Problem 1: Circuit Design

Consider circuits used to calculate “bit tally”—the sum of the number of “1” bits in a word. For example, the bit tally of 0010010 is 2, and the tally bit of 1101101 is 5.

(a) Serial Solution:

Using only the components listed below, design a tally circuit for an 8-bit input word, X, initially stored in a set of 8 flip-flops. Your circuit should complete operation in 8 clock cycles. (You don’t need to design the part of the circuit for loading new values.) Try to minimize the amount of hardware resources.

Library of components:

- Flip-flop (FF) with clock enable (CE), FF initially has value 0
- Full-adder (FA) cell
- simple 2-input logic gates (AND, OR, EXOR)
- 2 to 1 mux

Below, briefly explain the theory of operation behind your circuit. What happens on each clock cycle?

<table>
<thead>
<tr>
<th>Solution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>You can create an accumulator circuit (FA cell with a FF) to produce the tally, selecting the next bit by using a counter circuit to generate the select bits for an 8-1 MUX created from 7 2-1 MUXes. Each clock cycle, the tally is updated until the counter is finished.</td>
</tr>
</tbody>
</table>
(b) Parallel Solution:

Now assume that we would like a tally circuit that completes it’s operation in 1 clock cycle. Again, using the supplied library of components, draw below an appropriate circuit. Again, try to minimize the amount of hardware resources.

Below, briefly explain the theory of operation behind your circuit. How does your circuit arrive at the proper output?

Solution:

We can use FA cells to tally groups of 2-3 bits \( X[0:2], X[3:5], \) and \( X[6:7] \) creating 3 2-bit numbers. We can sum the tallies of \( X[0:2] \) and \( X[3:5] \), creating a 2-bit number with a carry out that we can use as the carry-in when adding with the result of the tally of \( X[6:7] \). This requires in total 7 FA cells.
Problem 2: Computing SAXPY (Single-precision A.X Plus Y)

In this problem, we would like to ask you to map a software code, SAXPY – a common linear algebra routine, to an efficient hardware implementation.

```c
for (i = 0; i < 1024; i = i + 1)
    z[i] = a * x[i] + y[i];
```

where \(x\), \(y\), and \(z\) are 32-bit single-precision floating point 1024-element arrays stored in three separate 1024 \(\times\) 32-bit dual-ported, synchronous memory blocks (one cycle read/write per memory port). \(a\) is a 32-bit float scalar input. A 4-stage pipelined multiplier and 3-stage pipelined adder are used to implement floating-point multiplication and addition, respectively. You may use as many multipliers, adders, registers, comparators, muxes, or logic gates as you like.

Draw a hardware circuit, including both datapath (computation) and control (read/write to the memory blocks), to compute 1024 elements of \(z\) using the minimum number of cycles. Hint. You may transform the software code to match your circuit implementation.
Solution:

Since the memory blocks are dual-ported, we can read/write two array elements of each memory block in the same cycle. Therefore, we unroll the loop by a factor of 2 for parallel execution.

```c
for (i = 0; i < 1024; i = i + 2) {
    z[i + 0] = a * x[i + 0] + y[i + 0];
    z[i + 1] = a * x[i + 1] + y[i + 1];
}
```

In addition, as the floating-point multiplier and adder are pipelined, they can accept new input every clock cycle. The multiplier takes 4 cycles to produce a result, and 3 cycles for the adder. To make sure that the correct input of $y$ arrives in the same cycle with the result of $x \times a$, additional registers are required. Another acceptable solution is delaying reading $y$ by 4 cycles.

Below is the datapath and control diagram shown the hardware blocks as well as their connections (some details are omitted for brevity – see the complementary Verilog code). The system is fully pipelined. It will take $1 + 4 + 3$ cycles to compute the first two $z$s, and 511 following cycles to compute the remaining $z$s, and one additional cycle to write the final $z$s to MEM_Z. Thus the total number of cycles required to calculate 1024 elements of $z$ is 520 cycles.

```verilog
// MEM_X
assign MEM_X_addr0 = xy_q + 0;
assign MEM_X_addr1 = xy_q + 1;
```
Problem 3: Tree Engine

In this problem, you are asked to design a hardware engine for traversing a tree data structure stored in memory. The tree is represented in memory as a collection of nodes. Each node contains three n-bit fields: `value` holds a two’s complement integer, `left` holds a pointer to the left subtree, and `right` holds a pointer to the right subtree. All 0’s in a pointer field is used to indicate a “null” pointer (no subtree). Trees are stored in an n-bit wide memory block. The fields of a node, `value`, `left`, `right`, are stored in consecutive memory locations. The tree root node always begins at address 0.

Your task is to design the datapath and specify the controller for a hardware block that traverses a tree and outputs the maximum node value (the max over all the `value` fields of all the tree nodes). Your circuit should output the final max value, (max), and a done signal (done). Your design objective is to minimize the number of clock cycles required for a tree traversal, then the cost.

