ECSE-323
Digital System Design

Datapath/Controller Lecture #1

Synchronous Digital Systems are often designed in a modular hierarchical fashion.

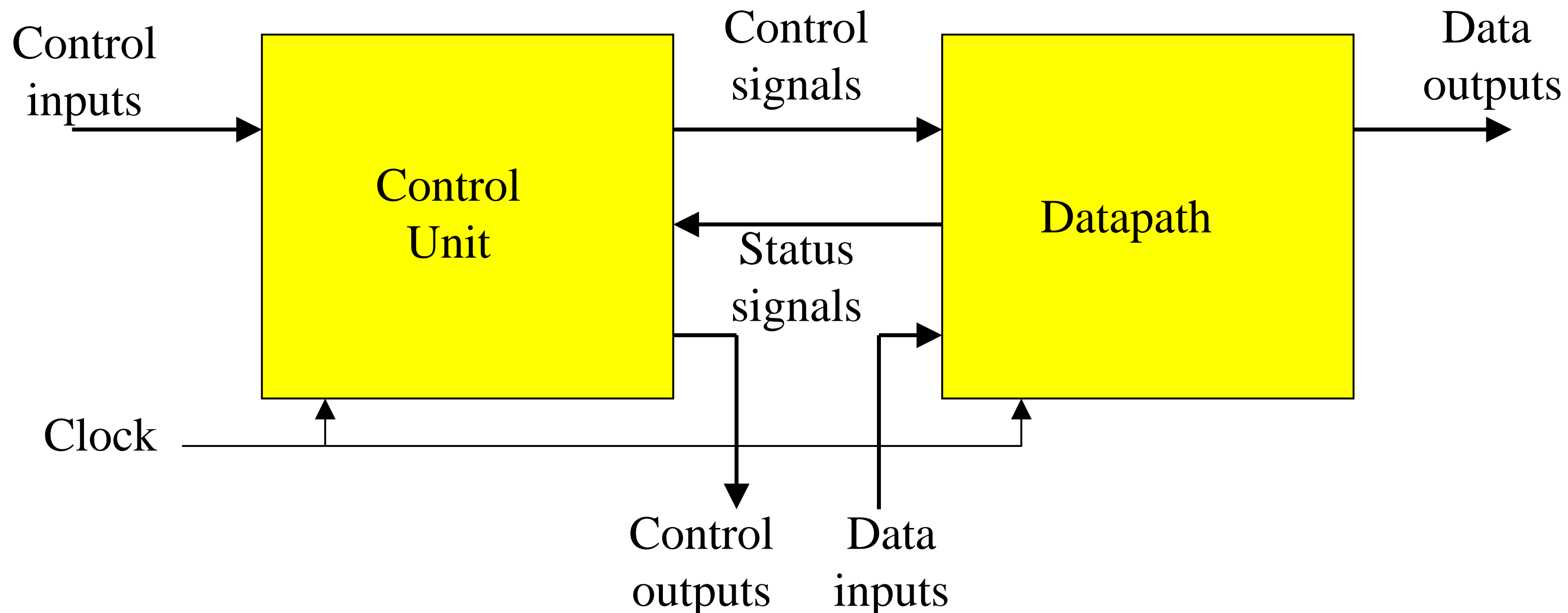
The system consists of modular subsystems, each of which performs some functional task, such as addition, multiplication, storage, counting, etc.

A good way to design modular digital systems is to partition the system into two types of modules:

Datapath Modules, which process and manipulate data.

Control Modules, which generate ***control signals*** that modify the processing of the datapath modules. The datapath modules also send ***status signals*** to the control modules.

A system built in this way is said to use a ***Datapath/Controller*** architecture.



Examples of Control Signals:

- Multiplexers - select
- Registers - load_en, clear, set
- Shift Registers - load_en, shift_en, clear, set
- Counters - count_en, clear, set, up/down

Examples of Control Inputs:

- start, reset, mode, begin
- all external non-data inputs (such as button presses)

Examples of Control Outputs:

- done, ready, error

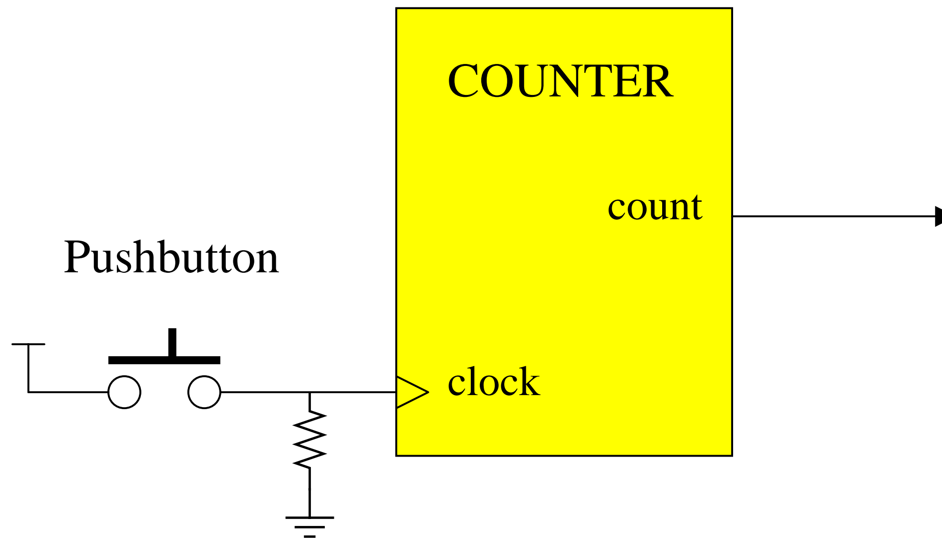
Examples of Status Signals:

- Serial Multipliers, Adders - done, ready
- Counters - count_decode, zero
- Comparators - AeqB, AltB, AgtB

A Simple Example - Pushbutton Counter

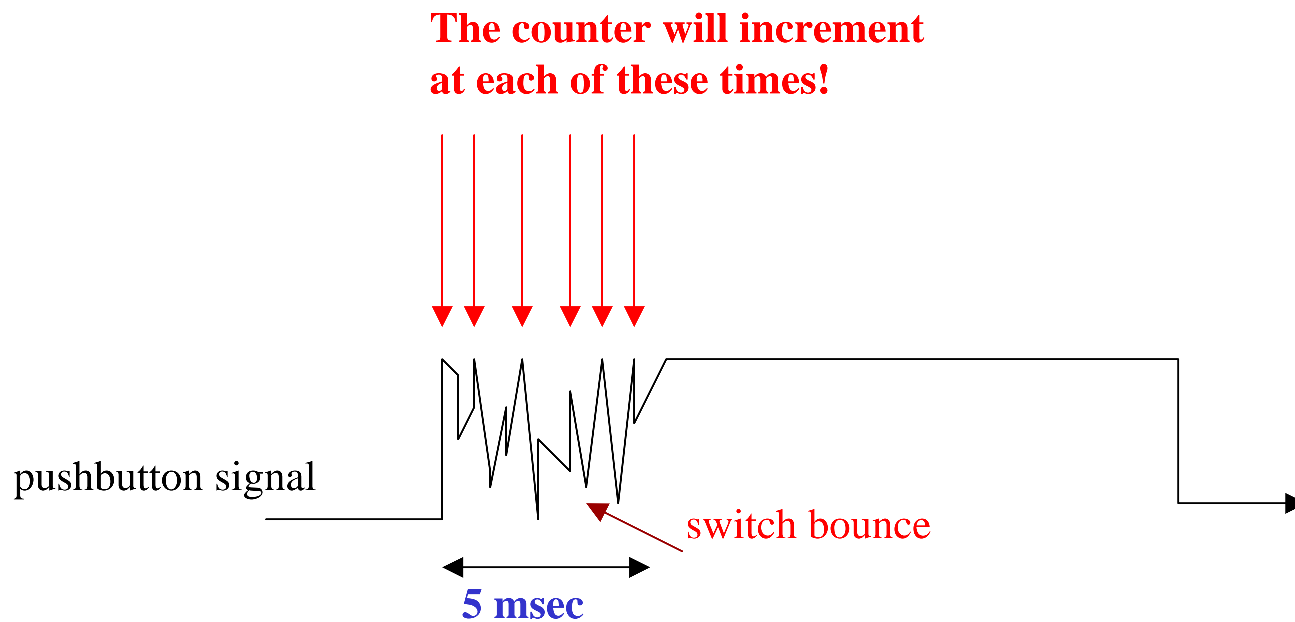
Design a circuit that counts the number of times a button has been pressed.

