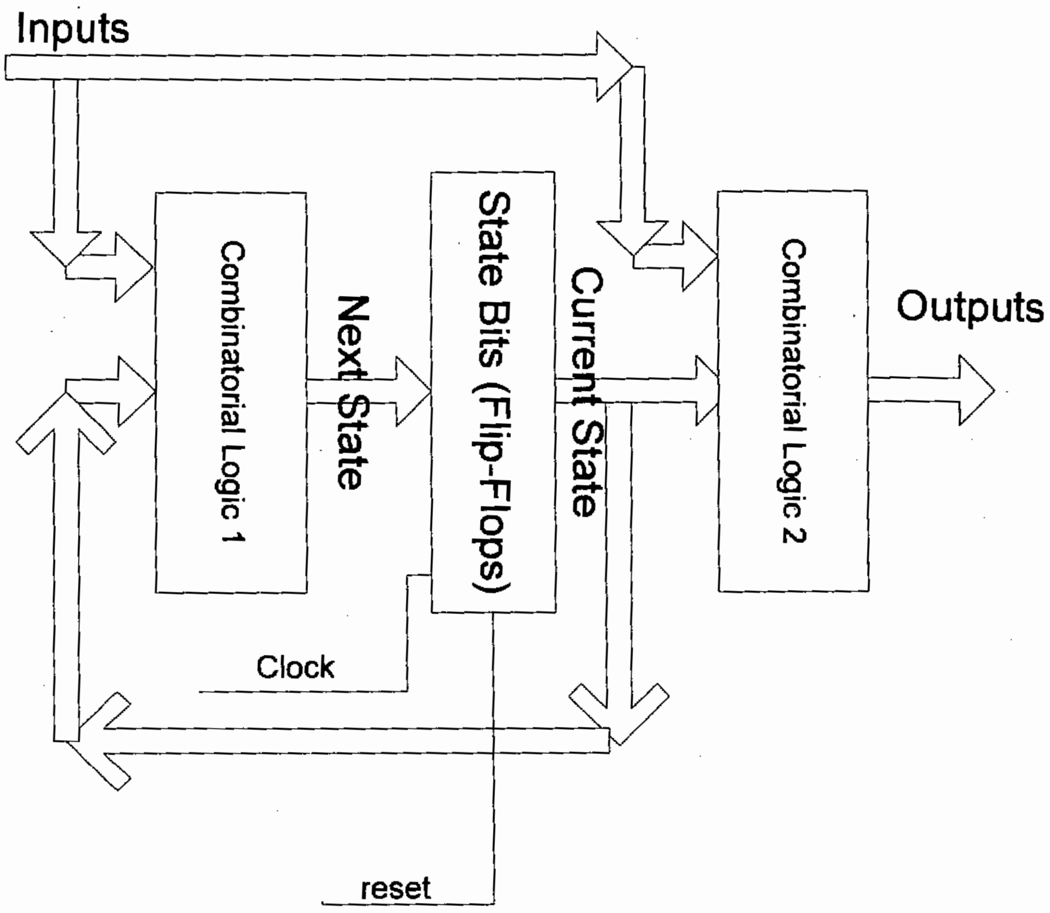
State Machines and Glitches

State Machines

What is a state machine?

State machine – a practical definition:

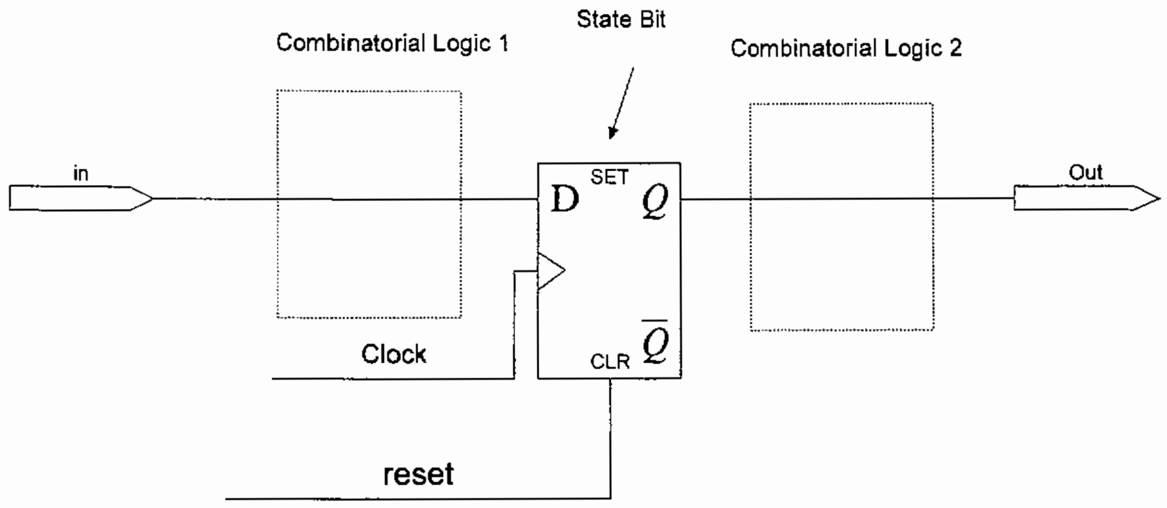
A state machine has the following structure:



Explanation:

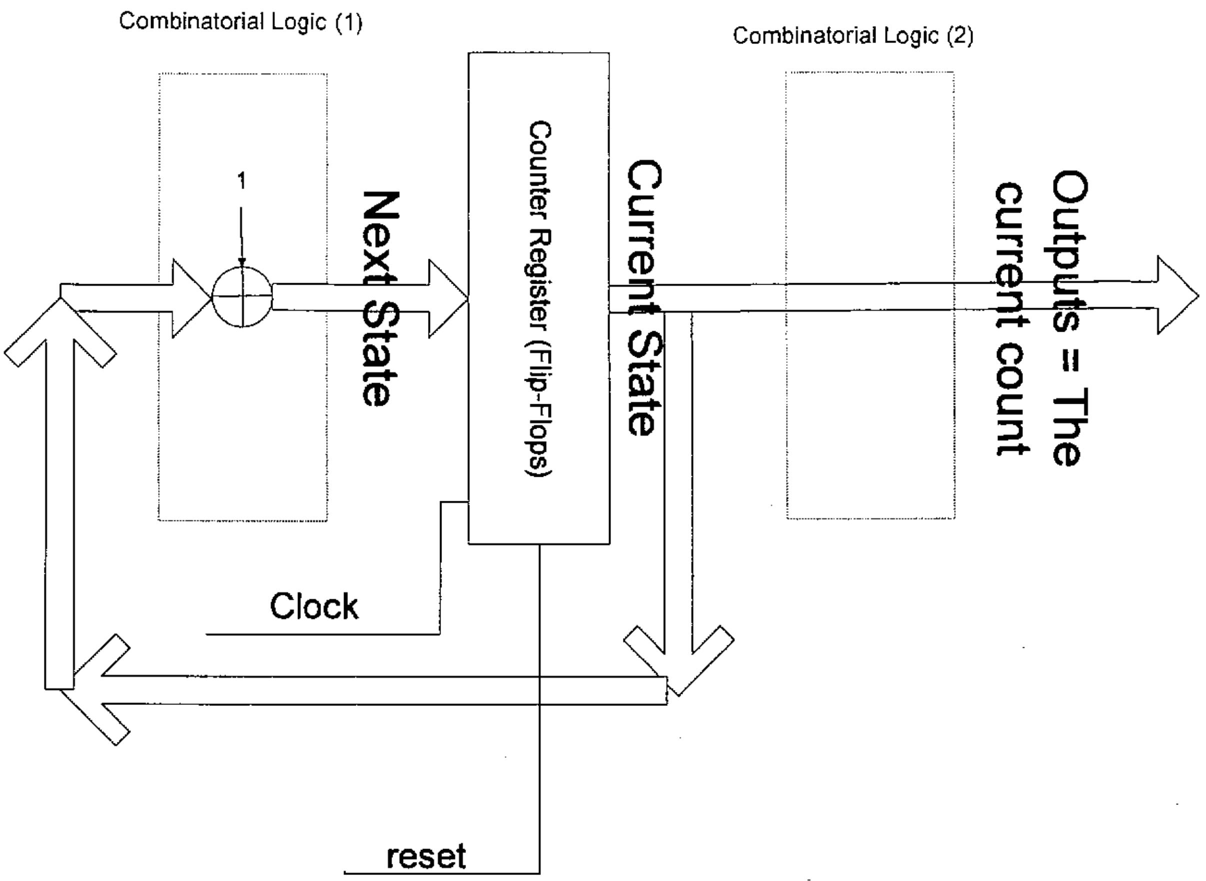
The "core" of the state machine are the State Bits, which are simply flip-flops. The current state of the State Bits is referred to simply as the Current State. ***Every*** clock cycle the State Bits change to the Next State, which then becomes the Current State. The Next State is a combinatorial function (denoted Combinatorial Logic 1) of the inputs and the Current State. The outputs of the state machine are a combinatorial function (denoted Combinatorial Logic 2) of the inputs and the Current State. A reset signal is responsible for setting the state machine to its initial state.

