

University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

From FSM to Computer

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];
```

What does
this mean?

```
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];
```

What does
this mean?

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 Array “Address” is Specified within Brackets

```
int values[10];
int idx;
int min = values[0];

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

The first array element is used here.

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

How Does the Code Work?

```
int values[10];
int idx;
int min = values[0];

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

These values must be filled in before the code executes.

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

How Does the Code Work?

```
int values[10];
int idx;
int min = values[0];

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

`idx` ranges from 1 to 9 in the `for` loop.

How Does the Code Work?

```
int values[10];
int idx;
int min = values[0];

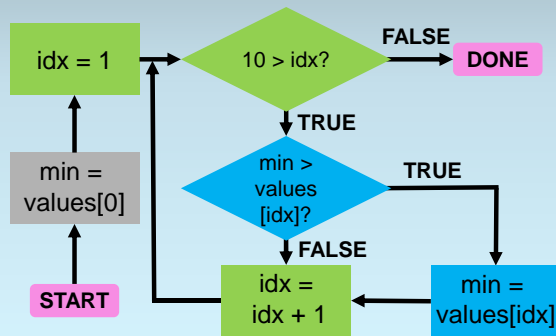
for (idx = 1; 10 > idx;
     idx = idx + 1) {
    if (min > values[idx]) {
        min = values[idx];
    }
}
```

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

Let's draw a flow chart.

When loop ends, `min` holds the minimum integer.

Flow Chart with Colors by Statement



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.

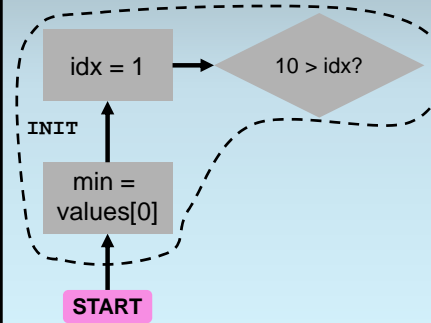
Not every flow chart box becomes a state.

For example, in our flow chart, we can

- Initialize **min**
- Initialize **idx**, and
- Perform the first comparison ($10 > 1$)

all in one cycle!

The First FSM State is INIT



We Can Predicate Execution with Logic

Predication means that something happens only under certain conditions.

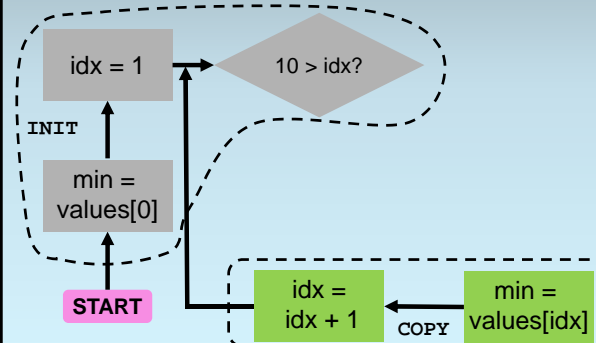
For example: If you give me an apple, I will give you a peach.

When our comparator is done, we can

- use the comparator output to determine whether **min** copies **values[idx]**, and
- increment **idx**

in the same cycle!

The Second FSM State May Copy a New Min Value



Sometimes the FSM Just Needs to Wait

How can we use our FSM?

Other logic must control the FSM.

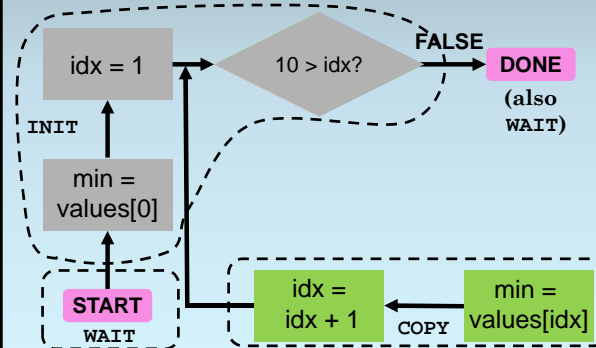
Using the FSM works as follows:

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

What’s left? Just the **if** statement.

Sometimes

- we may need more than one state
- to implement a simple step in the flow chart.

Our **serial comparator**

- takes bits from two **shift registers, A and B,**
- and uses a **counter** to measure 32 cycles.

We need to prepare **A, B,** and the **counter** for the comparison!