A Bad Solution:

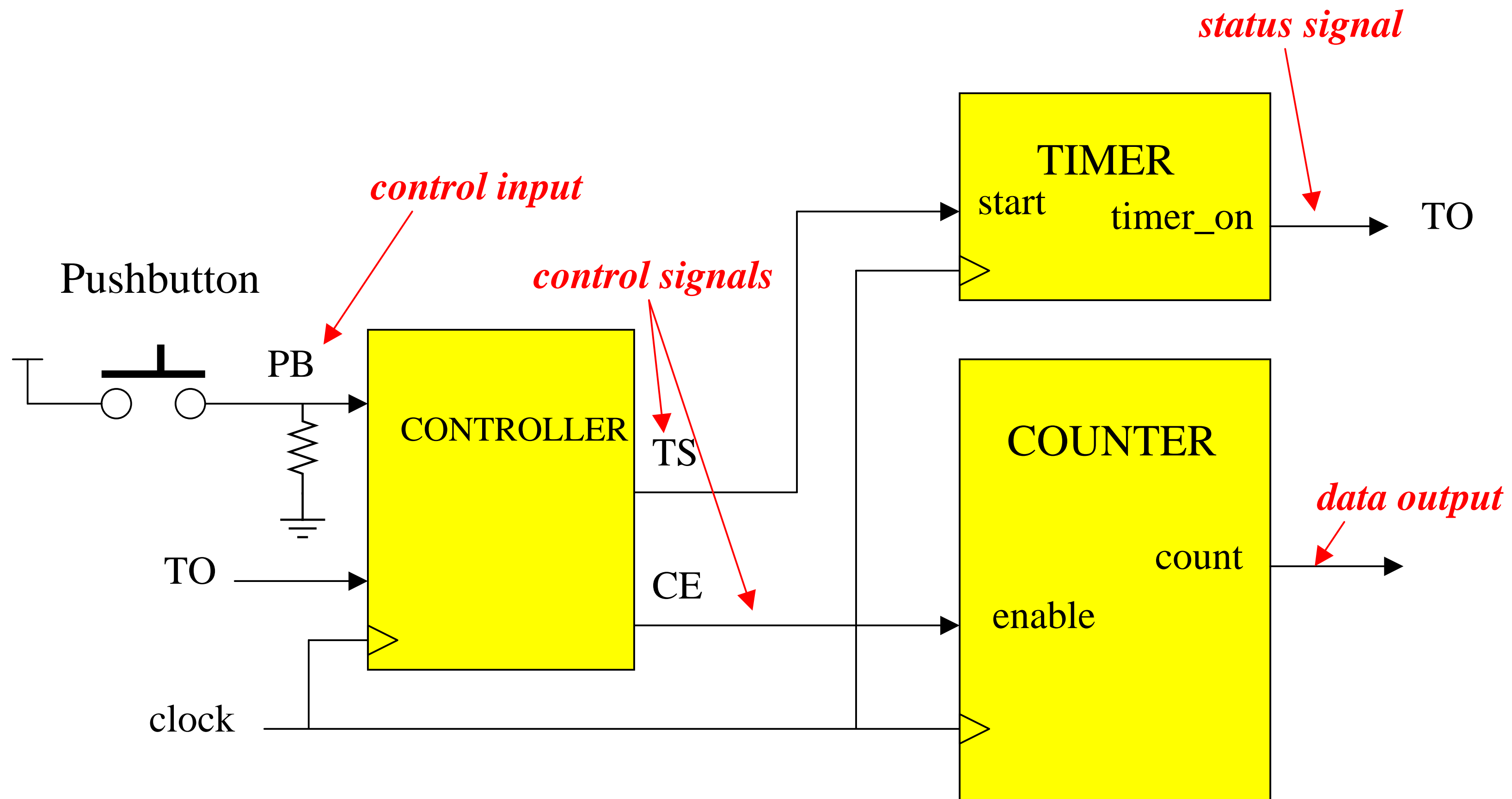


Why is the design bad?

