CMPT 250 : Week 3 (Sept 19 to Sept 26)

1. DESIGN FROM FINITE STATE MACHINES (Continued)

1.1. ONE FLIP-FLOP PER STATE METHOD

From a state diagram specification, a sequencer can be constructed using the one-D-flip-flop-per-state method instead of having to construct the characteristic table. The states are encoded as:

```
S0 = 001,
S1 = 010,
S2 = 100
```

Denote the *i*th bit of this encoding by Qi. Then Qi is asserted only when we are in state Si. By examining the state diagram construct product terms for each transition arrow consisting of the departing state (Qi) and the external boolean input expression labelling the arrow. Then sum the product terms of all arrows with the same destination state Qj. This sum of products is the flip flop input function for the flip flop representing Sj.

The resulting flip-flop input functions are:

```
D0 = data’*Q0 + eq0*Q1

D1 = data*Q0 + data*Q2

D2 = eq0’*Q1 + data’*Q2
```

Only one of Q0, Q1, or Q2 is asserted when we are in states S0, S1, or S2 respectively.

1.2. MULTIPLEXER / REGISTER / DECODER METHOD

With this method, we summarize for each state the required inputs necessary to achieve the next state suggested by the state diagram. We then define FF input function assignments for the set of transitions from a particular state. Using the example state diagram above:

```
PRES    NEXT    REQD    D FF INPUTS
STATE   STATE   INPUTS
-----------------------------------
00      00      data’   D1=0, D0=data
00      01      data

01      00      eq0     D1=eq0’, d0=0
01      10      eq0’

10      01      data    D1=data’, D0=data
10      10      data’
```

From this table we "sum" the individual cases for each flip-flop (without simplification):

```
D1 = 0*Q1’Q0’ + eq0’*Q1’Q0 + data’*Q1Q0’

D0 = data*Q1’Q0’ + 0*Q1’Q0 + data*Q1Q0’
```

Now it is possible to implement these functions with gate level logic. However these equations suggest using 4x1 multiplexers whose behavior is defined by:

```
mux(d1,d1,d2,d3,s0,s1) =
        d0*s1’s0’ + d1*s1’s0 + d2*s1s0’ + d3*s1s0
```

We can implement the required combination logic using one 4x1 multiplexer per flip-flop by substituting:

```
for D1:
        s1 = Q1, s0 = Q0
        d0 = d3 = 0, d1 = eq0’, d2 = data’
for D0:
        s1 = Q1, s0 = Q0
        d0 = data, d2 = data, d1 = d3 = 0
```

A decoder can be used in the same manner as with the excitation method to obtain the outputs S0, S1, S2 from the state outputs Q1 and Q0.

2. SPECIFICATION WITH ALGORITHMIC STATE MACHINES

Every algorithm can be implemented in hardware. While most programming languages provide a means for describing sequential processes, hardware implementations include the need to take advantage of parallel processing wherever possible. Therefore a notation system is required that can describe both sequential and parallel steps in an algorithm. One such notation system is the **Algorithmic State Machine** or ASM, which can be represented as a directed graph. With this notation system algorithms are defined by diagrams using three types of vertex, more commonly called boxes:

1. **state box**: labelled rectangle, possibly containing "assignment

statemtents" called *register transfer statements*, or variables that define external output statements. Each register transfer statement defines a task that can be performed in one clock cycle, each variable defines an output or status bit that is to be enabled for one clock cycle. There is only one arrow leaving each state box, which can have multiple arrows entering it.

2. **decision box**: hexagon, defining one or more boolean control signals; that is, a boolean function of one or more variables. There is always one input arrow into a condition box, and two arrows out for each boolean control signal. Each output arrow is labelled with one of two logic levels of the associated signal.
3. **conditional action box**: Oval, containing register transfer expressions and possibly variables, as described for state boxes. Oval boxes have one or more arrows coming from a decision box, and one arrow leaving.

An **ASM block** consists of a single state box as well as all the decision and conditional action boxes that can be reached by arrow tracing from the state box until another state box is encountered. An ASM block defines all the "activity" that can take place in one clock cycle.