Preparing for the Serial Comparison Requires a State

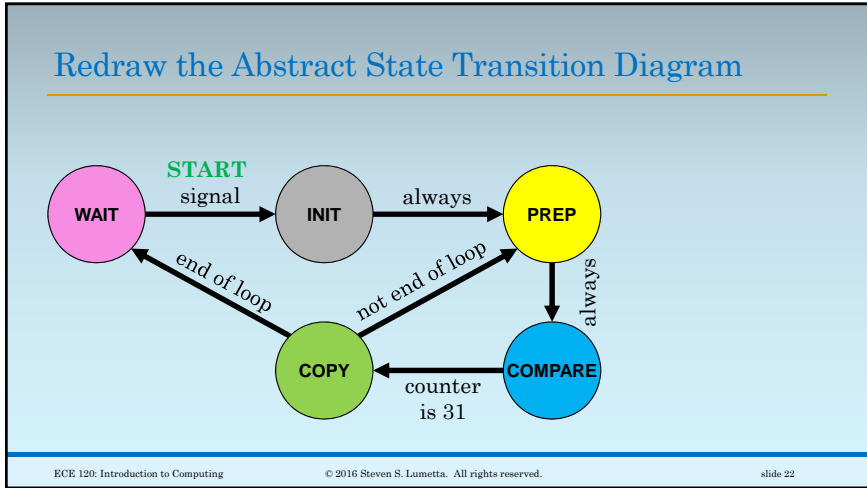
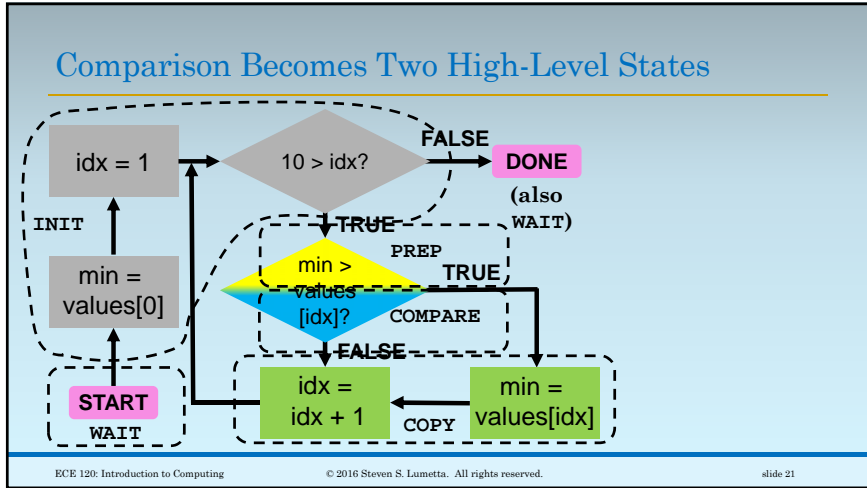
In the **PREP** state, the FSM

- Copies **min** to shift register **A,**
- Copies **values[idx]** to shift register **B,** and
- Resets the **counter** to 0.

In the **COMPARE** state (for 32 cycles), the

serial comparator performs the comparison.

When the **counter** has value 31, the FSM moves to the **COPY** state.



IDX is a Binary Counter with CNT and RST Inputs

Let's think again about the components.

- idx is a **32-bit 2's complement** value
 - used to count from 0 to 9, so
 - let's **use a 4-bit binary counter, IDX.**
- In **COPY**, we increment **IDX**.
- Let's **use a CNT input** to control the counting.
- If we reset the counter to 0 in **WAIT**,
 - it can count to 1 in **INIT**,
 - but we **need a reset input, RST.**

ECE 120: Introduction to Computing © 2016 Steven S. Lumetta. All rights reserved. slide 23

The Memory VALUES Can Use IDX as ADDR Input

values

- ten **32-bit 2's complement** values, so
- let's **use a 16x32-bit memory, VALUES.**

Notice that

- we **only read** from memory, and
- we **always read values[idx].**

So we can

- make the memory **always read,***
- and **connect the IDX counter to ADDR.**

*Whatever logic controls the FSM will have to override this simplification, of course.

ECE 120: Introduction to Computing © 2016 Steven S. Lumetta. All rights reserved. slide 24

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**.

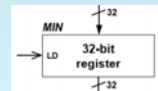
Load input **LD** controls changes to **MIN**.

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

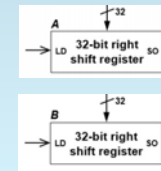
What about the shift registers **A** and **B**?