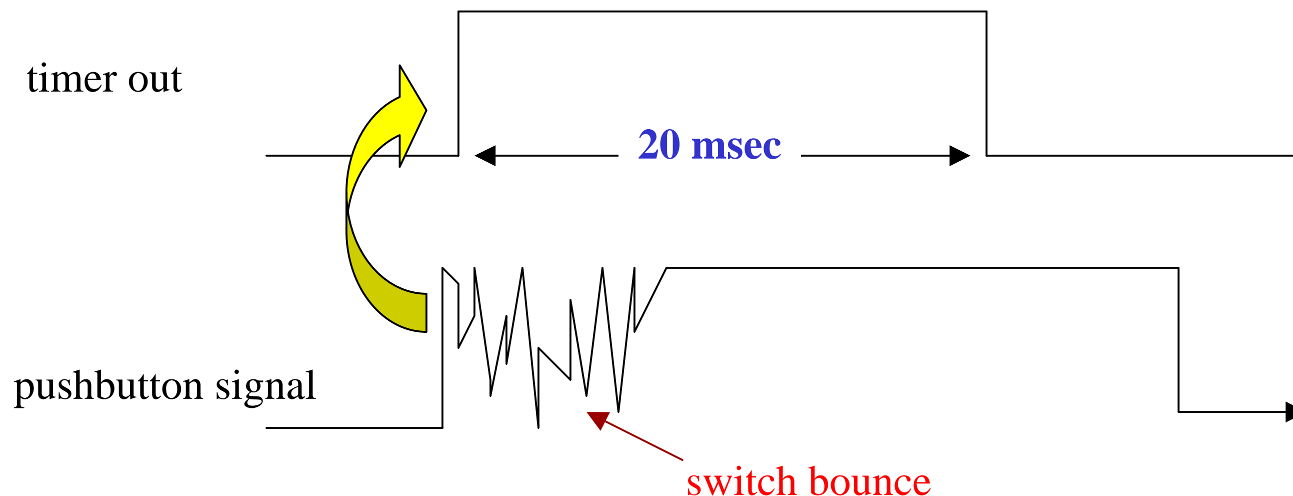
- The external signal from the pushbutton can be ***noisy*** due to ***switch bounce***, causing multiple counts for every button press.
- The count output changes at unpredictable times, which could cause difficulty (and glitches) synchronizing with other circuits.


The counter will increment
at each of these times!
pushbutton signal
switch bounce
5 msec

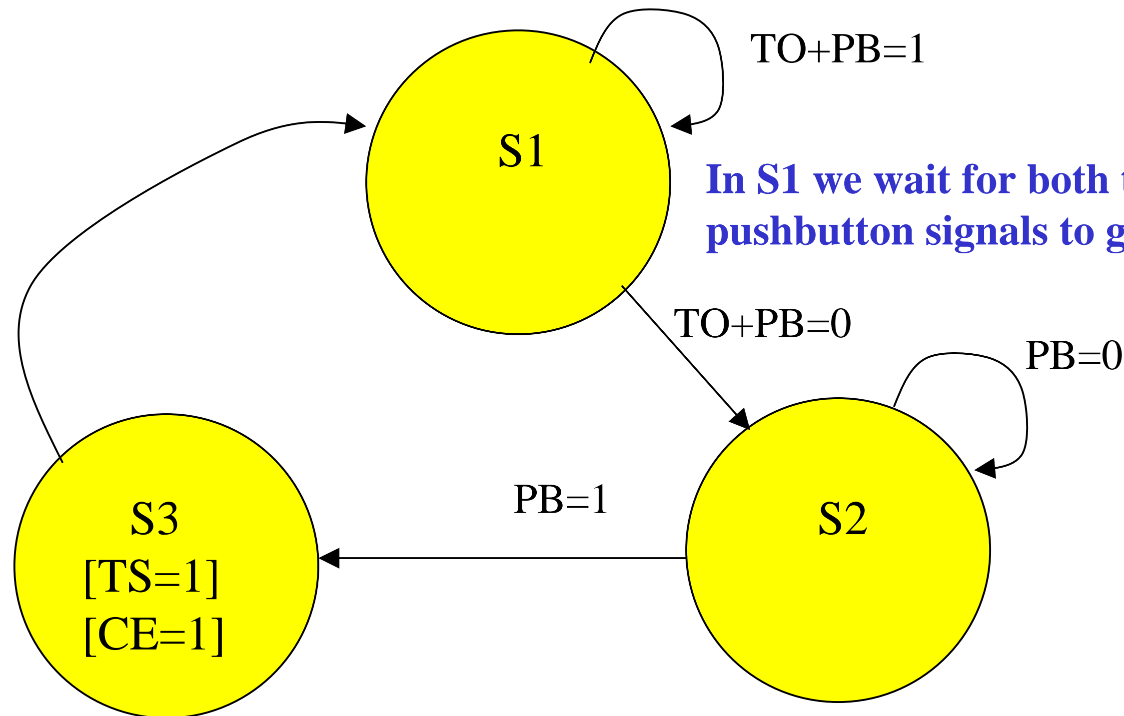
A Better Solution:



The timer is used to "wait out" the extra signal transitions created by switch bounce.



State Diagram for Counter Controller FSM



In S1 we wait for both the Timer and the pushbutton signals to go low.

In S3 we enable the counter and start the timer, then go to S1.

In S2 we wait for the pushbutton signal to go high.