The example on the next page illustrates an ASM diagram for a machine to find the sum of N numbers. This machine accepts as input a sequence of numbers beginning with the value of N, the number of numbers to be summed. Each number is provided only when requested, using an set of interface signal lines:

- **`rdy`**: An output by the sum machine when it has completed a task and is waiting for a new request (via **`st`**) to do a new sum.
- **`data`**: An input that tells the sum machine when there is valid data on its input bus. It is also used to start the system.
- **`ack`**: An output by the sum machine to indicate that it has satisfactorily processed the data that was on the input bus and therefore the value does not need to be maintained any longer on the bus.

3. CLASSIFICATION OF INPUTS AND OUTPUTS

The inputs and outputs of a "black-box" can be classified according to the role they will play in the behavior of the circuit:

- **Data inputs**: set of signal lines that represent the data to be manipulated by the circuit.
- **Control inputs**: set of signal lines that select or otherwise affect

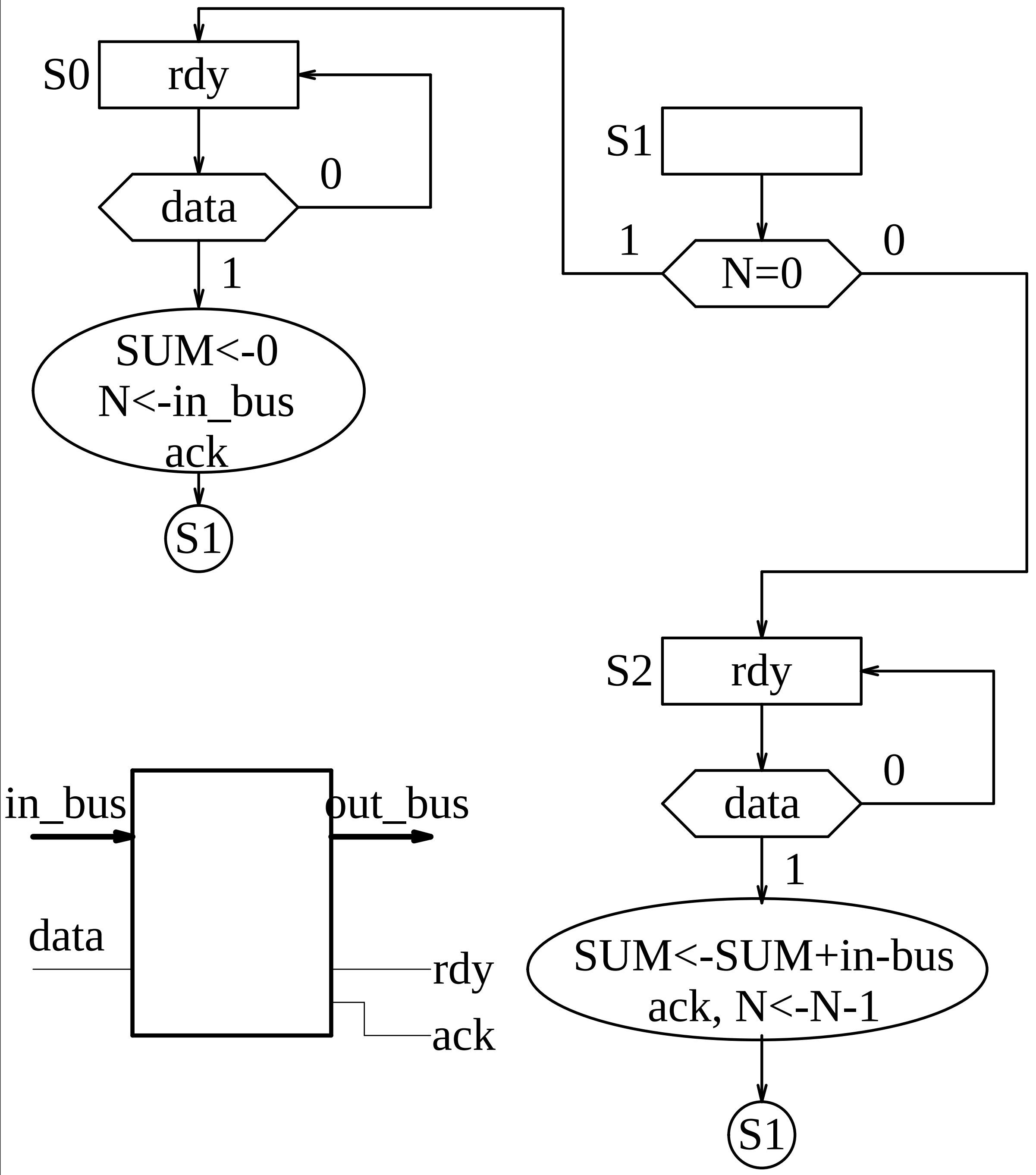


Figure 2-1: ASM Diagram for the Sum Machine

whether a circuit performs its function, and which functions are performed by multi-functional circuits.

- **Data outputs**: set of signal lines that represent the result generated by the circuit.
- **Status outputs**: Set of signal lines used to odentify or interpret the current output or state of the system.

For example the 4x1 MUX described previously has 4 data input lines and 2 control inputs that select which data input is delivered to the single data output bit.

As a second example, a 4 bit full adder consists of 8 data input lines (4 per operand), 4 data output lines, and 1 status output, if the carry-out is interpreted as an overflow indicator. The carry-in can be interpreted as a third operand, and thus a data input.

4. DESIGN FROM ASM SPECIFICATIONS

There is a general design strategy based on the information contained in the ASM specification. This design method leads to a machine architecture consisting of two principal components:

- A **processing component** which defines the logic required to provide the computational requirements and data flow paths necessary for the machine to perform the task for which it is designed.
- A **controller component** which manages the order in which the various components of the processing logic are enabled so that they can perform their tasks. The controller itself consists of two parts:
  - A **sequencer** which keeps track of which ASM block is currently being executed by the processing logic and determines which ASM block will be executed in the next clock cycle.
  - A **control point selector** which generates the necessary control signal assignments to every control input within the processing component.

ASM Design Method

1. Construct a black box for the circuit and identify each input and output and its role as a data carrier, function select, circuit enable, etc.
2. Formulate an algorithm for the process, expressed in an ASM diagram.
3. From the ASM identify all register and other memory requirements