Example: The simplest state machine – a single flip-flop:



The combinatorial logic blocks are the simplest possible – a wire. The next state is the input to the D port of the flip-flop, and the current state is present at the Q port of the flip-flop.

Another example: a counter:



The above examples are important because now you know that even if didn't call it by name – youv'e been using state machines in your code since the first day you programmed in Verilog!!!

Glitches

One fundamental goal of design should be that all signals be glitch free. There are several reasons for avoiding glitches. Among the most important:

1. Edge triggered inputs such as clocks may be falsely triggered by glitches

2. Even if the lines are not edge triggered, they can be affected by glitches. Imagine for example, that a line which is an output enable for a FIFO has glitches on it. If the glitch occurs just before the output of the FIFO is sampled, and the FIFO is fast enough to react to that glitch, then a 3-state output (=garbage) will be sampled.

3.Even in simulation, one canot find all of the glitches and even if one could, that would not necessarily be what would happen in the device because of actual device delays. Thus if one would want to be sure that there were no glitches in the outputs of a chip, one has to design the code so that glitches could not occur because of an *intrinsic* quality of the code.

4.When debugging a card and looking at the signals via a logic analyzer, if one were to see glitches in the signals, the cause would be only noise spikes and not glitches, if the code was written in a glitch-free fashion (how to do that will be explained shortly). Thus, when glitches appear in the signals when seen in a logic analyzer, they can be attributed solely to noise spikes and one need not worry that the code in the chips generated the glitches. This greatly shortens debugging time since when we see a glitch we know for sure it is caused by noise and not by the code, and can take action to combat it.

5.Problems with glitches in internal logic inside an FPGA (i.e. they do not go to the pins of the FPGA) are virtually impossible to observe, and so unexplained "voodoo" problems results, and you have no idea why

Consider the following task:

Mission: Design a module whose input is a clock, and its output is a clock pulse every 4th clock.

Solution:

```
module pulse_every_4_clks(in_clk,out_clk,reset);

input in_clk;
output out_clk;
input reset;

wire in_clk;
wire out_clk;
wire reset;

reg[1:0] count_reg;

always @(posedge in_clk or negedge reset)
begin
     if (!reset)
      count_reg <= 2'h0;
     else
       count_reg <= count_reg + 2'h1;
end

assign out_clk = (count_reg == 2'h3);

endmodule
```

What's wrong with the above module? This module produced a pulse every 4th clk, but it also produced a glitch between every second and third clock. Why?

Glitch-Free Solution:

```
module pulse_every_4_clks(in_clk,out_clk,reset);

input inclk;
output outclk;
input reset;

wire in_clk;
reg out_clk;
wire reset;

reg[1:0] count_reg;

always @(posedge in_clk or negedge reset)
begin
     if (!reset)
      count_reg <= 2'h0;
     else
       count_reg <= count_reg + 2'h1;
end

always @(posedge in_clk)
     out_clk <= (count_reg == 2'h2);

endmodule
```

Note that this problem of glitches will be present in every counter – that is you cannot have an output combinatorially dependent on the output of a counter without fear of glitches. This will be seen shortly to be a particular case of a more general problem.

Note also that

```
out_clk <= (count_reg == 2'h2);
```

and not

```
out_clk <= (count_reg == 2'h3);
```

This is because now there is an additional delay of one clock that is incurred because out_clk is registered, so to preserve exactly the same response the change is needed.