In general, even if you don't have to worry about de-bouncing inputs, you should always check for transitions on input signals in a ***2-stage*** manner:

1. **Wait for the signal to go *low*.**
2. **Wait for the signal to go *high***
3. **(then do stuff…)**

This sort of 2-stage wait for a signal transition is useful for *starting* a sequence of actions, such as in a serial conversion from binary to BCD, or in a serial multiplier…

That is, to know when the actions should start, we check the value of the start input. We first wait for it to go low, then we wait for it to go high. Only then do we commence the operation.

The Boss gives you a more complicated TASK:

Design a circuit that finds the *minimum* and *maximum* values of a set of values stored in a set of 3 registers

15, 72, 5

General Approach to Design of Datapath/Controller Systems

1. Describe the function to be performed
2. Determine what datapath elements are needed
3. Specify the interconnections of the datapath elements
4. Identify the controller input and output signals
5. Sketch the sequence of control signal values needed to carry out the desired function
6. Design a Finite State Machine that will implement the required sequence
7. Simulate (by hand or by computer) the complete system to verify the proper execution of the desired function. Go back to step if there are any problems.
8. Implement and test complete system. Go back to step 2 if there are any problems with the implementation

Do the design by going step-by-step through the steps outlined in the previous slide

Step 1: Describe the function to be performed

It is usually easiest to describe the function in a "pseudo-code" form which can be easily translated into a hardware implementation.

Digression: Register-Transfer Language (RTL)

Datapath/Controller systems are often specified by describing the ***transfer of data between registers***, and the processing applied to the data.

These descriptions use a language known as a ***Register-Transfer Language***, or ***RTL***.

Register-Transfer Language (RTL)

RTL is a type of ***hardware description language***.

RTL descriptions are often used by hardware design software (simulators, synthesis) rather than VHDL,

Use of RTL can produce more efficient synthesis (smaller circuits) since the compiler has an easier job understanding just what it is that the designer wants.

RTL statements specify how, and when data is moved from one register to another.

The basic syntax of an RTL statement is:

$$R2 \leftarrow R1$$

This says that, on the next clock transition, the contents of register R1 are transferred to register R2.

Usually we don't want to transfer on every clock, but only when some *control condition* is asserted.

K : R2 ←R1

This says that, when Boolean condition K is true, on the next clock transition, the contents of register R1 are transferred to register R2.

Multiple transfer operations on the same clock edge can be specified with a comma-delimited list:

K : R2 ←R1, R1 ←R2

This statement describes a *register-swap* operation, controlled by signal **K**. In this example, when **K** is true the contents of register **R1** are copied to register **R2** and vice-versa.

You can specify multiple conditions on the comma-delimited list:

$$\mathbf{K : R2 \leftarrow R1,\ \overline{K} : R1 \leftarrow R2}$$

Operations on data are described as operations on the contents of register(s):

$$\mathbf{K : A \leftarrow A + R1}$$

This statement describes an *accumulation* operation, controlled by signal **K**.

Whenever **K** is true, the contents of register **A** are added to the contents of register **R1** and the result stored back into register **A**.

Memory Operations require address operands:

READ : DR ← M([AR])
WRITE : M([AR]) ← DR

The expression [R] means the contents of register R.
The expression M(A) means the memory slot with address A. Memory slots are treated like registers.

Register-Transfer Language is not covered in the text (except for a brief mention on p. 466)

RTL is used extensively in the specification of computer architectures, to be covered in the ***Microprocessors*** and ***Computer Architecture*** courses.

Returning back to the Min/Max circuit, we can write the following RTL description of the desired operation:

```
           RMAX ← R1, RMIN ← R1
R2<RMIN : RMIN ← R2, R2>RMAX : RMAX ← R2
R3<RMIN : RMIN ← R3, R3>RMAX : RMAX ← R3
```

Each line corresponds to events occuring in response to a single clock edge. The statements are read sequentially, top to bottom, corresponding to successive clocks.

Step 2: Determine what datapath elements are needed

We can read off our needs from the pseudo-code:

- Two ***registers*** to store the max and min values
- Two ***comparators***
- One ***multiplexers*** to control input to the RMAX and RMIN registers

(note: only 1 multiplexer is needed, since the set of possible inputs to the two registers is the same, so we can share the multiplexer. We could use two multiplexers, and save a clock cycle. There are usually such tradeoffs.)

