EECS 151/251A Homework 9

Instructor: Prof. John Wawrzynek, TAs: Christopher Yarp, Arya Reais-Parsi

Due Monday, Apr 22nd, 2019

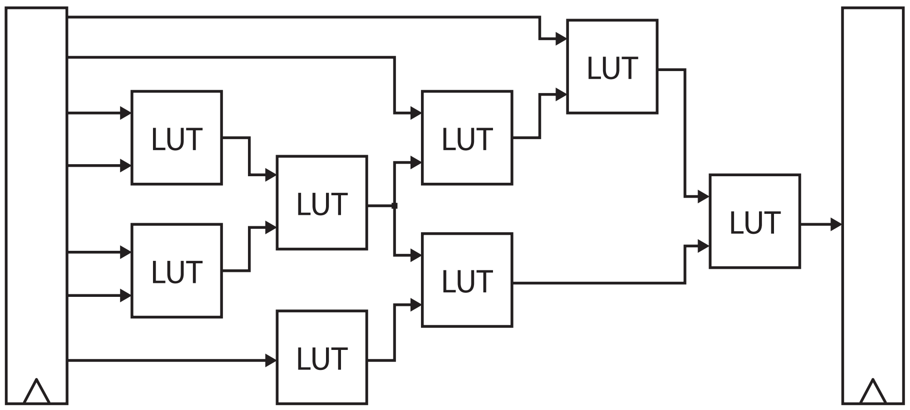
Problem 1: Pipelining for Speed [8 pts]

Given the circuit shown below with $\tau_{clk-q} = \tau_{setup} = 50ps$, and $\tau_{LUT} = 100ps$.

(a) Using pipelining, find the number of stages where the clock frequency is at least 2 times the original (unpipelined) clock frequency.

(b) Find the number of stages where the clock frequency is at least 3x the original.

Assume that LUTs are indivisible (cannot be internally pipelined). For each question above, draw dashed lines in the circuit diagram to indicate where to add registers.

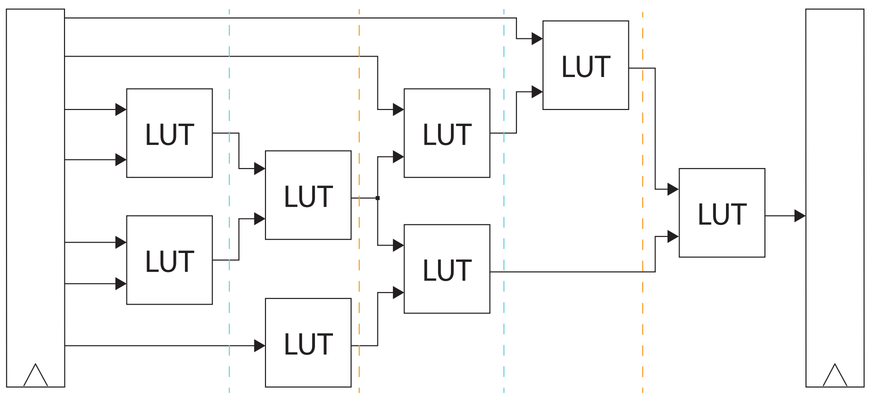


Solution:

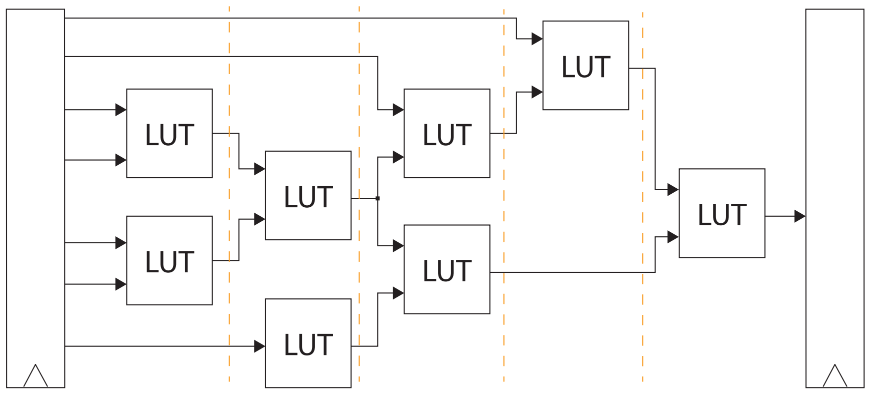
The original critical path was $\tau_{clk-q} + 5\tau_{LUT} + \tau_{setup} = 50ps + 5(100ps) + 50ps = 600ps$.