You are allowed to make instances of the following design blocks:

- Asynchronous read memory with single read port, n-bit wide address (`addr`), and n-bit wide data output (`dout`). This block is given to you preinitialized storing a tree.
- n-bit wide register with clock enable (ce), and reset (rst) inputs.
- n-bit wide 2-to-1 multiplexor.
- n-bit wide adder/subtractor block with function control (SUB).
- n-bit wide “equal zero” comparator. Takes an n-bit word and outputs a 1 if the input is =0.
• n-bit wide stack block, with clock enable (ce), synchronous push/pop control (PUSH), and a empty flag (EMPTY). (For a push operation, set PUSH=1, for pop, set PUSH=0.) The top of the stack is always available on the output dout. The data input din is used for the pushing data onto the stack (when PUSH = 1). EMPTY is an output signal that is =1 when the stack contains no elements.

• Simple logic gates.

(a) In “register transfer language” style pseudo-code, write out the control algorithm. (Remember to use “,” to separate transfers that occur on the same clock cycle and “;” for transfers on consecutive clock cycles.)

Solution:

The high level pseudo-code for finding the max value in a tree using an iterative preorder tree traversal algorithm:

max = 0
stack = []
stack.push(root)
while(stack != empty):
    node = stack.pop()
    if node.value > max:
        max = node.value
    if node.right:
        stack.push(node.right)
    if node.left:
        stack.append(node.left)

done = 1

Translating to “register transfer language” style, using the flags and pins of the given hardware:

// push root node on stack, create variable regs
NODE <- 0, DONE <- 0, MAX <- 0,
STACK_CE <- 1, STACK_PUSH <- 1, STACK_DIN <- 0;
repeat {
    // pop from stack
    STACK_CE <- 1, STACK_PUSH <- 0, NODE <- STACK_DOUT;
    // fetch node value from memory
    STACK_CE <- 0, VALUE <- Memory[NODE],
    // check if node value greater than current max
    if (VALUE > MAX)
        MAX <- VALUE;
    // fetch right pointer from memory, push onto stack if not null
    RIGHT <- Memory[NODE + 2*n],
if (RIGHT != 0)
    STACK_CE <- 1, STACK_PUSH <- 1, STACK_DIN <- RIGHT;
// fetch left pointer from memory, push onto stack if not null
LEFT <- Memory[NODE + n],
if (LEFT != 0)
    STACK_CE <- 1, STACK_PUSH <- 1, STACK_DIN <- NODE.left;
} until (STACK_EMPTY == 1);
DONE <- 1;

(b) In the space below, neatly draw your datapath of the tree engine. Circle the control signals—
the signals you intend to generate from a controller external to the datapath, or signals from
the datapath back to the controller. Label all circuit inputs and outputs.

Solution:
Reasonable Datapath given register transfer language.
Work in progress solution below:

Problem 4: Scheduling

Assume we have a datapath with three arithmetic blocks: two adders (add1), and (add2), and a
single pipelined multiplier (mult). The multiplier has two pipeline stages.

The arithmetic blocks are interconnected with a set of switch elements, so that any block can send
its output to any other block. Furthermore, the blocks are separated with pipeline registers.

The datapath is connected to a memory block that holds multiple arrays of data values, A, B, C, D
and E. Up to 4 data values can be read from the memory on a single clock cycle.

Your goal is to generate the schedule for iterative execution of the computation graph shown below
on the datapath described above using as few cycles as possible.

The result value \( r_i \) for each iteration of the graph is sent out of the datapath as an output signal
(not written to the memory).
Without applying any algebraic manipulation, or modifying the structure of the graph, derive the information needed for scheduling the computation, and illustrate your solution by filling in the schedule chart for four iterations of the computation (generating \( r_0, \ldots, r_3 \)).

<table>
<thead>
<tr>
<th>resource</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Solution:
Problem 5: Questions from Deep Neural Networks Lecture

(a) Two different microarchitectures were presented in the DNN lecture. Name and describe the differences between the two.

Solution:
There is Spatial and Layer-based Design.

(b) What is the purpose of the line buffer in the hardware implementation of neural networks?

Solution:
The line buffer’s function is similar to a cache, buffering inputs to perform spatial operations and reuse to improve arithmetic intensity.

(c) High Level Synthesis:

- Describe several advantages of using HLS vs RTL descriptions in Verilog.

Solution:
HLS allows for abstraction and makes writing routines like loops much easier without having to worry about low level implementation. This reduces development time significantly and the number of errors.

- List several limitations of HLS compilers.
<table>
<thead>
<tr>
<th>Solution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLS compilers may not generate the most efficient or highest performance design, since there is no control over the low level implementation.</td>
</tr>
</tbody>
</table>