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

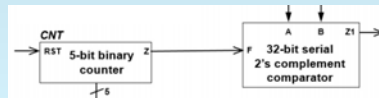
Finally, we need a counter to drive the serial comparator for 32 cycles.

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.

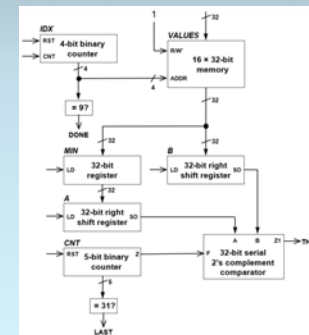
CNT should generate a zero output **Z**.



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.

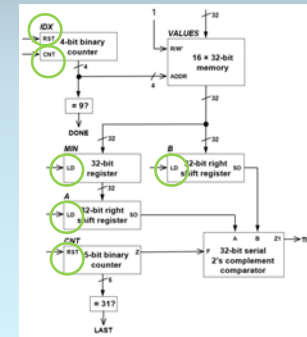
Remaining input signals to the components in the datapath are called **control signals**.

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

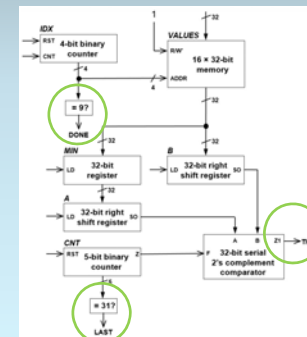
THEN a new minimum value has been found ($A > B$)

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

state	actions (simultaneous)	condition	next state
WAIT			

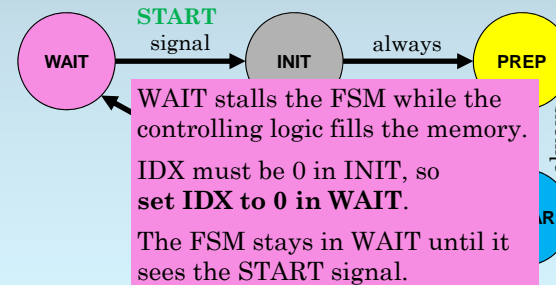
Let's make an abstract next state table.

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?



Write the Information for WAIT

state	actions (simultaneous)	condition	next state
WAIT	IDX ← 0	START START'	INIT WAIT

The **WAIT** state sets **IDX** to 0.

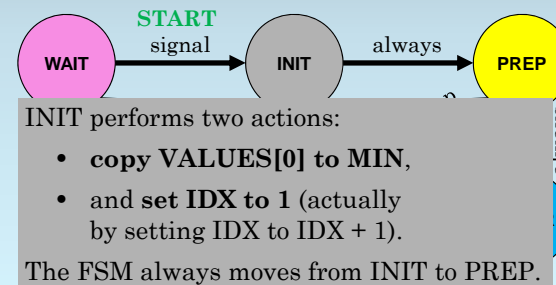
In RTL, we **write "IDX ← 0"** to indicate that the register **IDX** is filled with the value 0 (all 0 bits).

What about the next state(s)?

On **START**, move to **INIT**.

Otherwise, stay in **WAIT**.

What Happens in the INIT State?



Write the Information for INIT

state	actions (simultaneous)	condition	next state
WAIT	$IDX \leftarrow 0$	START START'	INIT WAIT
INIT	$MIN \leftarrow VALUES[IDX]$ $IDX \leftarrow IDX + 1$	(always)	PREP

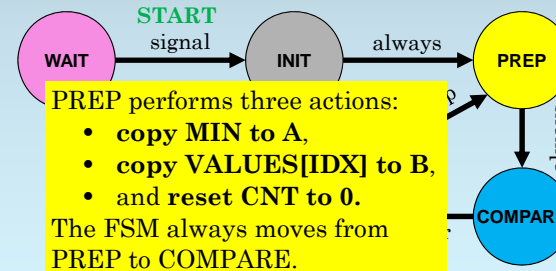
IMPORTANT: The order of actions here does not matter!

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.

What Happens in the PREP State?



Write the Information for PREP

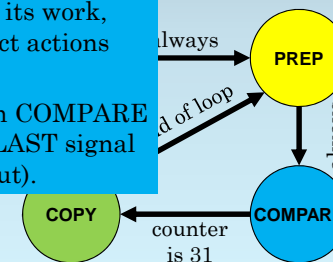
state	actions (simultaneous)	condition	next state
WAIT	$IDX \leftarrow 0$	START START'	INIT WAIT
INIT	$MIN \leftarrow VALUES[IDX]$ $IDX \leftarrow IDX + 1$	(always)	PREP
PREP	$A \leftarrow MIN$ $B \leftarrow VALUES[IDX]$ $CNT \leftarrow 0$	(always)	COMPARE

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

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).

