We Can Perform Any Computation with an FSM

Let's build an FSM to implement a piece of C code.

We'll use components:
◦ to store the variables, and
◦ to execute the statements.

The FSM states will use the components:
◦ the FSM’s outputs
◦ are control signals for the components.

Find the Minimum Value Among Ten Integers

```
int values[10]; int idx; int min = values[0];
for (idx = 1; 10 > idx;
   idx = idx + 1) {
   if (min > values[idx]) {
      min = values[idx];
   }
}
```

This Declaration Creates an Array of Ten Integers

```
int values[10];
```

The variable declaration above creates ten 32-bit 2’s complement numbers (ten ints).
◦ Such a group is called an array,
◦ and the declaration names this particular group “values”.
◦ Individual ints are then called values[0] through values[9].

An array is the software analogue of a memory.
The first array element is used here.

Which element is accessed here depends on the value of variable `idx`.

The variable `min` is initialized to the value of the first of the ten integers.

idx ranges from 1 to 9 in the for loop.

Each integer is compared with `min` and may replace it.

When loop ends, `min` holds the minimum integer.
Flow Chart with Colors by Statement

START

idx = 1

10 > idx?

FALSE

DONE

TRUE

min =

values[0]

min >

values[idx]?

TRUE

FALSE

idx =

idx + 1

min =

values[idx]

What Components Do We Need?

Now, let’s think about how to turn the code/flow chart into an FSM.

What components should we use...
- ... for the array? a memory
- ... for other variables? registers/counters
- ... for the if statement (min > values[idx])? a comparator

Let’s use a serial comparator (to again illustrate hierarchy in an FSM).

So we also need shift registers to feed it bits, and a counter to keep track of its progress.

FSM States Must Execute in a Fixed Number of Cycles

We have to implement each high-level FSM state in a fixed number of cycles (or at least a controllable number of cycles).

Simple components imply more cycles (slower, but smaller).

Complex components reduce the number of states needed (larger, but may be faster).

For example, if we design a 10-operand comparator, our task is fairly simple!

Choice of Components Affects the FSM Design

How we select components affects
- how we choose FSM states, and
- how the FSM moves between those states.

That’s why we started by thinking about components.

In a real design process, one goes back and forth, tuning components to FSM, and tuning FSM to components.
We Can Sometimes Merge Several Boxes into One State

So how do we pick states?
Break the flow chart into pieces.
Not every flow chart box becomes a state.
For example, in our flow chart, we can
- Initialize \( \text{min} \)
- Initialize \( \text{idx} \), and
- Perform the first comparison \((10 > 1)\)
all in one cycle!

---

The First FSM State is INIT

![Flowchart](image)

We Can Predicate Execution with Logic

**Predication** means that something happens only under certain conditions.
For example: If you give me an apple, I will give you a peach.
When our comparator is done, we can
- use the comparator output to determine whether \( \text{min} \) copies \( \text{values[idx]} \), and
- increment \( \text{idx} \)
in the same cycle.

---

The Second FSM State May Copy a New Min Value

![Flowchart](image)
Sometimes the FSM Just Needs to Wait

**How can we use our FSM?**
Other logic must control the FSM.
Using the FSM works as follows:
1. Fill the memory with ten integers.
2. Execute the FSM “code.”
3. Read out the answer.

Let’s
- create a state for the FSM
to occupy during steps 1 and 3, and
- create a **START** input to begin Step 2.

The Third FSM State is for Waiting

Some Flow Chart Boxes May Require Multiple States

Preparing for the Serial Comparison Requires a State
Comparison Becomes Two High-Level States

Redraw the Abstract State Transition Diagram

IDX is a Binary Counter with CNT and RST Inputs

The Memory VALUES Can Use IDX as ADDR Input
MIN Only Needs to Load ARRAY[IDX]

`min` is a 32-bit 2’s complement value used to store the current minimum. Let’s use a 32-bit register, MIN. Load input LD controls changes to MIN. In COPY, we load VALUES[IDX] into MIN. In INIT, we load VALUES[0] into MIN. But in INIT, IDX is 0! So we connect VALUES’ data output to MIN’s data input.