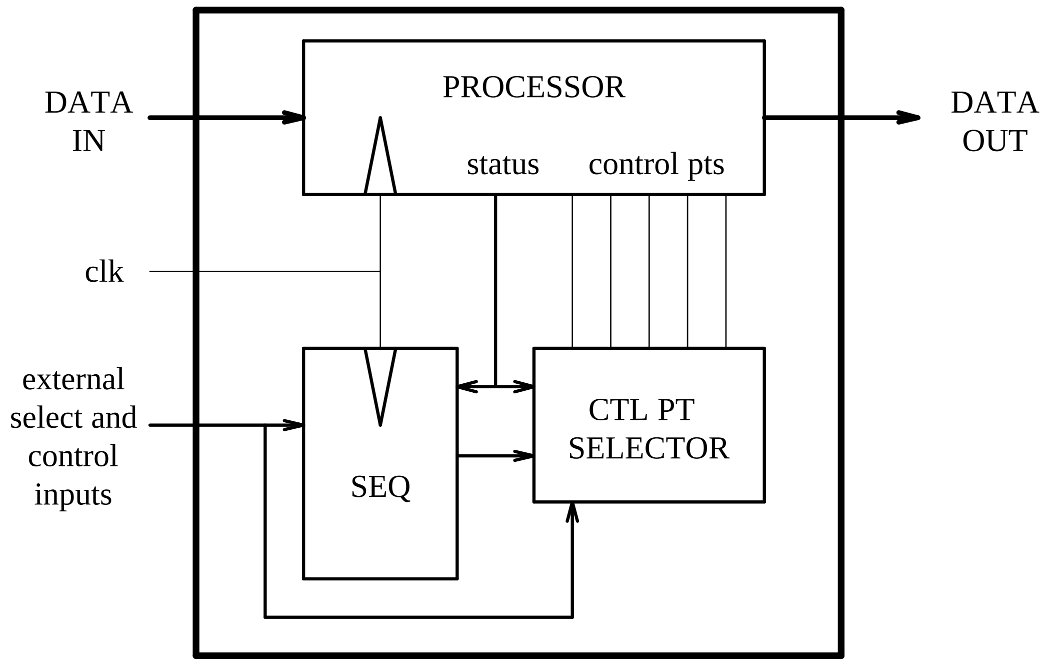


Figure 4-1: Typical System Architecture from ASM Design

and determine the operational capabilities of each required register. Express each component by a behavioral description.

4. From the ASM condition boxes determine the combinational requirements for generating each status bit that controls the sequencing of the steps of the algorithm. Agaion, define each component with a behavioral description.

5. Group the Register Transfer Expressions (RTEs) in the ASM to determine how many different data paths enter or leave each register. This will determine the steering logic requirements.

6. Construct a structural specification in the form of a logic diagram for the processor, showing the required register and combinational components and the data paths between them. For each component, label the necessary control inputs than must be used to obtain the functional behavior suggested by the register transfer expressions.

7. From the ASM design the sequencer by constructing a FSM. Each state of the FSM corresponds to a block of the ASM, and a state transition arrow is drawn between two states if there is an arrow

connecting the corresponding two ASM blocks. The state transition arrow of the FSM is labelled by a product of the input literals or status bits controlling the sequencing in the ASM.

8. From the ASM design the control point selector. For each ASM block determine all control inputs that must be enabled to perform all tasks defined within the block. In each case, construct a boolean function whose variables include the the current state of the sequencer and any status bits specified by condition boxes within the block.

The following example illustrates the method for the example ASM given above to sum N numbers.

5. DATAPATH DESIGN

To determine the register requirements from the ASM, group the RTEs in the ASM by `<destination>`; that is all RTEs with the same destination are grouped together:

- **SUM:** `SUM <- 0, SUM<- SUM + in-bus`. This can be implemented with a parallel load register, which will have one control input to enable it to load its contents in parallel and one control input to clear. In the diagram the control inputs will be labelled "`ls`" and "`cs`."
- **N:** `N <- in_bus, N <- N-1`. `N` can be implemented with a "down" counter that also has a parallel load capability. Again, two functions means that the package will have two control inputs. `ln` will denote the load control input and `dn` will identify the "decrement" (i.e., count down) control input.

From the ASM diagram,in addition to the registers `SUM`, and `N`, three combinational components are required; one to add `SUM` and `in-bus`, one to compare N with 0, and one to control the placing of `SUM` on `out_bus`. These packages can be constructed from simpler components or a commercial package may be available with the desired capabilities.

The identified functional components are organized in a *structural description* such as a logic diagram, by connecting them with bus lines to show the required PROCESSOR (DATAPATH) suggested by the RTEs is given in Figure 4-2 (next page).