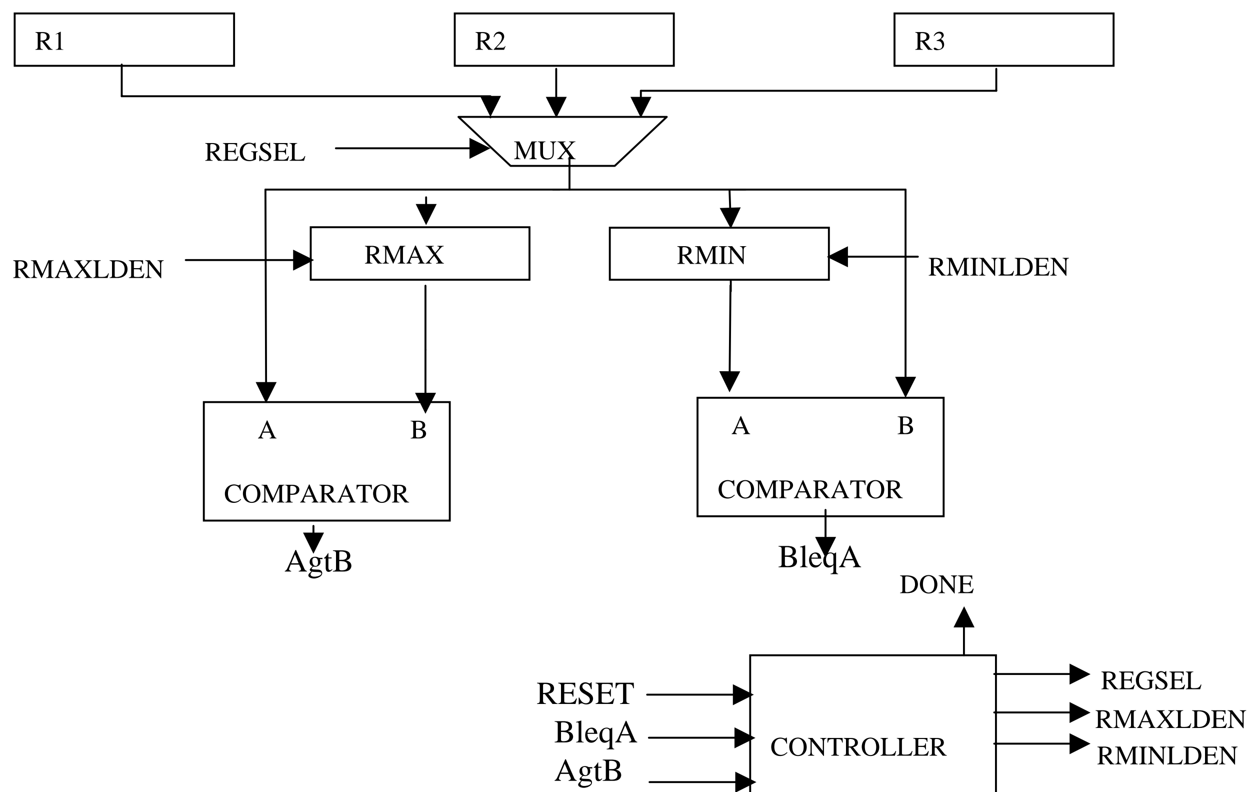
Step 3: Specify the interconnections of the datapath elements

Think of what data sources the datapath elements will operate on.

Some of the datapath elements will serve multiple purposes during the execution of an algorithm. Thus, different data sources may be operated on at different time. If this is the case, then Multiplexers should be used to choose between the different data sources.

Often, intermediate results need to be held before they are operated on. In this case, registers should be used to store these values.

The circuit implied by the RTL description is:



Note: ***Multiplexers*** are very important in the implementation of *datapaths*. They allow datapath elements (such as adders and registers) to be ***reused***, thereby reducing the size of the circuit.

Tri-state busses are also sometimes used to implement connections between datapath elements, as they require less circuitry than multiplexers.

Most FPGAs, however, only have a limited number of tri-state buffer circuits, usually located in output blocks. Therefore MUXs are usually used in FPGA datapath designs.

There are usually many different ways to implement a datapath.

Expensive datapath elements (such as comparators, and arithmetic operators) should be ***re-used*** and ***shared*** as much as possible. For such elements, their inputs and outputs should be connected to multiplexers which can distribute these signals to the appropriate destinations.

Enable signals can be used instead of multiplexers in some cases. We could use a multiplexer for each of the max and min value registers, but this would be overkill, as we can connect the register inputs together, and use the register load enables to determine which gets loaded.

For example, we could implement the MIN-MAX operation with just a single comparator, but it would require using more clock cycles, since we can only have one comparison operation per clock cycle:

```
                RMAX ← R1, RMIN ← R1
R2<RMIN   :     RMIN ← R2
R2>RMAX   :     RMAX ← R2
R3<RMIN   :     RMIN ← R3
R3>RMAX   :     RMAX ← R3
```

Step 4: Identify the controller input and output signals

Do this by looking at the control signals of the datapath elements.