For pipelining to increase the clock frequency by at least 2x, the critical path must be reduced to be $\leq 300ps$. This can only be accomplished with a critical path of a maximum of 2 LUTs in series per between pipeline registers. *Remember that when pipeline, you need to make sure you match the delay on parallel paths. Otherwise, signals will not be aligned when arriving at the combinational logic blocks..*

Possible pipeline insertion points are shown below. Each color represents a possible solution.



To increase the clock frequency by a factor of 3, the critical path must be reduce to be $\leq 200ps$. This can only be accomplished with a critical path of a single LUT between registers. In this case, there is only one possible pipelining solution which is shown below.



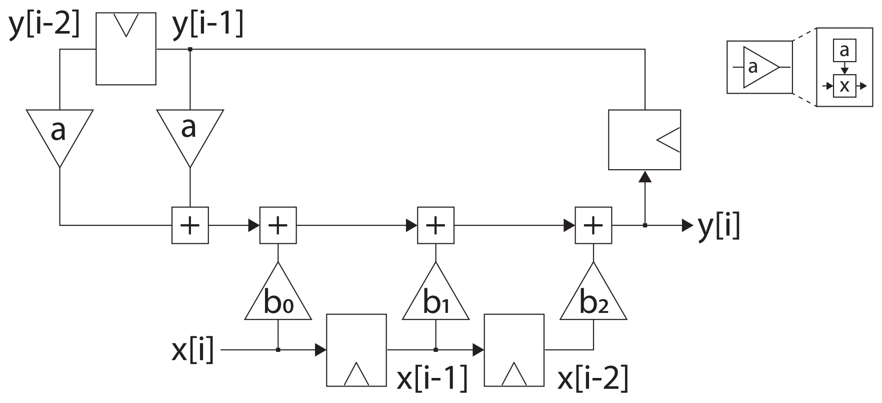
Problem 2: Pipelining in the Presence of Feedback [6 pts]

For the following streaming computation, draw a circuit that uses inter-operator pipelining to maximize the clock frequency.

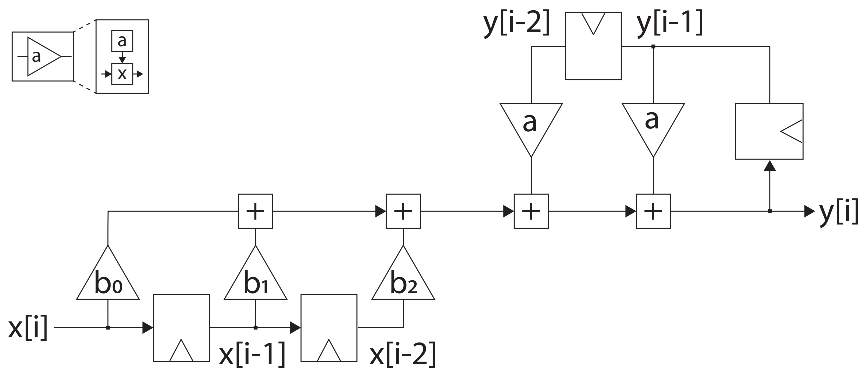
$y_i = ay_{i-1} + ay_{i-2} + b_0x_i + b_1x_{i-1} + b_2x_{i-2}$

Solution:

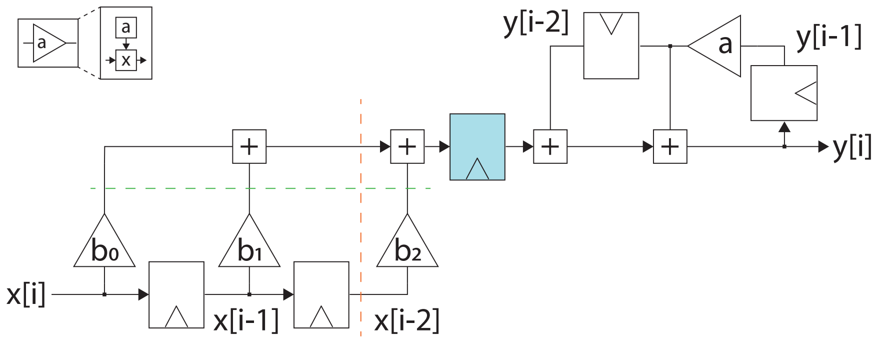
One direct implementation of the above expression is shown below:



The problem with this implementation is that is offers very little opertunity to pipeline because of the feedback loop. However, we can modify the implementation using the assocative property of addition. We re-arrange the adders in the adder chain to minimize the amount of logic in the feedback loop. The design after this transformation is shown below:



One other transform we can make to this design is to merge the two multiplies by `a` into a single multiplier. This is not required for this problem but does result in less logic.



In this solution, the blue pipeline register separates the feed-forward section from the feedback section. The feed forward section can be further pipelined along either of the dashed lines. The decision of which to choose depends on whether the critical path is two sums in series or a sum and product in series. If two sums in series is the critical path, inserting pipeline stages along the orange line is most beneficial. If the product and sum is the critical path, then pipelining along the green line is most beneficial. Note that it is possible for a constant multiply to have less delay than an add - particularly when the multiplication is by a power of 2.

Problem 3: Pipelining in the Presence of Feedback [7 pts]

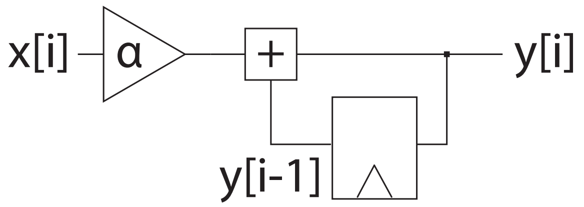
Consider the design of a datapath to implement the following streaming computation: $y_i = \alpha x_i + y_{i-1}$.

(a) Draw the diagram of a circuit that has a throughput of 1 result per cycle. Use inter-operator pipeline register(s) to maximize clock frequency.

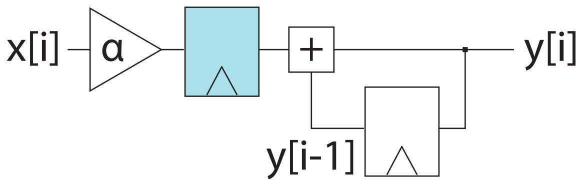
(b) Draw the diagram of a circuit that has a throughput of 8 results per cycle and operates at a high frequency. *Hint: unroll the loop and use rearrangement of the operators to maximize the clock frequency.*

Solution:

Part a: The direct implementation of the design is shown below:



There are limited opportunities for pipelining this structure because the feedback loop is already as small as possible (1 operation in the loop). The pipelined version of this design inserts a register after the feed-forward section of the design. The pipelined version is shown below:

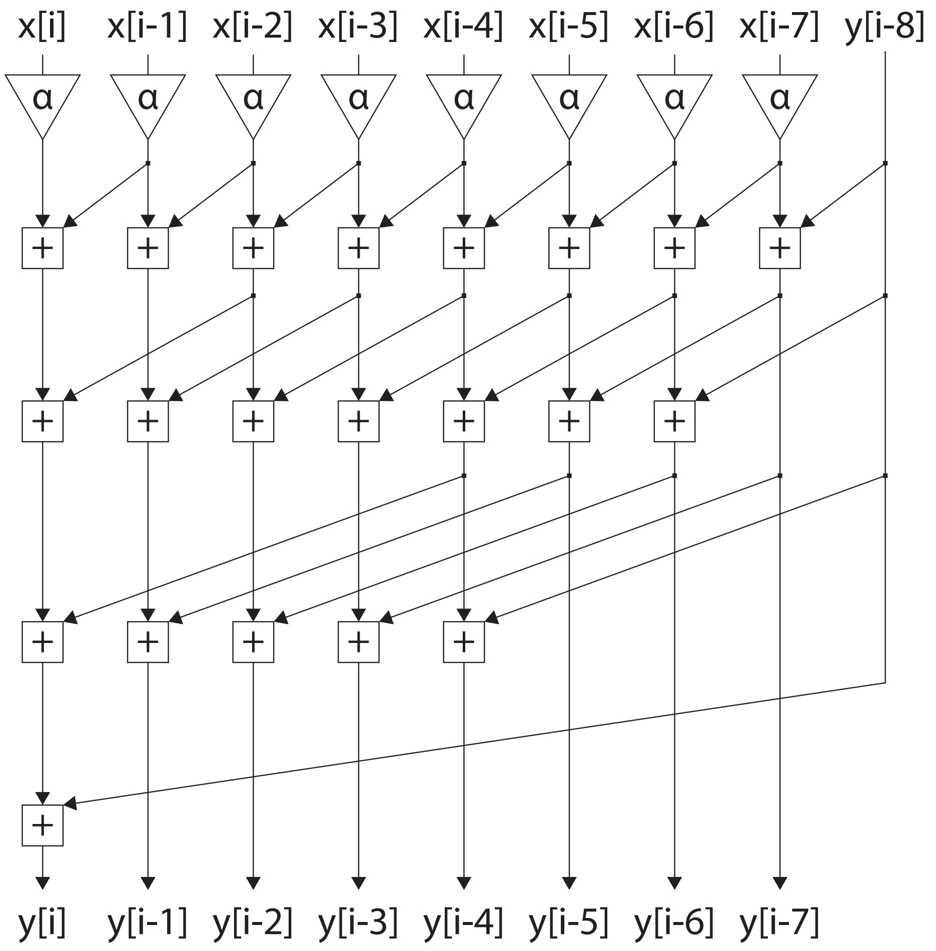


Part b: In this part of the problem, we are asked to modify the design so that it outputs 8 results per cycle. However, at first glance, this seems diffuclt due to the feedback loop. One strategy to deal with this is called loop-unrolling. In this strategy, we elaborate the logic for multiple itterations of the loop at once and work on blocks of input data rather than a single value. We can accomplish this by substituting the expression for y_i for the term y_{i-1}. Note that we need to make the appropriate changes to the index when we do this.

$$\begin{aligned}
y_i &= \alpha x_i + y_{i-1} &&= \alpha x_i + \alpha x_{i-1} + \alpha x_{i-2} + \alpha x_{i-3} + \alpha x_{i-4} + \alpha x_{i-5} + \alpha x_{i-6} + \alpha x_{i-7} + \\
& && \qquad y_{i-8} \\
y_{i-1} &= \alpha x_{i-1} + y_{i-2} &&= \alpha x_{i-1} + \alpha x_{i-2} + \alpha x_{i-3} + \alpha x_{i-4} + \alpha x_{i-5} + \alpha x_{i-6} + \alpha x_{i-7} + y_{i-8} \\
y_{i-2} &= \alpha x_{i-2} + y_{i-3} &&= \alpha x_{i-2} + \alpha x_{i-3} + \alpha x_{i-4} + \alpha x_{i-5} + \alpha x_{i-6} + \alpha x_{i-7} + y_{i-8} \\
y_{i-3} &= \alpha x_{i-3} + y_{i-4} &&= \alpha x_{i-3} + \alpha x_{i-4} + \alpha x_{i-5} + \alpha x_{i-6} + \alpha x_{i-7} + y_{i-8} \\
y_{i-4} &= \alpha x_{i-4} + y_{i-5} &&= \alpha x_{i-4} + \alpha x_{i-5} + \alpha x_{i-6} + \alpha x_{i-7} + y_{i-8} \\
y_{i-5} &= \alpha x_{i-5} + y_{i-6} &&= \alpha x_{i-5} + \alpha x_{i-6} + \alpha x_{i-7} + y_{i-8} \\
y_{i-6} &= \alpha x_{i-6} + y_{i-7} &&= \alpha x_{i-6} + \alpha x_{i-7} + y_{i-8} \\
y_{i-7} &= \alpha x_{i-7} + y_{i-8}
\end{aligned} \tag{1}$$

Note that the expressions depend on y_{i-8} rather than y_{i-1}.

