ECE120: Introduction to Computer Engineering

Notes Set 3.7 From FSM to Computer

The FSM designs we have explored so far have started with a human-based design process in which someone writes down the desired behavior in terms of states, inputs, outputs, and transitions. Such an approach makes it easier to build a digital FSM, since the abstraction used corresponds almost directly to the implementation.

As an alternative, one can start by mapping the desired task into a high-level programming language, then using components such as registers, counters, and memories to implement the variables needed. In this approach, the control structure of the code maps into a high-level FSM design. Of course, in order to implement our FSM with digital logic, we eventually still need to map down to bits and gates.

In this set of notes, we show how one can transform a piece of code written in a high-level language into an FSM. This process is meant to help you understand how we can design an FSM that executes simple pieces of a flow chart such as assignments, `if` statements, and loops. Later, we generalize this concept and build an FSM that allows the pieces to be executed to be specified after the FSM is built—in other words, the FSM executes a program specified by bits stored in memory. This more general model, as you might have already guessed, is a computer.

3.7.1 Specifying the Problem

Let's begin by specifying the problem that we want to solve. Say that we want to find the minimum value in a set of 10 integers. Using the C programming language, we can write the following fragment of code:

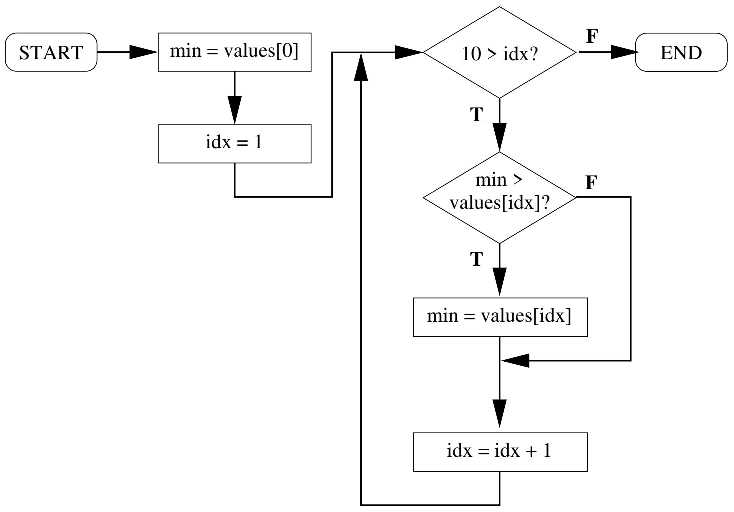
```
int values[10];    /* 10 integers--filled in by other code */
int idx;
int min

min = values[0];
for (idx = 1; 10 > idx; idx = idx + 1) {
    if (min > values[idx]) {
        min = values[idx];
    }
}
/* The minimum value from array is now in min.  */
```

The code uses array notation, which we have not used previously in our class, so let's first discuss the meaning of the code.

The code uses three variables. The variable `values` represents the 10 values in our set. The suffix "[10]" after the variable name tells the compiler that we want an array of 10 integers (`int`) indexed from 0 to 9. These integers can be treated as 10 separate variables, but can be accessed using the single name "`values`" along with an index (again, from 0 to 9 in this case). The variable `idx` holds a loop index that we use to examine each of the values one by one in order to find the minimum value in the set. Finally, the variable `min` holds the smallest known value as the program examines each of the values in the set.

The program body consists of two statements. We assume that some other piece of code—one not shown here—has initialized the 10 values in our set before the code above executes. The first statement initializes the minimum known value (`min`) to the value stored at index 0 in the array (`values[0]`). The second statement is a loop in which the variable `index` takes on values from 1 to 9. For each value, an `if` statement compares the current known minimum with the value stored in the array at index given by the `idx` variable. If the stored value is smaller, the current known value (again, `min`) is updated to reflect the program's having found a smaller value. When the loop finishes all nine iterations, the variable `min` holds the smallest value among the set of 10 integers stored in the `values` array.