- Inputs:
  - RESET (external control signal)
  - CLK
  - AgtB, BleqA (datapath status signals)
- Outputs:
  - RMINLDEN - enable loading of the RMIN register
  - RMAXLDEN - enable loading of the RMAX register
  - REGSEL - select the input for RMIN or RMAX
  - DONE - notify the user when the sorting is done

Step 5: Sketch the sequence of control signal values needed to carry out the desired function

Do this by looking at the pseudocode and determining, for each clock transition, what the datapath control signals need to be.

For example, for registers, you would decide when their load signals should be asserted, or when they should be cleared. For counters, you might determine when their count enables should be asserted.

Going back to the pseudocode for our example:

```
            RMAX ← R1, RMIN ← R1
R2<RMIN : RMIN ← R2, R2>RMAX : RMAX ← R2
R3<RMIN : RMIN ← R3, R3>RMAX : RMAX ← R3
```

We see that in the first clock we need to have the multiplexer control input set to select the R1 input. We also assert the load signals for both RMIN and RMAX.

In the second clock, we need to have the multiplexer select set to R2. However, we only assert the load for RMIN if the comparator signal BleqA is true (which implies R2<RMIN), and we only assert the load for RMAX if AgtB is true (implying R2>RMAX).

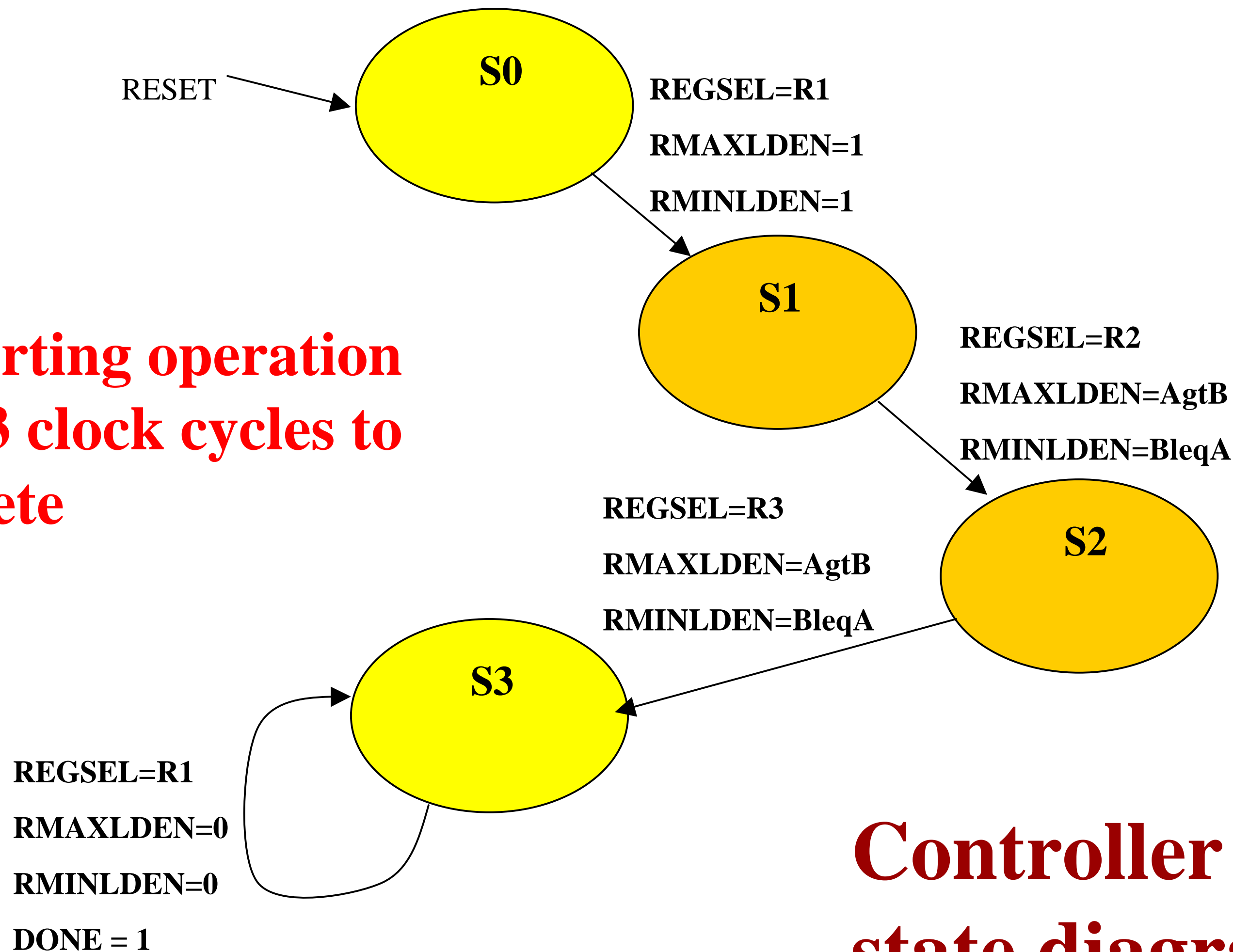
In the third clock, we need to have the multiplexer select set to R3. However, we only assert the load for RMIN if the comparator signal BleqA is true (which implies R3<RMIN), and we only assert the load for RMAX if AgtB is true (implying R3>RMAX).