Also, note that each of these results consists of adding scaled inputs. Adds are an associative operator and thus allow us to use parallel prefix trees to compute the result. The use of a parallel prefix tree is required since we need all of the intermediate results and not just $y[i]$. To minimize delay, we will use Kogge-Stone which minimizes the number of logic levels required in the tree:



Problem 4: Simultaneous Multithreading (SMT) [6 pts]

The c-slowing technique is used to allow independent computations to share a single pipeline. When applied to CPU design, this is called "multi-threading" because multiple instruction streams share a single pipeline. Because dependent instructions are spread out in the pipeline, this approach has the beneficial property that with a sufficient number of instruction streams, hazards are eliminated, without the need for special bypassing hardware for data hazards and branch prediction for control hazards. Based on our 3-stage RISC-V pipelined processor, how many threads would be needed to eliminate the data and control hazards? Explain what changes would need to be made to the processor to support this style of execution?

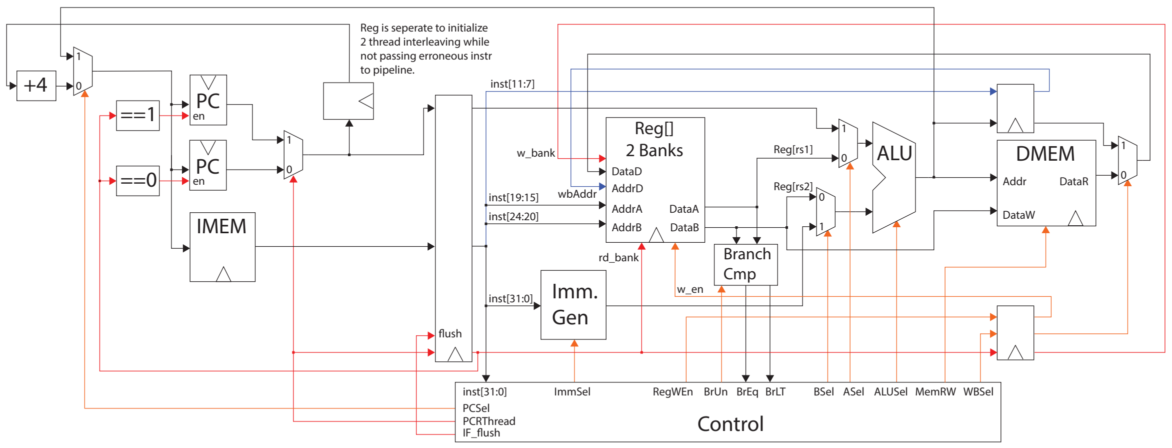
Solution:

For our 3 stage RISC-V pipelined processor, the interleaving of 2 threads is sufficient to eliminate both the control and data hazards in the design. To support multiple threads being concurrently executed on the processor, the thread from which instructions are fetched is alternated in each cycle. After an instruction has been fetched, it is allowed to progress through the pipeline as usual.

To illustrate this, let's say that we have 2 threads: A and B. In cycle 0, the instruction is fetched for thread A. In cycle 0, the instruction from thread A moves into the next pipline stage and the next instruction from thread B is fetched.

In order to accommodate these two threads in the pipeline, some modifications to the design are required. The architectural state must be replicates so that each thread is free to modify its own state without clobbering the state of the other thread. This involves duplicating the PC and RegFile. Additional pipeline registers are also required to carry information about which thread is being executed in each pipeline stage. This information is used to determine which PC or RegFile should be accessed.

An example modification to the RISC-V 3 stage pipeline for two thread interleaving is shown below. Note that there is some complexity in starting program execution that the controller must contend with. This is because the next instruction address for a thread is effectively determined in the Ex stage (although the final determination is drawn in the IF stage here). The controller must prime the correct address into the IF pipeline while preventing erroneously fetched instructions from being passed down the pipeline.



We will now detail why two thread interleaving is sufficient to eliminate the hazards in the RISC-V pipeline:

4.1 Control Hazards

Control hazards occur when the next instruction address is not known in the IF stage. This is because branch and jump instructions are not determined until the Ex stage of our design. By interleaving 2 threads, thread A is in the IF stage while thread B is in the Ex stage. At this point, the next instruction address of thread B is known and can be passed back to the IF stage. On the next cycle, the correct instruction for thread B is fetched.