As a first step towards designing an FSM to implement the code, we transform the code into a flow chart, as shown to the right. The program again begins with initialization, which appears in the second column of the flow chart. The loop in the program translates to the third column of the flow chart, and the `if` statement to the middle comparison and update of `min`.

Our goal is now to design an FSM to implement the flow chart. In order to do so, we want to leverage the same kind of abstraction that we used earlier, when extending our keyless entry system with a timer. Although the timer's value was technically also part of the FSM's state, we treated it as data and integrated it into our next-state decisions in only a couple of cases.

For our minimum value problem, we have two sources of data. First, an external program supplies data in the form of a set of 10 integers. If we assume 32-bit integers, these data technically form 320 input bits! Second, as with the keyless entry system timer, we have data used internally by our FSM, such as the loop index and the current minimum value. These are technically state bits. For both types of data, we treat them abstractly as values rather than thinking of them individually as bits, allowing us to develop our FSM at a high-level and then to implement it using the components that we have developed earlier in our course.

3.7.2 Choosing Components and Identifying States

Now we are ready to design an FSM that implements the flow chart. What components do we need, other than our state logic? We use registers and counters to implement the variables `idx` and `min` in the program. For the array `values`, we use a 16×32-bit memory.[14] We need a comparator to implement the test for the `if` statement. We choose to use a serial comparator, which allows us to illustrate again how one logical high-level state can be subdivided into many actual states. To operate the serial comparator, we make use of two shift registers that present the comparator with one bit per cycle on each input, and a counter to keep track of the comparator's progress.

How do we identify high-level states from our flow chart? Although the flow chart attempts to break down the program into 'simple' steps, one step of a flow chart may sometimes require more than one state in an FSM. Similarly, one FSM state may be able to implement several steps in a flow chart, if those steps can be performed simultaneously. Our design illustrates both possibilities.

How we map flow chart elements into FSM states also depends to some degree on what components we use, which is why we began with some discussion of components. In practice, one can go back and forth between the two, adjusting components to better match the high-level states, and adjusting states to better match the desired components.

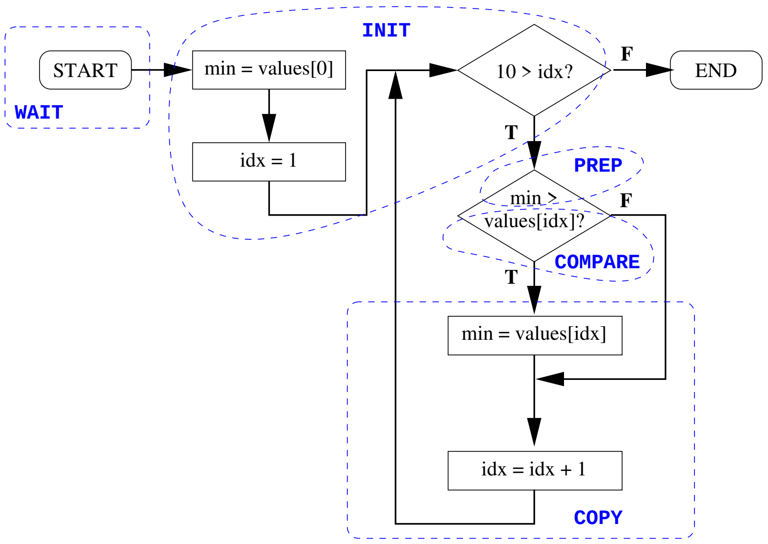
Finally, note that we are only concerned with high-level states, so we do not need to provide details (yet) down to the level of individual clock cycles, but we do want to define high-level states that can be implemented in a fixed number of cycles, or at least a controllable number of cycles. If we cannot specify clearly when transitions occur from an FSM state, we may not be able to implement the state.

