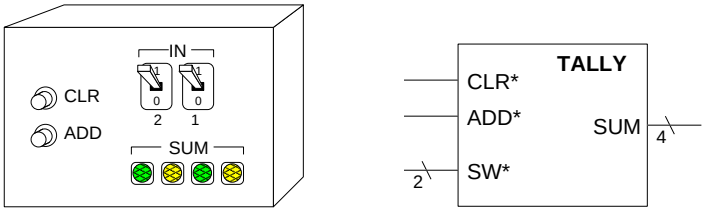
First Design Example

We are to design a device that tallies a sequence of inputs from the operator. Inputs include pushbutton switches to clear (CLR) and tally (ADD) the input from two toggle switches (SW0, SW1). For correct operation, the toggle switches must be set before the ADD button is pushed. Each time the ADD button is pushed the input is added to an accumulated SUM which is displayed as a binary number in four lights.

The picture below illustrates what the **TALLY** box might look like. All we are really concerned with, though, is the external logical view, shown to the right.



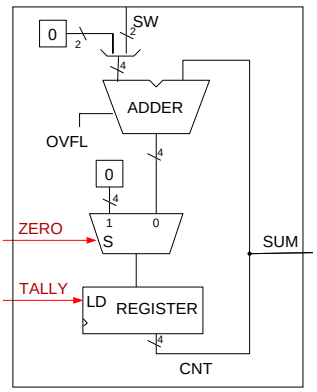
This description leaves several details unspecified. For example, it does not specify what happens if the accumulated sum exceeds four bits, or whether CLR takes precedence over ADD. Our implementation will have to resolve these issues and perhaps others (whether we recognize them or not!).

Let us reiterate some basic Design Principles:

- Separate *design* and *implementation* phases.
- In *design*
  - Make an estimate of architecture, but do not prematurely commit to representation details.
  - Develop a control algorithm for the architecture you sketched.
  - Refine the architecture in light of insights gained in control development.
  - If refinement degenerates to a design cycle, discard the design and start over.
- In *implementation*
  - Be ruthlessly systematic.
  - Maintain the over-all design structure.
  - Resist "tweaking" the design to achieve local implementation trade-offs.

Architecture

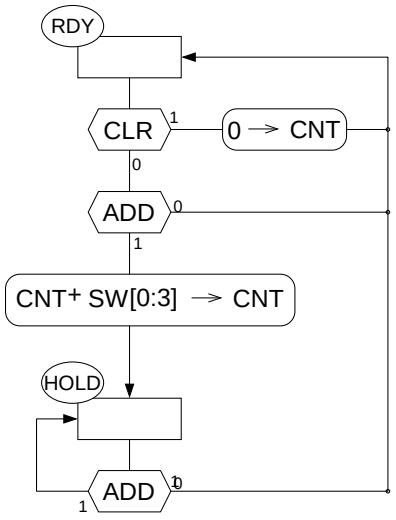
We need a register to accumulate and hold a 4-bit sum, a function to add a 2-bit input to the content of the register, and a means of clearing the accumulator. Abstractly, these needs give us the architecture shown to the right. Our controller does not need any *status* information from this architecture—its algorithmic flow is determined by the push buttons only. Its *control* signals will determine what operation, *clear*, *hold*, or *add*, is performed on the SUM.



The pushbuttons and switches are asynchronous, of course, but our architecture sketch (and later our control diagram) refer to synchronous inputs such as SW. Part of the work of the implementation is to synchronize these signals, but suppose, for the moment that external inputs have been synchronized.

Algorithm

Since the clock frequency is likely to be vastly higher than the rate at which the operator can push the buttons, we need to be sure that one button-pushing action is interpreted as a single *add* instuctions. Our ASM has two states, one ready for a button to be pushed and the other waiting for a button to be released. In the event that the CLR button is pushed, a command is issued to the architecture to clear the SUM. If the device is ready to perform an *add* an the ADD button is pushed, the switches are added to the accumulator, and control goes to a state in which it is waiting for the button to be released.