Another way to think of this is that our resolution of the control hazard is to predict that a branch is not taken. If our prediction is wrong, we insert a bubble into the pipeline. The interleaving of the two threads of execution is equivalent to always inserting a bubble into the pipeline (accounting for the worst case where we incorrectly predict the branch address) and filling the bubble with an instruction from an independent thread.

4.2 Data Hazard

Data Hazards occur when information required in the Ex stage has not been written back into the register file. This could either be the result of an ALU operation or Mem load in the last instruction. These values are written back into the register file in the Mem/WB stage. Let's say we have an instruction (instr x) from thread A in the Ex stage which the following instruction (inst y) in thread A is dependent on. On the next clock cycle, inst x enters the Mem/WB stage. However, an independent instruction from thread B enters the Ex stage. On the next cycle, the write-back by inst x occurs as the dependent instruction from thread A (instr y) enters the Ex stage. Since instr x was able to write its result into the register file, inst y can properly fetch the dependent data from the register file, effectivly eliminating the data hazard.

Another way to think about this is that one method to resolve data hazards is to stall the pipeline to allow the write back of the required data to complete. By interleaving threads, we are effectively always stalling the pipeline and filling the stall with an independent instruction from another thread.

Problem 5: Carry Select Adder [5 pts]

Consider the design of a 48-bit carry select adder. What set of block sizes would lead to minimal delay? Assume that $\tau_{FA} = \tau_{MUX}$.

Solution:

From MSB to LSB, use blocks of sizes (in bits): 9, 8, 7, 6, 6, 5, 4, 3

This yields $3 + 6 = 9$ delays, plus 1 delay for the final mux out, making 10 in total. The sum is 48, as is necessary.

There *might* be better solutions, since in general this is a hard optimisation problem to solve. This solution was developed with a few pieces of intuition:

- Since the delay of a multiplexor and an adder are the same here, the delay in computing addition on 1 bit is the same as the delay for adding another block in sequence.
- Heurisitically, we want our blocks to compute their addition just in time for the signal to be available to the select mux. That signal arrives from upstream (lesser significant bits). We can afford to compute an extra bit of addition while waiting for the mux. So each subsequent block downstream should be 1 extra bit in width.
- We also figure that the least significant block should be as narrow as possible to enable more parallel additions further down the line while the muxes propagate.

- We want the sum of the bits to be 48, we want to minimise the number of blocks, and blocks should be about 1 bit different in width. One way to find such a sum is to break 48 into its factors and vary each up and down so that the sum remains the same. E.g. 4×12 yields 48 in four blocks of 12, but the delay is bounded by the sum of the two values: 12 for each block and 4 for the muxes between them.
- It looks like we want the factors of 48 with the smallest sum: 8×6. That's conveniently 4 pairs of 6. Let's smear the values in our four pairs: 6 + 6; 6 (+1) + 6 (-1); 6 (+2) + 6 (-2); 6 (+3) + 6 (-3). This preserves the total sum, but now we have blocks that are at most 1-bit apart in width: 6, 6, 7, 5, 8, 4, 9, 3.
- Arranging these in order, we arrive at our solution: 9, 8, 7, 6, 6, 5, 4, 3.
- Again intuitively, if we made individual block bigger, we'd be wasting time by performing additions when muxes are all ready; if we used any fewer number of blocks (to avoid mux delays), we'd have have a much greater delay in the biggest block. So this "feels" optimal.

Problem 6: 16 Port Adder [8 pts]