[14]We technically only need a 10×32-bit memory, but we round up the size of the address space to reflect more realistic memory designs; one can always optimize later.

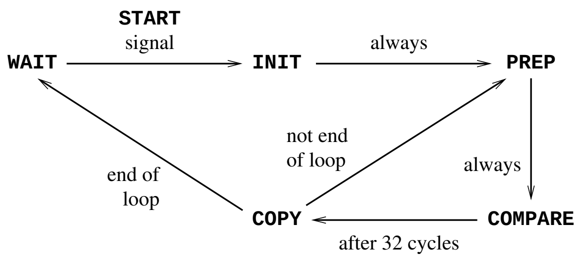
Now let's go through the flow chart and identify states. Initialization of `min` and `idx` need not occur serially, and the result of the first comparison between `idx` and the constant 10 is known in advance, so we can merge all three operations into a single state, which we call `INIT`.

We can also merge the updates of `min` and `idx` into a second FSM state, which we call `COPY`. However, the update to `min` occurs only when the comparison (`min > value[idx]`) is true. We can use logic to predicate execution of the update. In other words, we can use the output of the comparator, which is available after the comparator has finished comparing the two values (in a high-level FSM state that we have yet to define), to determine whether or not the register holding `min` loads a new value in the `COPY` state.

Our model of use for this FSM involves external logic filling the memory (the array of integer values), executing the FSM "code," and then checking the answer. To support this use model, we create a FSM state called `WAIT` for cycles in which the FSM has no work to do. Later, we also make use of an external input signal `START` to start the FSM execution. The `WAIT` state logically corresponds to the "START" bubble in the flow chart.



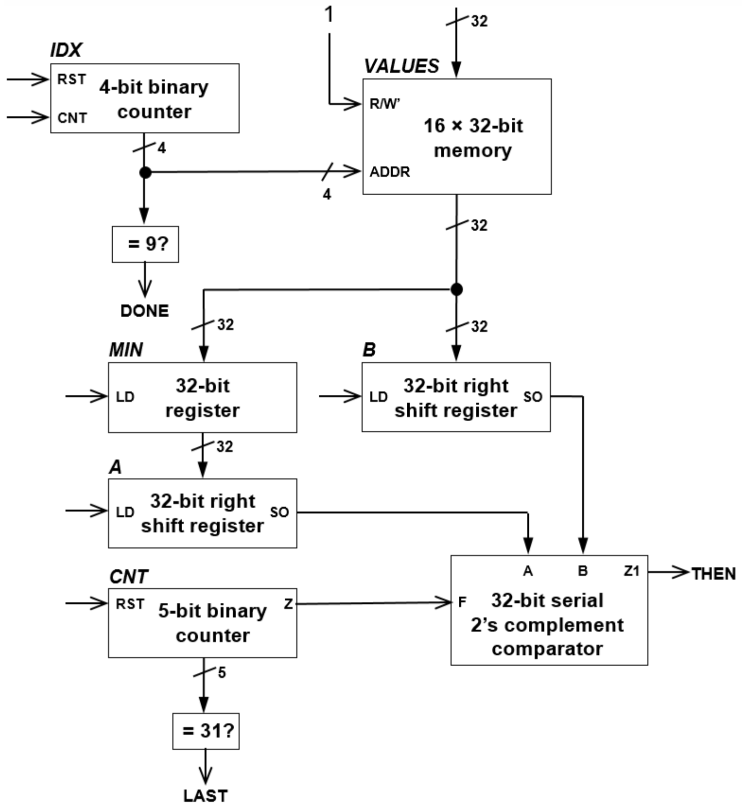
Only the test for the `if` statement remains. Using a serial comparator to compare two 32-bit values requires 32 cycles. However, we need an additional cycle to move values into our shift registers so that the comparator can see the first bit. Thus our single comparison operation breaks into two high-level states. In the first state, which we call `PREP`, we copy `min` to one of the shift registers, copy `values[idx]` to the other shift register, and reset the counter that measures the cycles needed for our serial comparator. We then move to a second high-level state, which we call `COMPARE`, in which we feed one bit per cycle from each shift register to the serial comparator. The `COMPARE` state executes for 32 cycles, after which the comparator produces the one-bit answer that we need, and we can move to the `COPY` state. The association between the flow chart and the high-level FSM states is illustrated in the figure shown to the right above.