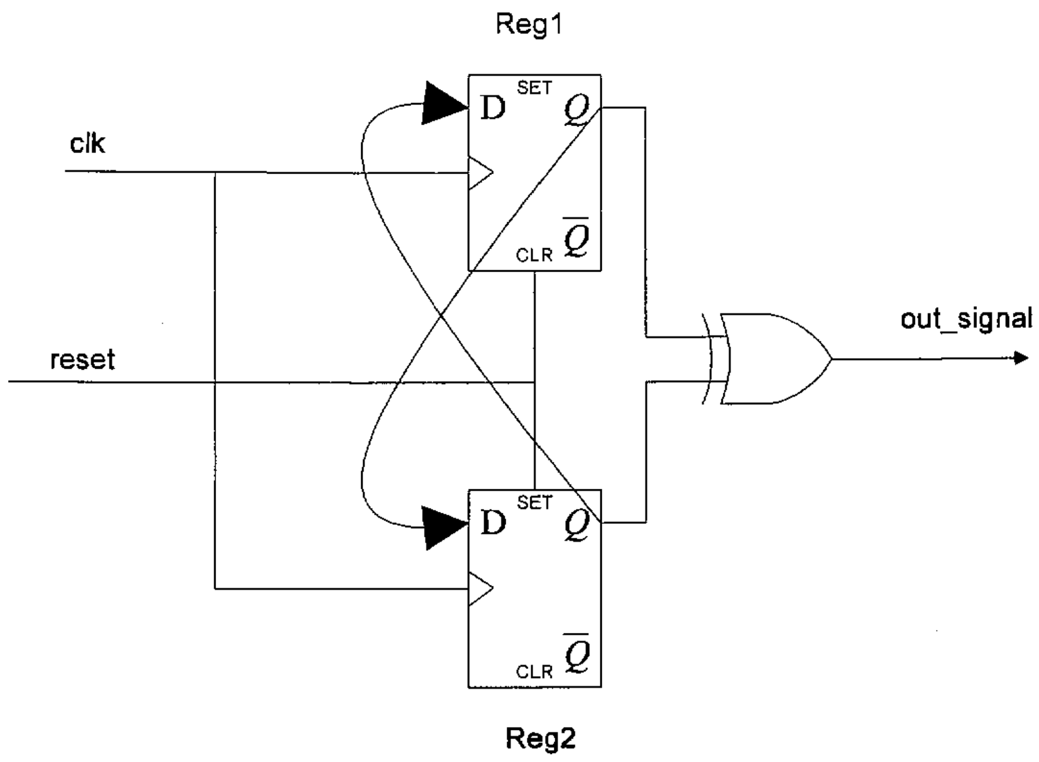
Challenge: Is it possible to build a module that will generate a glitch on EVERY clock cycle?

Answer: Yes. Presenting the "ideal glitch generating machine"!

```
module glitch_generator(clk,reset,out_signal);

input clk,reset;
output out_signal;

reg reg1,reg2;

always @(posedge clk or negedge reset)
begin
     if (!reset)
     begin
          reg1 <= 1'h0;
          reg2 <= 1'h1;
     end else
     begin
          reg1 <= reg2;
          reg2 <= reg1;
     end
end

assign out_signal = reg1 ^ reg2; // note that ^ means xor

endmodule
```