Shift Registers A and B Need Parallel Load Control LD

What about the shift registers A and B? In PREP, we set A to MIN and B to VALUES[IDX]. We need LD inputs on both A and B to enable parallel load. But the parallel load inputs can be wired directly from MIN to A, and from VALUES’ data output to B.

Use a Binary Counter to Control the Comparator

Finally, we need a counter to drive the serial comparator for 32 cycles. Let’s use a 5-bit binary counter, CNT. To reset the counter, use a reset input, RST. Comparator has an F / “first bit” input. CNT should generate a zero output Z.

The Datapath Consists of the Interconnected Components

Let’s take a look at the components. This figure shows the datapath for our FSM.
The FSM Delivers Control Signals to the Datapath

How does the datapath relate to the FSM?
Not all signals into the datapath are fixed.
Remaining input signals to the components in the datapath are called **control signals**.
Control signals are outputs of the FSM.
Using these signals, **each state of the FSM causes the elements of the datapath to perform actions** associated with the state.

Our Datapath Has Six Control Signals

IDX.RST
IDX.CNT
MIN.LD
A.LD
B.LD
CNT.RST

FSM State Transitions Use Datapath Outputs

The datapath also **produces output signals that affect FSM state transitions**.
Our datapath has three such signals:
DONE the last loop iteration has finished
LAST raised in the last cycle of serial comparison
THEN a new minimum value has been found \((A > B)\)
These signals are **inputs to the FSM**.

Our Datapath Produces Three Outputs for the FSM

For our FSM, the datapath outputs are produced using simple logic.
The **THEN** output depends on the representation used by the comparator; \(Z1\) means \(A > B\).
RTL Describes How Bits Move from Register to Register

<table>
<thead>
<tr>
<th>state</th>
<th>actions (simultaneous)</th>
<th>condition</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let’s make an abstract next state table.
We’ll use register transfer language (RTL) notation to describe the state’s actions on the datapath.
RTL describes how bits move from register to register.
Start with the WAIT state.

What Happens in the WAIT State?

WAIT stalls the FSM while the controlling logic fills the memory. IDX must be 0 in INIT, so set IDX to 0 in WAIT. The FSM stays in WAIT until it sees the START signal.

What Happens in the INIT State?

INIT performs two actions:
- copy VALUES[0] to MIN,
- and set IDX to 1 (actually by setting IDX to IDX + 1).
The FSM always moves from INIT to PREP.

Write the Information for WAIT

<table>
<thead>
<tr>
<th>state</th>
<th>actions (simultaneous)</th>
<th>condition</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT</td>
<td>IDX ← 0</td>
<td>START</td>
<td>INIT</td>
</tr>
</tbody>
</table>

The WAIT state sets IDX to 0.
In RTL, we write “IDX ← 0” to indicate that the register IDX is filled with the value 0 (all 0 bits).
What about the next state(s)?
On START, move to INIT.
Otherwise, stay in WAIT.
Write the Information for INIT

<table>
<thead>
<tr>
<th>state</th>
<th>actions (simultaneous)</th>
<th>condition</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT</td>
<td>IDX ← 0</td>
<td>START</td>
<td>INIT</td>
</tr>
<tr>
<td>INIT</td>
<td>MIN ← VALUES[IDX],</td>
<td>(always)</td>
<td>PREP</td>
</tr>
<tr>
<td></td>
<td>IDX ← IDX + 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IMPORTANT: The order of actions here does not matter! INIT does two things.

Unlike languages like C, RTL actions for a state occur in parallel, in the same cycle.

After INIT, the FSM moves to PREP.

Write the Information for PREP

<table>
<thead>
<tr>
<th>state</th>
<th>actions (simultaneous)</th>
<th>condition</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT</td>
<td>IDX ← 0</td>
<td>START</td>
<td>INIT</td>
</tr>
<tr>
<td>INIT</td>
<td>MIN ← VALUES[IDX],</td>
<td>(always)</td>
<td>PREP</td>
</tr>
<tr>
<td></td>
<td>IDX ← IDX + 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PREP</td>
<td>A ← MIN, B ← VALUES[IDX], CNT ← 0</td>
<td>(always)</td>
<td>COMPARE</td>
</tr>
</tbody>
</table>

Again, RTL actions occur in parallel, all in one cycle.

What Happens in the PREP State?