We can now also draw an abstract state diagram for our FSM, as shown to the right. The FSM begins in the `WAIT` state. After external logic fills the `values` array, it signals the FSM to begin by raising the `START` signal. The FSM transitions into the `INIT` state, and in the next cycle into the `PREP` state. From `PREP`, the FSM always moves to `COMPARE`, where it remains for 32 cycles while the serial comparator executes a comparison. After `COMPARE`, the FSM moves to the `COPY` state, where it remains for one cycle. The transition from `COPY` depends on how many loop iterations have executed. If more loop iterations remain, the FSM moves to `PREP` to execute the next iteration. If the loop is done, the FSM returns to `WAIT` to allow external logic to read the result of the computation.

3.7.3 Laying Out Components

Our high-level FSM design tells us what our components need to be able to do in any given cycle. For example, when we load new values into the shift registers that provide bits to the serial comparator, we always copy `min` into one shift register and `values[idx]` into the second. Using this information, we can put together our components and simplify our design by fixing the way in which bits flow between them.



The figure at the right shows how we can organize our components. Again, in practice, one goes back and forth thinking about states, components, and flow from state to state. In these notes, we present only a completed design.

Let's take a detailed look at each of the components. At the upper left of the figure is a 4-bit binary counter called `IDX` to hold the `idx` variable. The counter can be reset to 0 using the `RST` input. Otherwise, the `CNT` input controls whether or not the counter increments its value. With this counter design, we can force `idx` to 0 in the `WAIT` state and then count upwards in the `INIT` and `COPY` states.

A memory labeled `VALUES` to hold the array `values` appears in the upper right of the figure. The read/write control for the memory is hardwired to 1 (read) in the figure, and the data input lines are unattached. To integrate with other logic that can operate our FSM, we need to add more control logic to allow writing into the memory and to attach the data inputs to something that provides the data bits. The address input of the memory comes always from the `IDX` counter value; in other words, whenever we access this memory by making use of the data output lines, we read `values[idx]`.

In the middle left of the figure is a 32-bit register for the `min` variable. It has a control input `LD` that determines whether or not it loads a new value at the end of the clock cycle. If a new value is loaded, the new value always corresponds to the output of the `VALUES` memory, `values[idx]`. Recall that `min` always changes in the `INIT` state, and may change in the `COPY` state. But the new value stored in `min` is always `values[idx]`. Note also that when the FSM completes its task, the result of the computation is left in the `MIN` register for external logic to read (connections for this purpose are not shown in the figure).

Continuing downward in the figure, we see two right shift registers labeled `A` and `B`. Each has a control input `LD` that enables a parallel load. Register `A` loads from register `MIN`, and register `B` loads from the memory data output (`values[idx]`). These loads are needed in the `PREP` state of our FSM. When `LD` is low, the shift registers simply shifts to the right. The serial output `SO` makes the least significant bit of each shift register available. Shifting is necessary to feed the serial comparator in the `COMPARE` state.

Below register `A` is a 5-bit binary counter called `CNT`. The counter is used to control the serial comparator in the `COMPARE` state. A reset input `RST` allows it to be forced to 0 in the `PREP` state. When the counter value is exactly zero, the output `Z` is high.

The last major component is the serial comparator, which is based on the design developed in Notes Set 3.1. The two bits to be compared in a cycle come from shift registers `A` and `B`. The first bit indicator comes from the zero indicator of counter `CNT`. The comparator actually produces two outputs (`Z1` and `Z0`), but the meaning of the `Z1` output by itself is `A > B`. In the diagram, this signal has been labeled `THEN`.