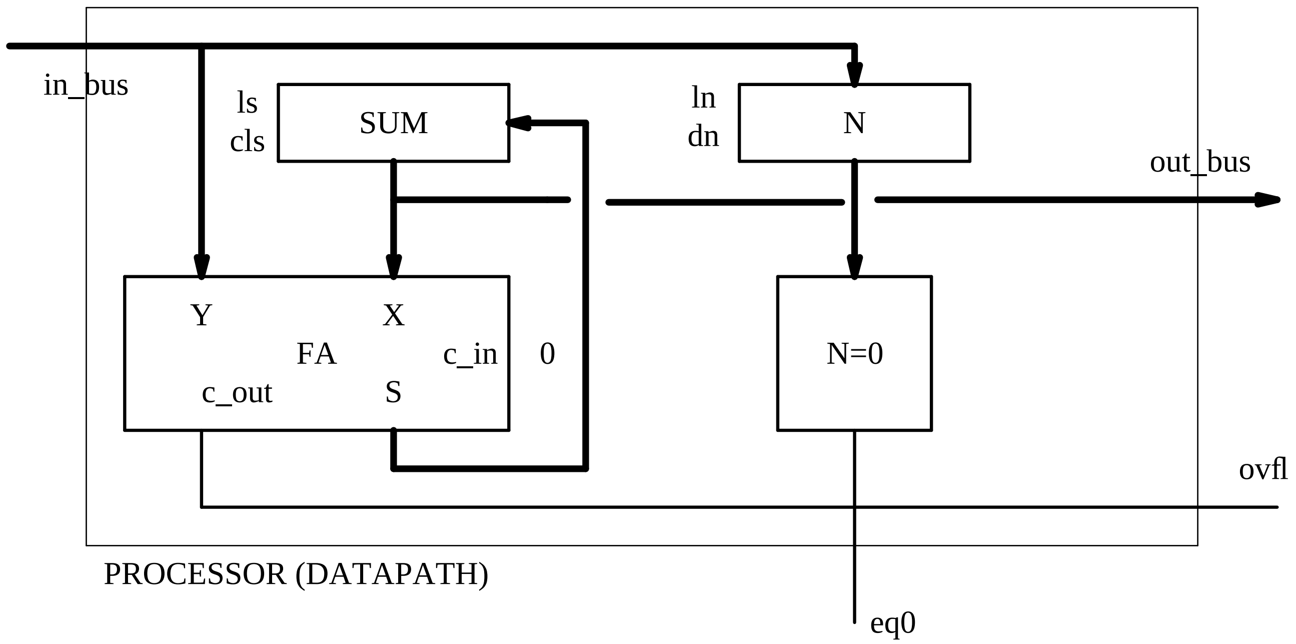