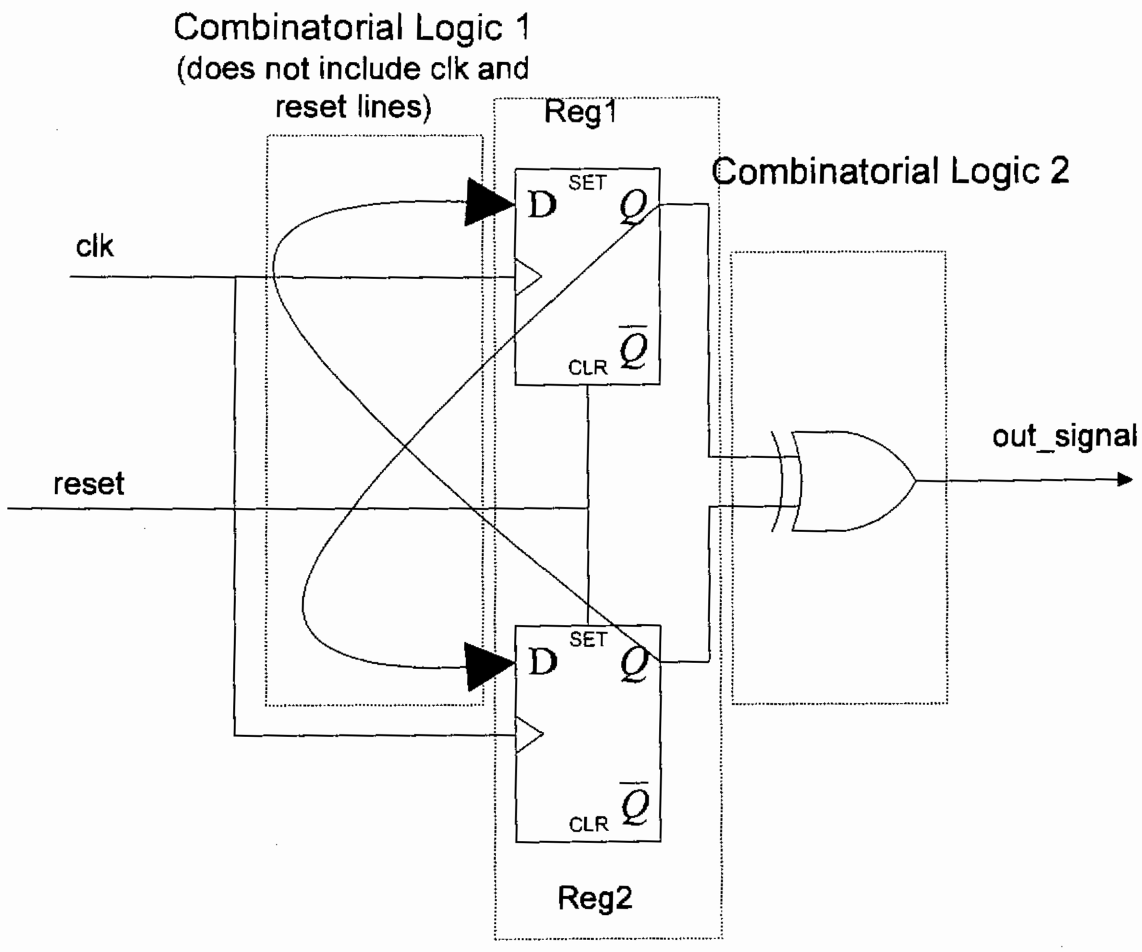
The output should be constant at 1, but there is a glitch every clock cycle!!!!

Why does this happen? It is instructive to see how the above module is synthesized:



The registers go between the state reg1=0, reg2=1 to the state reg1=1, reg2 = 0, and then back again, indefinitely. The glitch occurs because we live in a real world, the registers never switch at EXACTLY the same time, and there is ALWAYS an instant, however small, that the registers are at a state where they are both 1 or they are both 0. This causes a 0 to appear momentarily at out_signal – a glitch!!!

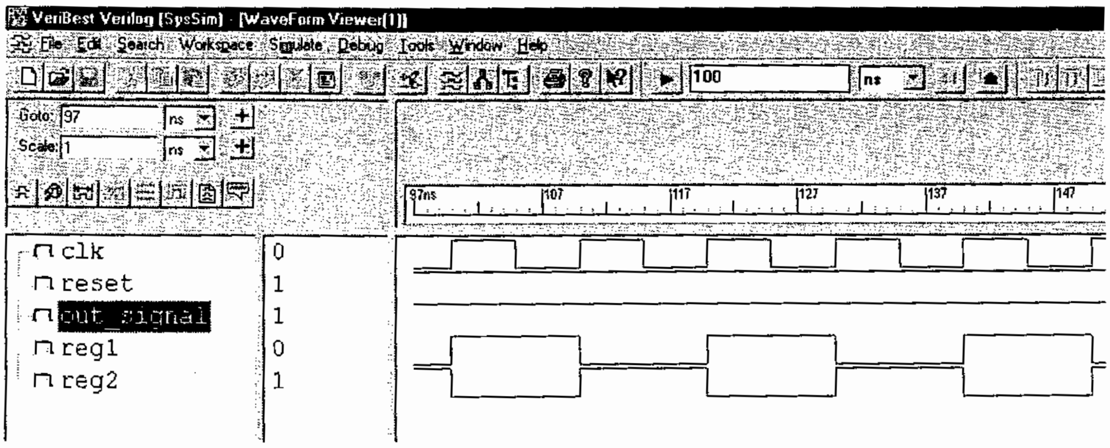
Now lets take a look at the above module in the context of it being a state machine:



State Bits

As can be seen that there occurs a glitch on EVERY clock edge for a block of Combinatorial Logic 2 which contains a SINGLE gate, and there are only TWO state bits!!! Just imagine the number of glitches (indeed multiple glitches!!!) that will result with more state bits and more complex combinatorial logic! The glitches (or multiple glitches) that appear at each output as the state machine changes states on each clock edge, may severely interfere with the system's operation.