At this point, one should stop to analyze pathelogical cases arising from our algorithm, such as how it behaves when both buttons are pushed, and so forth. Tracing the paths through the decision blocks, there do not appear

to be any pathelogical behaviors. In larger designs, isolating and resolving anomalous behavior is difficult because one must reason about the implications of all possible input events in all possible orders.

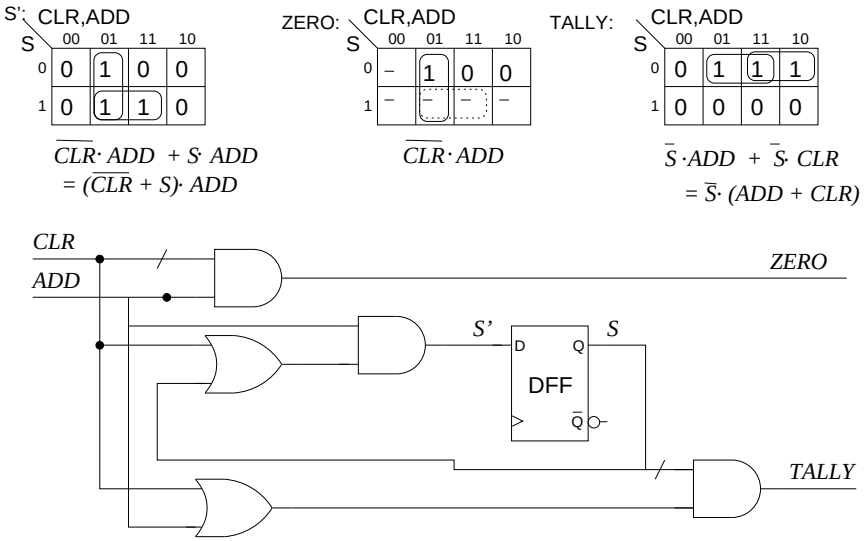
Implementation

During the implementation stage, we fix data representations and choose devices as we proceed. There are many opportunities for second-guessing our choices, as is the always the case in design. Experience helps guide our choices, but as we gain this experience, it is important to avoid getting mired by details. Even in this small example there are numerous ways to implement the behavior of the abstract architecture.

Control Implementation

The ASM has two states. One way to implement it is to have a single bit, S to distinguishe RDY ($S = 0, say$ and HLD. The truth table below develops *both* the *next-state* function, S', *and* the command issued to the architecture. Both these functions depend on S, CLR, and ADD.

	S	CLR	ADD	S'	ZERO	TALLY	*command*
	0	0	0	0 RDY	–	0	*hold*
RDY	0	0	1	1 HLD	1	1	*add*
	0	1	X	0 RDY	0	1	*clear*
HLD	1	X	0	0 RDY	–	0	*hold*
	1	X	1	1 HLD	–	0	*hold*



Architecture Implementation

Gate Level Implementation.

The adder. We could use a 4-bit combinational adder, developed earlier. The 2-bit SW input is expanded to 4-bits by adding two constant-0 signals. If saving gates were critical, the adder could be specialized in three ways:

– C_0 is always 0, so

$$\begin{aligned} S_0 &= A_0 \oplus B_0 \oplus 0 = A_0 \oplus B_0 \\ C_1 &= A_0\, B_0 + A_0\, 0 + B_0\, 0 = A_0\, B_0 \end{aligned}$$

This combination is called an *half adder*.

– The upper two bits of the SW operand are 0s, so

$$\begin{aligned} S_2 &= A_2 \oplus 0 \oplus C_2 = A_2 \oplus C_2 \\ C_3 &= A_2\, 0 + A_2\, C_2 + 0\, C_2 = A_2\, C_2 \\ S_3 &= A_3 \oplus 0 \oplus C_3 = A_3 \oplus C_3 \\ C_4 &= \textit{not used} \end{aligned}$$

The selector. In the architectural sketch, a mux is used to select the next value for the register, a *zero* if the command is to clear the

counter, and the new *sum* from the adder otherwise. The mux symbol in the schematic represents a bank of four muxes, one for each bit of the sum.