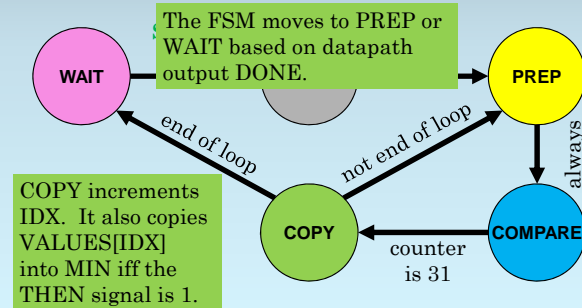


Write the Information for COMPARE

state	actions (simultaneous)	condition	next state
WAIT	$IDX \leftarrow 0$	START START'	INIT WAIT
INIT	$MIN \leftarrow VALUES[IDX]$ $IDX \leftarrow IDX + 1$	(always)	PREP
PREP	$A \leftarrow MIN$ $B \leftarrow VALUES[IDX]$ $CNT \leftarrow 0$	(always)	COMPARE
COMPARE	run serial comparator	LAST LAST'	COPY COMPARE

What are the next states?

What Happens in the COPY State?



Write the Information for COPY

state	actions (simultaneous)	condition	next state
WAIT	$IDX \leftarrow 0$	START START'	INIT WAIT
INIT	$MIN \leftarrow VALUES[IDX]$ $IDX \leftarrow IDX + 1$	(always)	PREP
PREP	$A \leftarrow MIN$ $B \leftarrow VALUES[IDX]$ $CNT \leftarrow 0$	(always)	COMPARE
COMPARE	run serial comparator	LAST LAST'	COPY COMPARE
COPY	THEN: $MIN \leftarrow VALUES[IDX]$ $IDX \leftarrow IDX + 1$	DONE DONE'	WAIT PREP

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?

state	$S_4S_3S_2S_1S_0$	IDX. RST	IDX. CNT	MIN. LD	A. LD	B. LD	CNT. RST
WAIT	10000	1	0	0	0	0	0
INIT	01000			Just set the other register inputs to 0 instead of don't care.			
PREP	00100						
COMPARE	00010						
COPY	00001						

Fill in the Table of FSM Outputs Based on RTL

INIT state:

$MIN \leftarrow VALUES[IDX], IDX \leftarrow IDX + 1$.

state	$S_4S_3S_2S_1S_0$	IDX. RST	IDX. CNT	MIN. LD	A. LD	B. LD	CNT. RST
WAIT	10000	1	0	0	0	0	0
INIT	01000	0	1	1	0	0	0
PREP	00100						
COMPARE	00010						
COPY	00001						

Fill in the Table of FSM Outputs Based on RTL

PREP state:

$A \leftarrow MIN, B \leftarrow VALUES[IDX], CNT \leftarrow 0$.

state	$S_4S_3S_2S_1S_0$	IDX. RST	IDX. CNT	MIN. LD	A. LD	B. LD	CNT. RST
WAIT	10000	1	0	0	0	0	0
INIT	01000	0	1	1	0	0	0
PREP	00100	0	0	0	1	1	1
COMPARE	00010						
COPY	00001						

Fill in the Table of FSM Outputs Based on RTL

COMPARE state: no RTL.

state	$S_4S_3S_2S_1S_0$	IDX. RST	IDX. CNT	MIN. LD	A. LD	B. LD	CNT. RST
WAIT	10000	1	0	0	0	0	0
INIT	01000	0	1	1	0	0	0
PREP	00100	0	0	0	1	1	1
COMPARE	00010	0	0	0	0	0	0
COPY	00001						

Fill in the Table of FSM Outputs Based on RTL

COPY state: $IDX \leftarrow IDX + 1$,
 THEN: $MIN \leftarrow VALUES[IDX]$.

state	$S_4S_3S_2S_1S_0$	IDX. RST	IDX. CNT	MIN. LD	A. LD	B. LD	CNT. RST
WAIT	10000	1	0	0	0	0	0
INIT	01000	0	1	1	0	0	0
PREP	00100	0	0	0	1	1	1
COMPARE	00010	0	0	0	0	0	0
COPY	00001	0	1	THEN	0	0	0