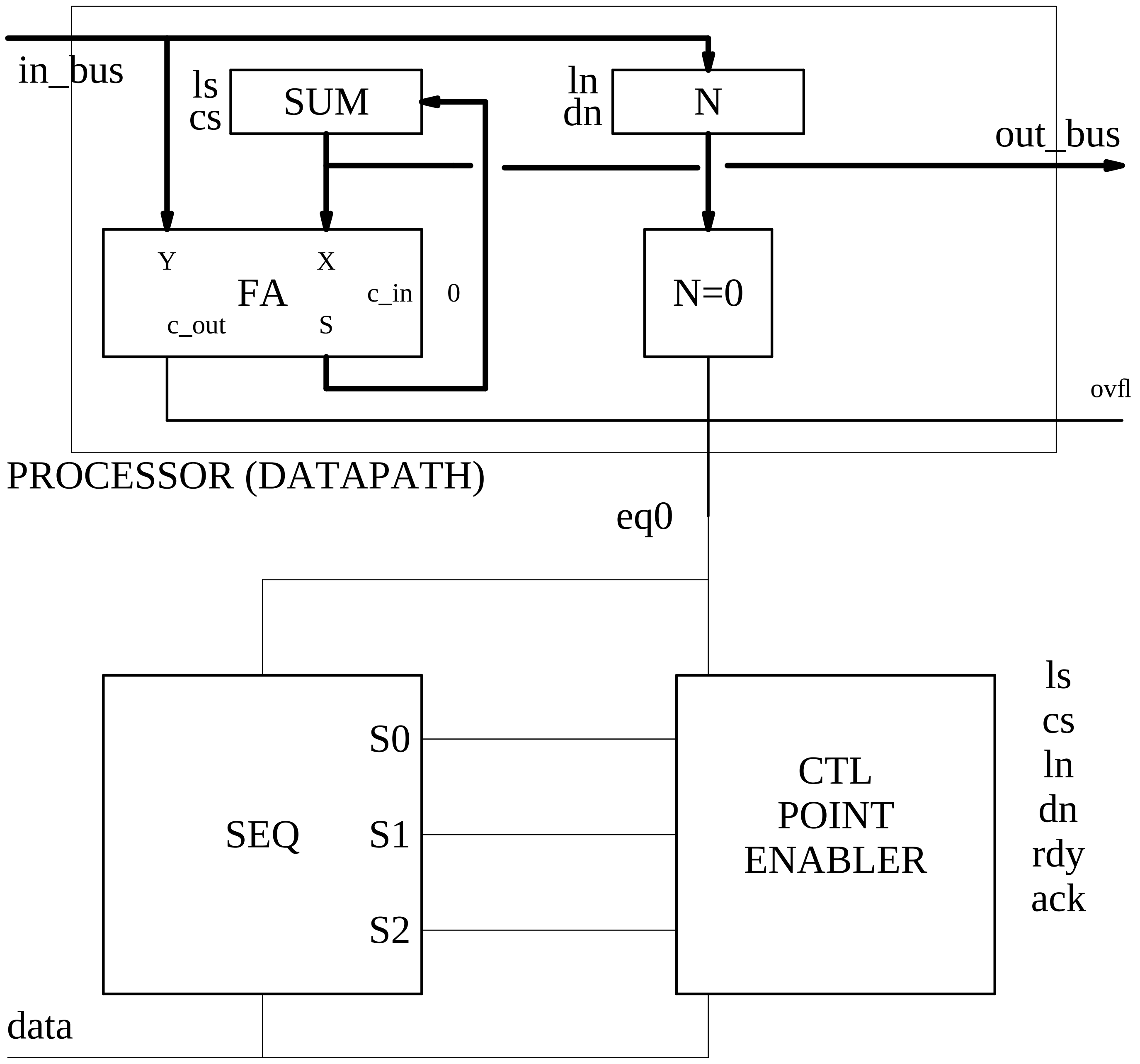