Step 6: Design a Finite State Machine that will implement the required sequence

The simplest way to do this is to specify a different state for each line in the RTL description (or in the pseudocode). Note that there may be loops in the pseudocode.

In our example, there would be 3 states, if we used a Mealy Machine implementation (4 states if we use a separate DONE state).

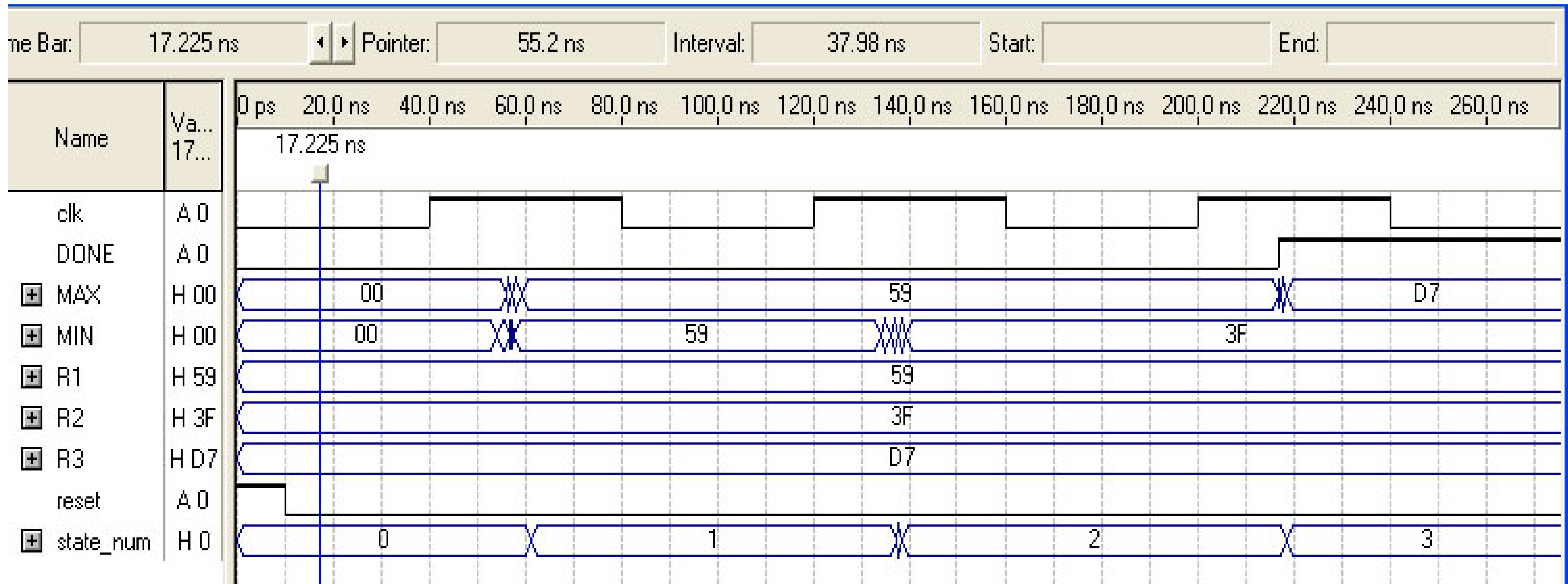
A Moore Machine would require more states, since we would not be able to have two different logical conditions applied on one clock transition.



The sorting operation takes 3 clock cycles to complete

Controller FSM state diagram (Mealy Machine)

Step 7: Simulate the complete system to verify the proper execution of the desired function



This Works! But note that the DONE line goes high slightly before the final value is ready. This is OK if the values are not used until the next clock edge.