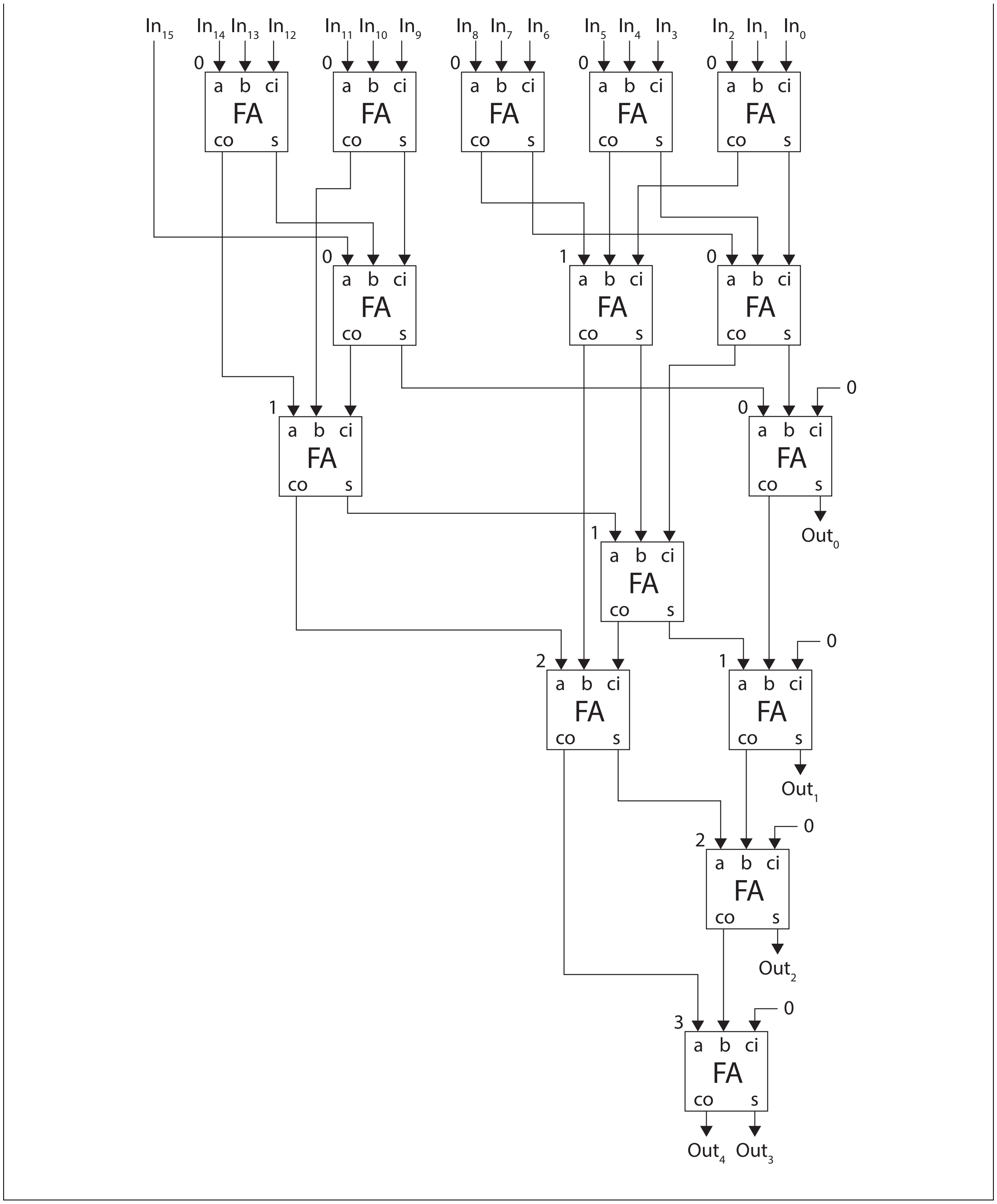
Based on full-adder cells, draw a circuit that can be used to add a set of 16 1-bit inputs and output the sum.

Minimize the amount of logic required for your implementation.

Solution:

The first thing to note about this problem is that the maximum result is 16 which requires 5 bits. If we were to construct this adder using a cascade of 2 input adders, it would be easy to follow the convention of expanding the width of the result by 1 for each adder. This would result in a width of 16 which is wider than we need.

The following implementation is inspired by the Carry Save adder. It exploits the fact that full adders have 3 inputs and 2 outputs. The first layer computes the sum of 3 bits. The sum results of these adders are then added to get the first bit of the output. The carries are then added to get the second bit of the output. The pattern continues until all bits of the output have been computed.