We Need Expressions for the Control Signals

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

Quickly now!

state	$S_4S_3S_2S_1S_0$	IDX. RST	IDX. CNT	MIN. LD	A. LD	B. LD	CNT. RST
WAIT	10000	1	0	0	0	0	0
INIT	01000	0	1	1	0	0	0
PREP	00100	0	0	0	1	1	1
COMPARE	00010	0	0	0	0	0	0
COPY	00001	0	1	THEN	0	0	0

We Need Expressions for the Control Signals

$IDX.RST = S_4$ $IDX.CNT = S_3 + S_0$

$MIN.LD = S_3 + THEN \cdot S_0$ (others) = S_2

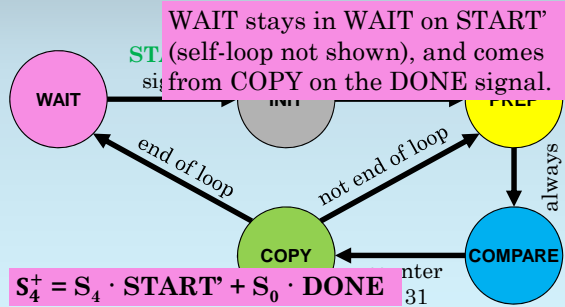
state	$S_4S_3S_2S_1S_0$	IDX. RST	IDX. CNT	MIN. LD	A. LD	B. LD	CNT. RST
WAIT	10000	1	0	0	0	0	0
INIT	01000	0	1	1	0	0	0
PREP	00100	0	0	0	1	1	1
COMPARE	00010	0	0	0	0	0	0
COPY	00001	0	1	THEN	0	0	0

And Expressions for Next-State Logic

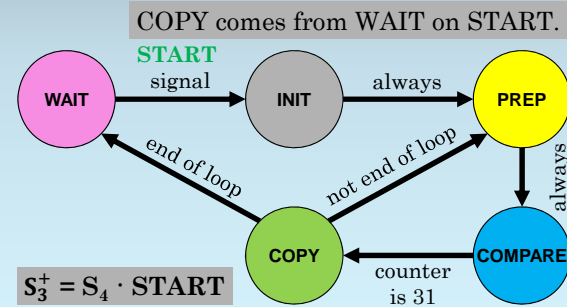
Expressions for next-state logic
 are similarly trivial.

However, must **look at incoming arcs**
 to write them.

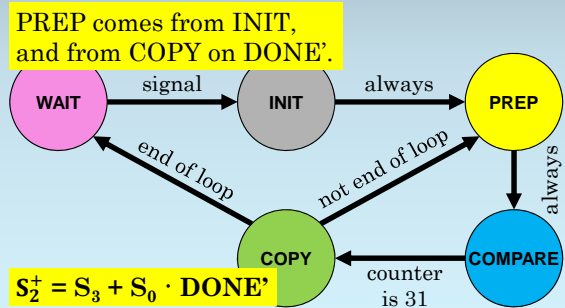
Look at the Incoming Arcs for WAIT



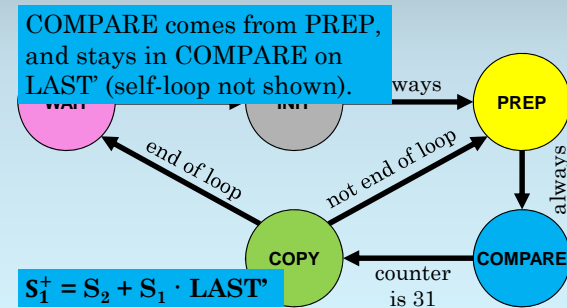
Look at the Incoming Arcs for COPY



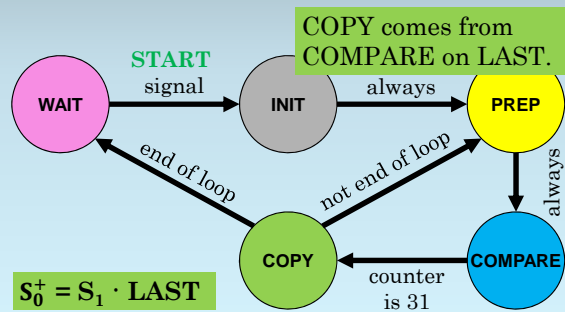
Look at the Incoming Arcs for PREP



Look at the Incoming Arcs for COMPARE



Look at the Incoming Arcs for COPY



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!