Figure 5-1: Processing Component of the SUM Circuit

6. CONTROLLER DESIGN

The controller is divided into two modules: the sequencer, that keeps track of which ASM block the datapath is performing, and the control point enabler, which asserts the appropriate control inputs in the datapath depending on the state of the sequencer, the external control inputs, and the status of the datapath. The black box diagrams for these components for the example being designed are given in the diagram on the next page. Each component will be designed separately:


in_bus
ls
cs
SUM
ln
dn
N
out_bus
Y
X
FA
c_in
0
c_out
S
N=0
ovfl
PROCESSOR (DATAPATH)
eq0
SEQ
S0
S1
S2
CTL
POINT
ENABLER
ls
cs
ln
dn
rdy
ack
data

6.1. Sequencer Design

The sequencer is obtained from the ASM diagram by constructing a state diagram, where each state corresponds to an ASM block:

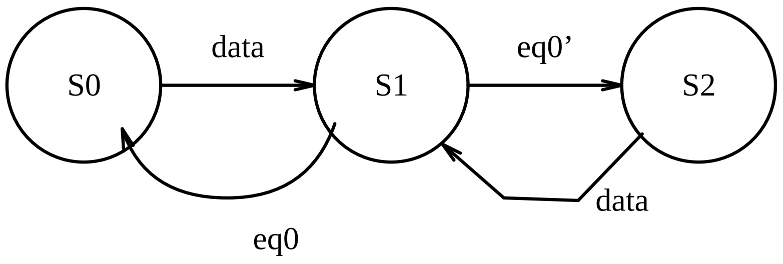


Figure 6-1: State Diagram for the Sequencer of the SUM Circuit

The structural description (logic diagram) can be constructed from this state diagram using any of the methods (state-transition, 1 D-flip-flop per state, or multiplexer/register/decoder) described previously.

6.2. Control Point Enabler Design

The control point enabler must determine which control inputs to assert based on the current ASM block (provided by the sequencer), the external control inputs, and the status outputs (from the processor).

The control point enabler can be developed from a function table. This table describes the appropriate control points to enable (i.e., set to logic 1) when in a particular ASM block (as specified by the values of **`S0`**, **`S1`**, and **`S2`**). This of course depends on the values of the external control and status inputs `data` and `eq0`.

Alternatively, one can capture the information of the ASM diagram algebraically by examining which ASM block and which control inputs must be asserted for each control input to be enabled.