Since one of the selected inputs is 0, the selection can be specialized to

$$\mathsf{SUM}'_i = \overline{M} \cdot 0 + M \cdot S_i = M \cdot S_i$$

So selection in this case can be implemented with an *and* gate.

The accumulator. The register containing SUM will capture its input when a *load* command (LD) is asserted.

MSI Implementation. For the purpose of concreteness, refer to Table 21–1 in the textbook (p. 512), which lists some common MSI devices.

The adder. A 74LS181 4-bit ALU would also subsume the selector, since it includes an operation to generate a 0000 output.

The selectors. A 74LS157 quad-2-input multiplexor packages four multiplexors. As noted above, a 74LS00 quad-nand package would also suffice; resulting in fewer gates but no fewer chips.

The accumulator. A 74LS175 4-bit D register packages four D flipflops.

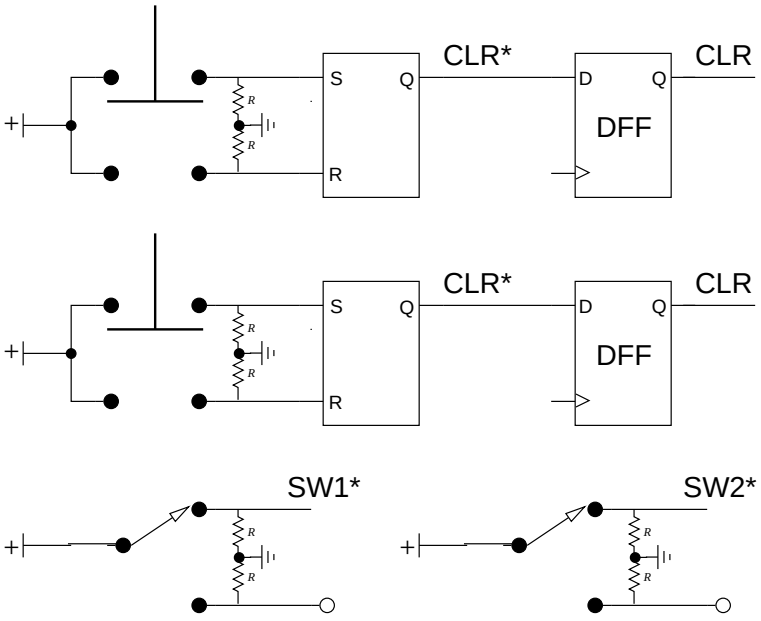
FPGA Implementation. In most modern design environments low-level optimization details are left to design automation facilities in the final stages of design synthesis. The design engineer is dealing with logical abstractions contained an a hierarchical library of design elements, including conceptual building blocks, such as *adder*, *multiplexor* and *register*.

This is a mixed blessing. Most of the time, the synthesis tools do an adequate job of minimization—usually better than a human would do manually. Occasionally, however, they don't; and when automation fails to achieve sufficient performance goals, the designer must deduce what went awry, often with very indirect knowledge of the synthesis process. The "art" of design becomes a process of discovering what modes of specification lead to good outcomes.

Refinements

In the course of this example, considerations have been raised that have been deferred.

Input Synchronization . The *CLR* and *ADD* pushbuttons should be debounced (using an RS flip-flop) and synchronized using a D flip-flop. The *SW* data switches do not need to be debounced because the protocol requires that their positions be set prior to pushing the *ADD* button.



Exercises

1. Modify the design to stop accumulating values once *SUM* exceeds the maximum value of 16. If the count overflows, *SUM* should be set to 1111 (15) and an *overflow* indicator should be lit.

2. Reduce the design to gate level and optimize the selectors.

3. Build an addition circuit that incorporates the *clear* mode and *overflow* condition.