PREP performs three actions:
- copy MIN to A,
- copy VALUES[IDX] to B,
- and reset CNT to 0.

The FSM always moves from PREP to COMPARE.

What Happens in the COMPARE State?

COMPARE allows the serial comparator to do its work, requiring no direct actions on the datapath.

The FSM stays in COMPARE until it sees the LAST signal (a datapath output).
Write the Information for COMPARE

<table>
<thead>
<tr>
<th>state</th>
<th>actions (simultaneous)</th>
<th>condition</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT</td>
<td>IDX ← 0</td>
<td>START</td>
<td>INIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>START'</td>
<td>WAIT</td>
</tr>
<tr>
<td>INIT</td>
<td>MIN ← VALUES[IDX]</td>
<td>(always)</td>
<td>PREP</td>
</tr>
<tr>
<td></td>
<td>IDX ← IDX + 1</td>
<td></td>
<td>COMPARE</td>
</tr>
<tr>
<td>PREP</td>
<td>A ← MIN, B ← VALUES[IDX], CNT ← 0</td>
<td>(always)</td>
<td>COMPARE</td>
</tr>
<tr>
<td>COMPARE</td>
<td>run serial comparator</td>
<td>LAST</td>
<td>COPY</td>
</tr>
<tr>
<td>COPY</td>
<td>THEN: MIN ← VALUES[IDX], IDX ← IDX + 1</td>
<td>DONE</td>
<td>WAIT</td>
</tr>
</tbody>
</table>

Write the Information for COPY

<table>
<thead>
<tr>
<th>state</th>
<th>actions (simultaneous)</th>
<th>condition</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT</td>
<td>IDX ← 0</td>
<td>START</td>
<td>INIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>START'</td>
<td>WAIT</td>
</tr>
<tr>
<td>INIT</td>
<td>MIN ← VALUES[IDX]</td>
<td>(always)</td>
<td>PREP</td>
</tr>
<tr>
<td></td>
<td>IDX ← IDX + 1</td>
<td></td>
<td>COMPARE</td>
</tr>
<tr>
<td>PREP</td>
<td>A ← MIN, B ← VALUES[IDX], CNT ← 0</td>
<td>(always)</td>
<td>COMPARE</td>
</tr>
<tr>
<td>COMPARE</td>
<td>run serial comparator</td>
<td>LAST</td>
<td>COPY</td>
</tr>
<tr>
<td>COPY</td>
<td>THEN: MIN ← VALUES[IDX], IDX ← IDX + 1</td>
<td>DONE</td>
<td>WAIT</td>
</tr>
</tbody>
</table>

What Happens in the COPY State?

The FSM moves to PREP or WAIT based on datapath output DONE.
COPY increments IDX. It also copies VALUES[IDX] into MIN iff the THEN signal is 1.

Use a One-Hot Encoding to Represent States

It's time for...
bits!
What representation should we use?
How many bits do we need (5 states)?
Only 3 bits?
Let's use 5.
Let's use a one-hot encoding; each state has exactly one 1 bit.
You'll see why soon.
Fill in the Table of FSM Outputs Based on RTL

**WAIT** state: $IDX \leftarrow 0$.
What are the control signals?

<table>
<thead>
<tr>
<th>state</th>
<th>$S_4S_3S_2S_1$</th>
<th>IDX</th>
<th>RST</th>
<th>IDX</th>
<th>CNT</th>
<th>MIN</th>
<th>LD</th>
<th>A</th>
<th>LD</th>
<th>B</th>
<th>LD</th>
<th>CNT</th>
<th>RST</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT</td>
<td>10000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INIT</td>
<td>01000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PREP</td>
<td>00100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPARE</td>
<td>00010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COPY</td>
<td>00001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Just set the other register inputs to 0 instead of don’t care.

**INIT** state:
$MIN \leftarrow \text{VALUES}[IDX]$, $IDX \leftarrow IDX + 1$.

<table>
<thead>
<tr>
<th>state</th>
<th>$S_4S_3S_2S_1$</th>
<th>IDX</th>
<th>RST</th>
<th>IDX</th>
<th>CNT</th>
<th>MIN</th>
<th>LD</th>
<th>A</th>
<th>LD</th>
<th>B</th>
<th>LD</th>
<th>CNT</th>
<th>RST</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT</td>
<td>10000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INIT</td>
<td>01000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PREP</td>
<td>00100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPARE</td>
<td>00010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COPY</td>
<td>00001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PREP** state:
$A \leftarrow \text{MIN}$, $B \leftarrow \text{VALUES}[IDX]$, $CNT \leftarrow 0$.