In the example above, the following boolean equations can be derived:

```
ls = S2*data
cs = S0*data
ln = S0*data
dn = S2*data
rdy = S0 + S2
ack = S0*data + S2*data
```

6.3. Step Action Sequences

In situations where the same boolean expressions define a number of outputs, it is more convenient to express these relationships using *step action statements*. A step action statement is simply a boolean expression (called the *control expression* followed by a list of outputs that are to be asserted when the control expression is true.

A combinational circuit can be defined by a set of step action statements called a *step action sequence*. The above example, expressed as a step action sequence is:

```
S0+S2: rdy
S2*data: ack, ls, dn
S0*data: ack, cs, ln
```

Note that each step action statement defines the value of all control point enabler outputs: `{ls,cs,ln,dn,rdy,ack}`. Those listed explicitly in a step action statement are assumed to be given logic 1 and those not listed explicitly are to be given the value 0 if the control expression is true. Finally, if none of the control expressions is true then all control outputs are to be assigned the value 0.

Rather than implementing the Conrol Point Enabler with discrete gate components, programmable logic devices are frequently used.

7. GUIDELINES ON DEFINING ASMs

The formulation of a behavioral description expressed as an ASM is based on identifying important aspects of the problem from the original description. Since this may be incomplete, design decisions are often required. The following steps are a guideline as to what needs to be addressed:

1. Express the solution to the problem algorithmically, using pseudo-code or a programming language. To obtain the example ASM above, the following algorithm was used as the starting point:

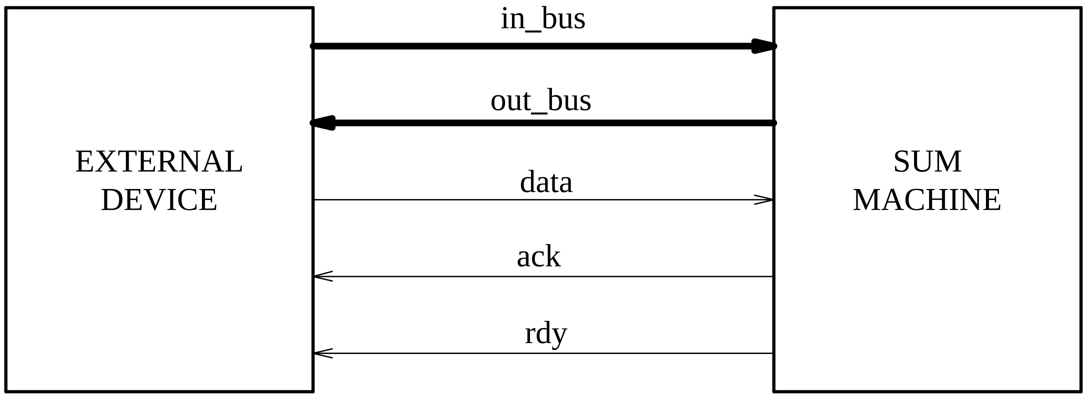
```
SUM = 0;
N = GET_INPUT();
WHILE (N > 0)
BEGIN
   SUM = SUM + GET_INPUT();
   N = N - 1;
END;
```

2. Translate the algorithm into a sequence of register transfer statements. This may require expressing some statements of the original algorithm using more than one register transfer statement if the required operation cannot be performed in one clock cycle.

3. Group adjacent register transfer statements into a single ASM block if there are no dependencies between the statements. This step aims to take advantage of any parallelism that may be present and so improve the performance of the eventual solution.

4. Introduce control signals to manage the interface between the device being defined and any external devices with which it must communicate. In the example a control signal from the external device (`data`) is required to intiate action by the device being designed and to advise it when there is valid data on the input bus, `in_bus`. As well the external device requires control signals to advise it when the device being designed requires input (`rdy`), and when it no longer needs the data on `in_bus` (`ack`).

   The communications interface can often be characterized using a diagram showing all the required signal lines between the device under construction and the external device(s) with which it must communicate. The ASM defined previously was based on the following communications interface definition:


EXTERNAL DEVICE
in_bus
out_bus
data
ack
rdy
SUM MACHINE