, 4]$$

- **Cycle 2: Shift and add 1**
  
  $$r[1:4] \leftarrow r[1:4] + [0, r[1], r[2], r[3]] = [1, 2, 3, 4] + [1, 2, 3] = [1, 2, 3, 5, 7]$$

- **Cycle 3: Shift and add 2**
  
  $$r[1:4] \leftarrow [r[1:4] + [0, 0, r[1], r[2]] = [1, 3, 5, 7] + [0, 0, 1, 3] = [1, 3, 6, 10]$$

- **Cycle 4: Parallel load (and previous iteration store)**
  
  *$r[1:4] \leftarrow a[5:8] + [r[4], 0, 0, 0] = [5, 6, 7, 8] + [10, 0, 0, 0] = [15, 6, 7, 8]$*

  *$b[1:4] \leftarrow r[1:4] = [1, 3, 6, 10]$*  
  *Occurs in parallel*

- **Cycle 5: Shift and add 1**
  
  $$r[1:4] \leftarrow r[1:4] + [0, r[1], r[2], r[3]] = [15, 6, 7, 8] + [0, 15, 6, 7] = [15, 21, 13, 15]$$

- **Cycle 6: Shift and add 2**
  
  $$r[1:4] \leftarrow r[1:4] + [0, 0, r[1], r[2]] = [15, 21, 13, 15] + [0, 0, 15, 21] = [15, 21, 28, 36]$$

- **Cycle 6+: Termination**
  
  $$b[5:8] \leftarrow r[1:4] = [15, 21, 28, 36]$$

After the sixth cycle, $b$ will contain the correct result of $b = [1, 3, 6, 10, 15, 21, 28, 36]$

### 3.2 Structure of PA Modules

The overall design is split into three components i) The PA3 top module, ii) The state machine controller module, and iii) The datapath module. The overall figure of the PA structure with each module’s ports is shown below:
In the top module `pa3.vhd`, the design performs port maps and the `ready` signal logic. In `controller.vhd`, a state machine keeps track of the algorithm phase and provides the corresponding control signals (`load`, `shift[1:0]`, and `ready_flag`), starting load (LDX) index for input array, and starting store index (SDX) for the output array. In `datapath.vhd`, the inputs `a`, `clk`, and `reset` are ported from the top module, while the inputs `load`, `shift`, LDX, and SDX are generated by the controller. The output `b` is ported back through the top module to the test benches.

### 3.3 PA3 Top Module

The input array `a` is initialized in the testbench (using `array.vhd`). Additionally, the `m` integer is initialized in the testbench with size of the input array. For example, for an array of four unsigned integers, `m=4`. You may assume that the size of input arrays are always a multiple of 4. The `clk` and `reset` signals are also from the testbench. The output array `b` is initialized in the testbench, and updated in the datapath synchronously. It contains the prefix sum values as they are computed by the design. The signal `ready` is written to the testbench to indicate when the computation is completed. It takes the `ready_flag` from the controller and delays its by a cycle to indicate the testbench that all updates to the output array are complete.

The port maps for the controller and datapath are provided, as is the `ready` signal synchronization process. Therefore, the `pa3.vhd` module should not be modified.
3.4 Datapath

In `datapath.vhd`, the logic for producing the output must be implemented correctly given the control signals `load` and `shift[1:0]` produced by the state machine. The structure of the datapath is given below:

![Datapath Diagram]

Figure 3.2: Inner operation of the datapath

Four elements at a time are loaded into the datapath from `a`, with the first being indexed with `LDX`. As such, `a(LDX)`, `a(LDX+1)`, `a(LDX+2)`, and `a(LDX+3)` are used to perform a parallel load of the appropriate array indices for parallel prefix sum computations. Additionally, the registers `R1`, `R2`, `R3`, and `R4`, and the output array `b` are updated synchronously. The outputs are indexed into the 4 elements of `b` given by `b(SDX)`, `b(SDX+1)`, `b(SDX+2)`, and `b(SDX+3)`.

Within each "Input Select Logic" block, you must use `load` and `shift` as MUXing logic to select the correct input into each adder. There are 9 possibilities for each input: i) The indexed array values `a(LDX) - a(LDX+3)` ii) The four internal register values `R1 - R4` or iii) 0 (zero). The `load` and `shift` signal values will correspond to a certain operation that needs to be performed (i.e. load from input and accumulate, shift registers by 1 and add, shift registers by 2 and add, or hold register values) based on the state of the controller module (discussed in the next section). The addition block outputs should be combinational with your selected inputs, and the addition block should just be implemented using the "+" operator.

When the reset signal is asserted, *asynchronously* the outputs `R1 - R4` are cleared to zero. Otherwise on the rising edge of the clock, the outputs of the adder modules update the registers `R1 - R4`. Also on the rising edge, the output entries `b(SDX) - b(SDX+3)` are also updated with current values of registers `R1 - R4`.