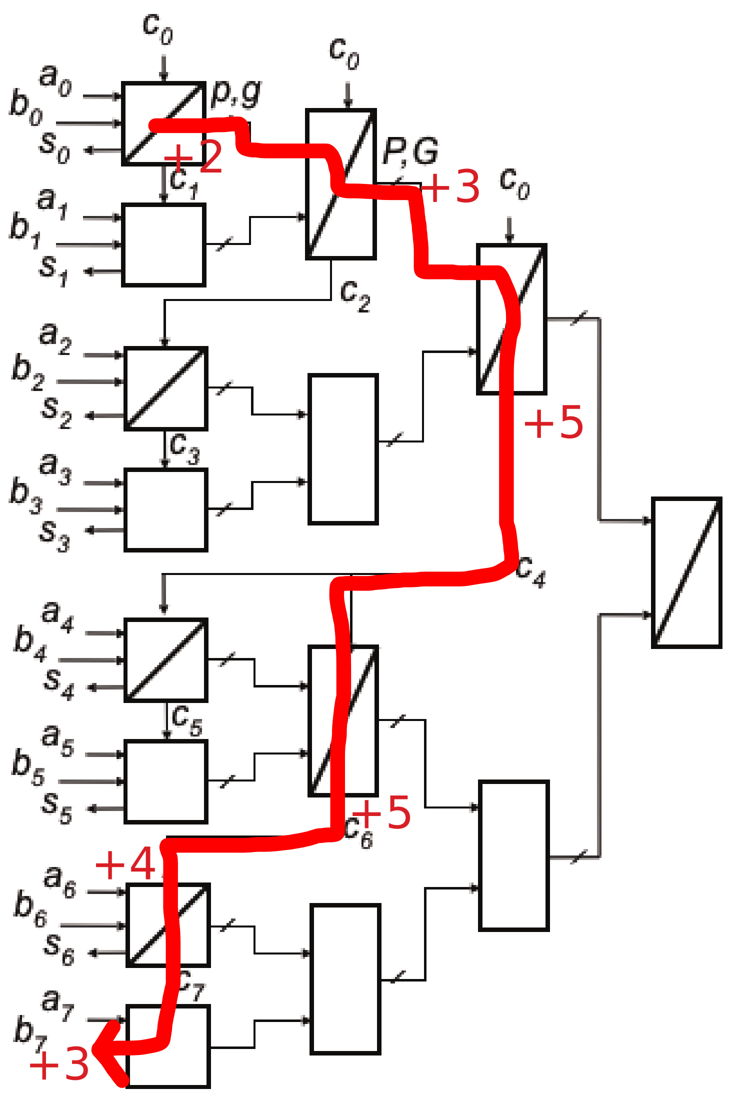

In15 In14 In13 In12 In11 In10 In9 In8 In7 In6 In5 In4 In3 In2 In1 In0
FA
a b ci
co s
0
1
2
3
Out0
Out1
Out2
Out3
Out4

Problem 7: Carry Look-ahead Adder [5 pts]

For the CLA adder shown in slide 24 of lecture 20, what is total number of gates in series on the critical path, assuming 2-input gates? (It's okay to count AND and OR each as only one gate).

Solution:

The critical path is shown below. Note that at each block, the delay depends on the particular output signal we're considering.



There are 22 gates in the shown path, which includes the final bit in the output sum, s_7.

Problem 8: Brent-Kung and Kogge-Stone Adders [8pts]

For an 8-bit version of the Brent-Kung adder architecture, and considering its implementation using 2-input gates:

(a) What is the total number of gates?

(b) How many gates are in series on the critical path?

Repeat for the Kogge-Stone adder architecture.

Solution:

Consider the architectural synopsis shown on Lecture 20, Slide 28. Each node in the tree produces a *propagate* (1 gate) and a *generate* (2 gates), so each node contributes 3 gates to the sum of gates in the design. The critical path includes every node but only depends on the *generate* signals, since they have more gates in series, so each node contributes 2 gates to the critical path. The final sum for each bit requires an additional gate. The critical path has only one final sum in it, so only incurs 1 additional gate. Thus:

Number of gates	Total	Critical Path
Brent-Kung	$11 \times 3 + 8 = 41$	$11 \times 2 + 1 = 23$
Kogge-Stone	$17 \times 3 + 8 = 59$	$17 \times 2 + 1 = 35$