There are two additional elements in the figure that we have yet to discuss. Each simply compares the value in a register with a fixed constant and produces a 1-bit signal. When the FSM finishes an iteration of the loop in the `COPY` state, it must check the loop condition (`10 > idx`) and move either to the `PREP` state or, when the loop finishes, to the `WAIT` state to let the external logic read the answer from the `MIN` register. The loop is done when the current iteration count is nine, so we compare `IDX` with nine to produce the `DONE` signal. The other constant comparison is between the counter `CNT` and the value 31 to produce the `LAST` signal, which indicates that the serial comparator is on its last cycle of comparison. In the cycle after `LAST` is high, the `THEN` output of the comparator indicates whether or not `A > B`.

3.7.4 Control and Data

One can think of the components and the interconnections between them as enabling the movement of data between registers, while the high-level FSM controls which data move from register to register in each cycle. With this model in mind, we call the components and interconnections for our design the **datapath**—a term that we will see again when we examine the parts of a computer in the coming weeks. The datapath requires several inputs to control the operation of the components—these we can treat as outputs of the FSM. These signals allow the FSM to control the motion of data in the datapath, so we call them **control signals**. Similarly, the datapath produces several outputs that we can treat as inputs to the FSM. The tables below summarize the control signals (left) and outputs (right) from the datapath for our FSM.

datapath input	meaning
`IDX.RST`	reset `IDX` counter to 0
`IDX.CNT`	increment `IDX` counter
`MIN.LD`	load new value into `MIN` register
`A.LD`	load new value into shift register `A`
`B.LD`	load new value into shift register `B`
`CNT.RST`	reset `CNT` counter

datapath output	meaning	based on
`DONE`	last loop iteration finished	`IDX = 9`
`LAST`	serial comparator executing last cycle	`CNT = 31`
`THEN`	`if` statement condition true	`A > B`

Using the datapath controls signals and outputs, we can now write a more formal state transition table for the FSM, as shown below. The "actions" column of the table lists the changes to register and counter values that are made in each of the FSM states. The notation used to represent the actions is called **register transfer language** (**RTL**). The meaning of an individual action is similar to the meaning of the corresponding statement from our C code or from the flow chart. For example, in the `WAIT` state, "`IDX` ← `0`" means the same thing as "`idx = 0;`". In particular, both mean that the value currently stored in the `IDX` counter is overwritten with the number 0 (all 0 bits).

state	actions (simultaneous)	condition	next state
`WAIT`	`IDX` ← 0 (to read `VALUES[0]` in `INIT`)	`START` $\overline{\texttt{START}}$	`INIT` `WAIT`
`INIT`	`MIN` ← `VALUES[IDX]` (`IDX` is 0 in this state) `IDX` ← `IDX` + 1	(always)	`PREP`

`PREP`	`A` ← `MIN`
	`B` ← `VALUES[IDX]`
	`CNT` ← 0
		(always)	`COMPARE`

`COMPARE`	run serial comparator	`LAST` $\overline{\texttt{LAST}}$	`COPY` `COMPARE`
`COPY`	`THEN`: `MIN` ← `VALUES[IDX]` `IDX` ← `IDX` + 1	`DONE` $\overline{\texttt{DONE}}$	`WAIT` `PREP`

The meaning of RTL is slightly different from the usual interpretation of high-level programming languages, however, in terms of when the actions happen. A list of C statements is generally executed one at a time. In contrast, the entire list of RTL actions for an FSM state is executed simultaneously, at the end of the clock cycle. As you know, an FSM moves from its current state into a new state at the end of every clock cycle, so actions during different cycles usually are associated with different states. We can, however, change the value in more than one register at the end of the same clock cycle, so we can execute more than one RTL action in the same state, so long as the actions do not exceed the capabilities of our datapath (the components must be able to support the simultaneous execution of the actions). Some care must be taken with states that execute for more than one cycle to ensure that repeating the RTL actions is appropriate. In our design, only the `WAIT` and `COMPARE` states execute for more than one cycle. The `WAIT` state resets the `IDX` counter repeatedly, which causes no problems. The `COMPARE` statement has no RTL actions—all of the shifting, comparison, and counting activity needed to do its work occurs within the datapath itself.