<table>
<thead>
<tr>
<th>state</th>
<th>$S_4S_3S_2S_1$</th>
<th>IDX</th>
<th>RST</th>
<th>IDX</th>
<th>CNT</th>
<th>MIN</th>
<th>LD</th>
<th>A</th>
<th>LD</th>
<th>B</th>
<th>LD</th>
<th>CNT</th>
<th>RST</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT</td>
<td>10000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INIT</td>
<td>01000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PREP</td>
<td>00100</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPARE</td>
<td>00010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COPY</td>
<td>00001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**COMPARE** state: no RTL.

<table>
<thead>
<tr>
<th>state</th>
<th>$S_4S_3S_2S_1$</th>
<th>IDX</th>
<th>RST</th>
<th>IDX</th>
<th>CNT</th>
<th>MIN</th>
<th>LD</th>
<th>A</th>
<th>LD</th>
<th>B</th>
<th>LD</th>
<th>CNT</th>
<th>RST</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT</td>
<td>10000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INIT</td>
<td>01000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PREP</td>
<td>00100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPARE</td>
<td>00010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COPY</td>
<td>00001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fill in the Table of FSM Outputs Based on RTL

COPY state: IDX ← IDX + 1, THEN: MIN ← VALUES[IDX].

<table>
<thead>
<tr>
<th>state</th>
<th>S₀S₁S₂S₃S₄</th>
<th>IDX.RST</th>
<th>IDX.CNT</th>
<th>MIN.LD</th>
<th>A.LD</th>
<th>B.LD</th>
<th>CNT.RST</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT</td>
<td>10000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>INIT</td>
<td>01000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>PREP</td>
<td>00100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>COMPARE</td>
<td>00010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>COPY</td>
<td>00001</td>
<td>0</td>
<td>1</td>
<td>THEN</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

We Need Expressions for the Control Signals

Now we can see the value of our one-hot state encoding. Express IDX.RST.

Quickly now!

<table>
<thead>
<tr>
<th>state</th>
<th>S₀S₁S₂S₃S₄</th>
<th>IDX.RST</th>
<th>IDX.CNT</th>
<th>MIN.LD</th>
<th>A.LD</th>
<th>B.LD</th>
<th>CNT.RST</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT</td>
<td>10000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>INIT</td>
<td>01000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>PREP</td>
<td>00100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>COMPARE</td>
<td>00010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>COPY</td>
<td>00001</td>
<td>0</td>
<td>1</td>
<td>THEN</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Expressions for next-state logic are similarly trivial. However, must look at incoming arcs to write them.
Look at the Incoming Arcs for WAIT

WAIT stays in WAIT on START' (self-loop not shown), and comes from COPY on the DONE signal.

\[ S_4^\prime = S_4 \cdot \text{START'} + S_0 \cdot \text{DONE} \]

Look at the Incoming Arcs for COPY

COPY comes from WAIT on START.

\[ S_4^\prime = S_4 \cdot \text{START} \]

Look at the Incoming Arcs for PREP

PREP comes from INIT, and from COPY on DONE'.

\[ S_3^\prime = S_3 + S_0 \cdot \text{DONE}' \]

Look at the Incoming Arcs for COMPARE

COMPARE comes from PREP, and stays in COMPARE on LAST' (self-loop not shown).

\[ S_1^\prime = S_2 + S_0 \cdot \text{LAST}' \]
Look at the Incoming Arcs for COPY

$S_0^+ = S_1 \cdot \text{LAST}$

$S_0^+ = S_1 \cdot \text{LAST}$

Finally, We are Done!

*And we’re done!*

Whew!

We can design an FSM for any code, for any computation.

If We Generalize the Instructions, We Have a Computer!

What if, instead, we design an FSM to execute some number of different statements.

We can use a datapath to manage bits.

We can use memory to give the FSM instructions as to what it should do (in terms of the FSM’s built-in statements).

We can use sequences of FSM states to execute each instruction.

That’s a computer!