5
3.5 State Machine

In the state machine module `controller.vhd`, the control signals are produced for `pa3.vhd` and `datapath.vhd`. You are responsible for designing the state machine that produces the datapath control signals with correct transition logic. Guidelines for the state machine are given below:

- The state machine module should have one state for each phase (i.e. "Start / First Load", "Load", "Shift by 1 and Add", "Shift by 2 and Add", and "Done"). You can follow the examples given in section 3.
- The state definitions along with the "current state" signal will be internal to the `controller.vhd` module.
- The state machine must keep track of the required number of iterations to complete the algorithm, as well as what the current iteration is. The internal signal `iter` can keep track of the current iteration, and the `m` input signal (input array size) is used to determine when to transition to the "Done" state.
- When `reset` is asserted, the state machine must asynchronously transition to the "Start / First Load" state. Additionally, any signals used to keep track of iterations should be reset.
- As a Moore machine, state transitions should occur on the rising edge of the clock. Additionally, all outputs (`LDX`, `SDX`, `load`, `shift[1:0]`, and `ready_flag`) are only a function of the current state.
- The `LDX` and `SDX` outputs index into the input / output arrays in the datapath module, and as such, use `iter` in the logic for their computation. It is important to be careful about managing these to ensure they are reading/writing to correct and valid locations in `a` and `b`. During the parallel load cycle, you need to be storing to the previous iteration output location as shown in algorithm 3.
- It is important that the control signals `load` and `shift[1:0]` are set up correctly for the datapath. The following describes how the states control the datapath:
  1. "Start / First Load": This state will be transitioned to after `reset` is asserted, and as such, the datapath outputs (and thus accumulation) should be zero. The datapath control signals should indicate that a load from input array `a` is occurring.
  2. "Shift by 1": The first shift after loading new input data, where `R1` - `R4` need to be added with their shifted-by-one equivalents. The datapath control signals should indicate that a shift-by-1 and add is occurring.
  3. "Shift by 2": The second shift, where the previously computed `R1` - `R4` need to be added with their shifted-by-two equivalents. The datapath control signals should indicate a shift-by-2 and add is occurring.
  4. "Load": When `m > 4`, there will be more than one iteration, so additional loads will take place besides the initial one. Similar to the "Start / First Load" state, the datapath control signals should indicate that a load from input array `a` is occurring.
  5. "Done": The last iteration output is updated in output `b`. As all iterations have completed, the `ready_flag` is asserted indefinitely until reset. To ensure datapath continues to function properly, the `R1` - `R4` signals should be summed with zero indefinitely (to hold the outputs) until a `reset` is asserted.
- The `ready_flag` signal is forwarded by the top module on the rising edge of the clock to indicate that the computation is finished. Therefore, `ready_flag` is asserted one cycle after the state machine transitions to "Done".

4 Test Bench

There are three total test benches: `prefix_tb1.vhd`, `prefix_tb2.vhd`, and `prefix_tb3.vhd`. Each of these test benches calculates a different number of entries. `prefix_tb1.vhd` is computing a length-4 array, `prefix_tb2.vhd` is computing a length-12 array, and `prefix_tb1.vhd` is computing a length-32 array. In each test bench, you must do the following:
1. Set the input array to the specified entries (entries outside of the a(1:m) range should be 0):
   - prefix_tb1.vhd: \( a = (1, 2, 3, 4); m = 4 \)
   - prefix_tb2.vhd: \( a = (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89); m = 12 \)
   - prefix_tb3.vhd: \( a = (2, 2, 2, \ldots, 2); m = 32 \)

   Note that entries of the array are of type unsigned(31 downto 0). Make sure when entering the input data, it is properly converted to a 32-bit value.

2. Calculate the expected output array for the given input arrays (either by hard-coding the result, or using a for-loop or similar method), and store it in an "expected output" array.

3. After the computation is finished, you must assert that the output \( b \) is equal to the expected output by using \texttt{assert}. Similar to PA2, if the output is different from what is expected, you should \texttt{report} an error message with a severity level of WARNING. Otherwise, you should print a success message, \textit{along with the number of cycles it took to complete the computation.} Note that the simulation will not halt in the case of an error, since we are setting the severity to WARNING. If there is an error (and as such, the error message is reported to the console), then the success message should not also be reported (or vice versa).

   You may use any method you choose to count the number of clock cycles that have passed. One method is to put a counter inside of the clock process that increments every rising edge. Another is to count how many clock cycles have passed since starting the state machine until \texttt{ready} is asserted. In either case, the counter should start the cycle after \texttt{reset} has been de-asserted, and the counter should end once \texttt{ready} is asserted by the design. Printing the number of cycles to the TCL console can be done with \texttt{report}. For example, if you store the number of cycles in an integer variable called "cycles", this value can be printed to the terminal using the following:

   \texttt{report "Success, computation took " & integer'image(cycles) & " cycles";}

5 Deliverables

Please submit a single PDF containing the following:

1. Your code of the following:
   - controller.vhd
   - datapath.vhd
   - prefix_tb1.vhd
   - prefix_tb2.vhd
   - prefix_tb3.vhd

2. The screenshots listed below. For the waveform screenshots, please \textbf{clearly} show \( a, b, m, \texttt{reset}, \texttt{clk}, \texttt{ready}, \texttt{LDX}, \texttt{SDX}, \texttt{load}, \texttt{shift[1:0]}, \texttt{state}, \) and R1 - R4. Some of these will need to be added to the waveform from the controller / datapath modules.
   - The output waveform from prefix_tb1.vhd from 0ns to 100ns.
   - The output waveform from prefix_tb2.vhd from 0ns to 130ns
   - The output waveform from prefix_tb3.vhd from 240ns to 320ns
   - The TCL consoles from the above 3 test benches. Make sure that your \texttt{report} messages and clock cycle numbers are clearly shown.