One additional piece of RTL syntax needs explanation. In the `COPY` state, the first action begins with "`THEN`:," which means that the prefixed RTL action occurs only when the `THEN` signal is high. Recall that the `THEN` signal indicates that the comparator has found `A` > `B`, so the equivalent C code is "`if (A > B) {min = values[idx]}`".

3.7.5 State Representation and Logic Expressions

Let's think about the representation for the FSM states. The FSM has five states, so we could use as few as three flip-flops. Instead, we choose to use a **one-hot encoding**, in which any valid bit pattern has exactly one 1 bit. In other words, we use five flip-flops instead of three, and our states are represented with the bit patterns 10000, 01000, 00100, 00010, and 00001.

The table below shows the mapping from each high-level state to both the five-bit encoding for the state as well as the six control signals needed for the datapath. For each state, the values of the control signals can be found by examining the actions necessary in that state.

state	$S_4S_3S_2S_1S_0$	IDX.RST	IDX.CNT	MIN.LD	A.LD	B.LD	CNT.RST
WAIT	1 0 0 0 0	1	0	0	0	0	0
INIT	0 1 0 0 0	0	1	1	0	0	0
PREP	0 0 1 0 0	0	0	0	1	1	1
COMPARE	0 0 0 1 0	0	0	0	0	0	0
COPY	0 0 0 0 1	0	1	THEN	0	0	0

The `WAIT` state needs to set `IDX` to 0 but need not affect other register or counter values, so `WAIT` produces a 1 only for `IDX.RST`. The `INIT` state needs to load `values[0]` into the `MIN` register while simultaneously incrementing the `IDX` counter (from 0 to 1), so `INIT` produces 1s for `IDX.CNT` and `MIN.LD`. The `PREP` state loads both shift registers and resets the counter `CNT` by producing 1s for `A.LD`, `B.LD`, and `CNT.RST`. The `COMPARE` state does not change any register values, so it produces all 0s. Finally, the `COPY` state increments the `IDX` counter while simultaneously loading a new value into the `MIN` register. The `COPY` state produces 1 for `IDX.CNT`, but must use the signal `THEN` coming from the datapath to decide whether or not `MIN` is loaded.

The advantage of a one-hot encoding becomes obvious when we write equations for the six control signals and the next-state logic, as shown to the right. Implementing the logic to complete our design now requires only a handful of small logic gates.

$$\begin{aligned}
\texttt{IDX.RST} &= S_4 & S_4^+ &= S_4 \cdot \overline{\texttt{START}} + S_0 \cdot \texttt{DONE}\\
\texttt{IDX.CNT} &= S_3 + S_0 & S_3^+ &= S_4 \cdot \texttt{START}\\
\texttt{MIN.LD} &= S_3 + S_0 \cdot \texttt{THEN} & S_2^+ &= S_3 + S_0 \cdot \overline{\texttt{DONE}}\\
\texttt{A.LD} &= S_2 & S_1^+ &= S_2 + S_1 \cdot \overline{\texttt{LAST}}\\
\texttt{B.LD} &= S_2 & S_0^+ &= S_1 \cdot \texttt{LAST}\\
\texttt{CNT.RST} &= S_2
\end{aligned}$$

Notice that the terms in each control signal can be read directly from the rows of the state table and OR'd together. The terms in each of the next-state equations represent the incoming arcs for the corresponding state. For example, the `WAIT` state has one self-loop (the first term) and a transition arc coming from the `COPY` state when the loop is done. These expressions complete our design.