The task of identifying all the possible glitches in a complex system is daunting if not impossible. Furthermore, even if one could detect all glitches in simulations (which is very unlikely), still the behavior of the device may differ wildly because of actual device delays, or different routing in different compilations. Infact, simulators do an extremely lousy job of detecting glitches. Look for example at the following simulation of the above module:


VeriBest Verilog [SysSim] - [WaveForm Viewer(1)]
File Edit Search Workspace Simulate Debug Tools Window Help
100
ns
Goto: 97 ns
Scale: 1 ns
97ns
107
117
127
137
147
clk 0
reset 1
out_signal 1
reg1 0
reg2 1

According to the simulator, there are no glitches in out_signal!!!! And this is for the simplest possible output logic – a single gate!!! It is evident that trying to detect glitches in simulation is futile!!!

Thus in order to completely prevent glitches, one has to design his code so that the design will be glitch free as a result of som *intrinsic* quality of the design method. Such a method is now discussed.

Analysis: Why did the glitch occur?

The glitch occurred as a result of the fact that the output was not supposed to change state when the state bits changed, and since the state bits did not change at EXACTLY the same time, a momentary glitch appeared at the output.

From this we may draw a more general conclusion that will help us formulate a design method that eliminates glitches:

If:

> A glitch occurs when the same output is generated by different values of the state bits

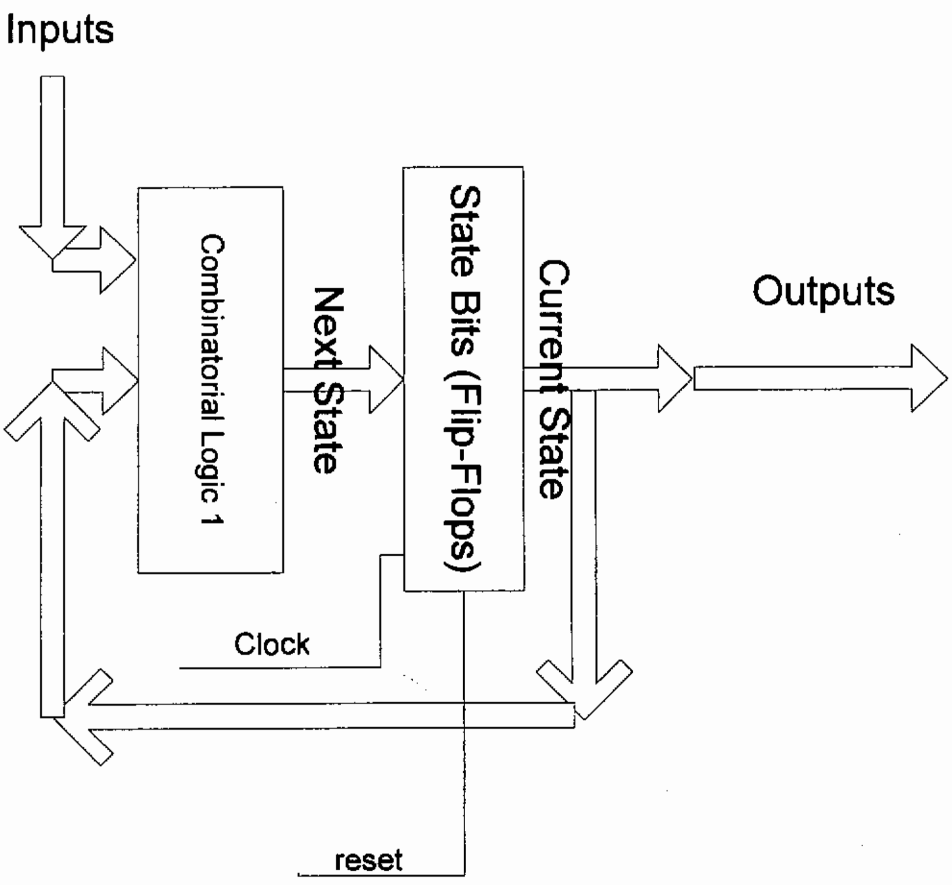
Then:

> To avoid glitches it follows that the outputs must be unique for each combination of the state bits' values

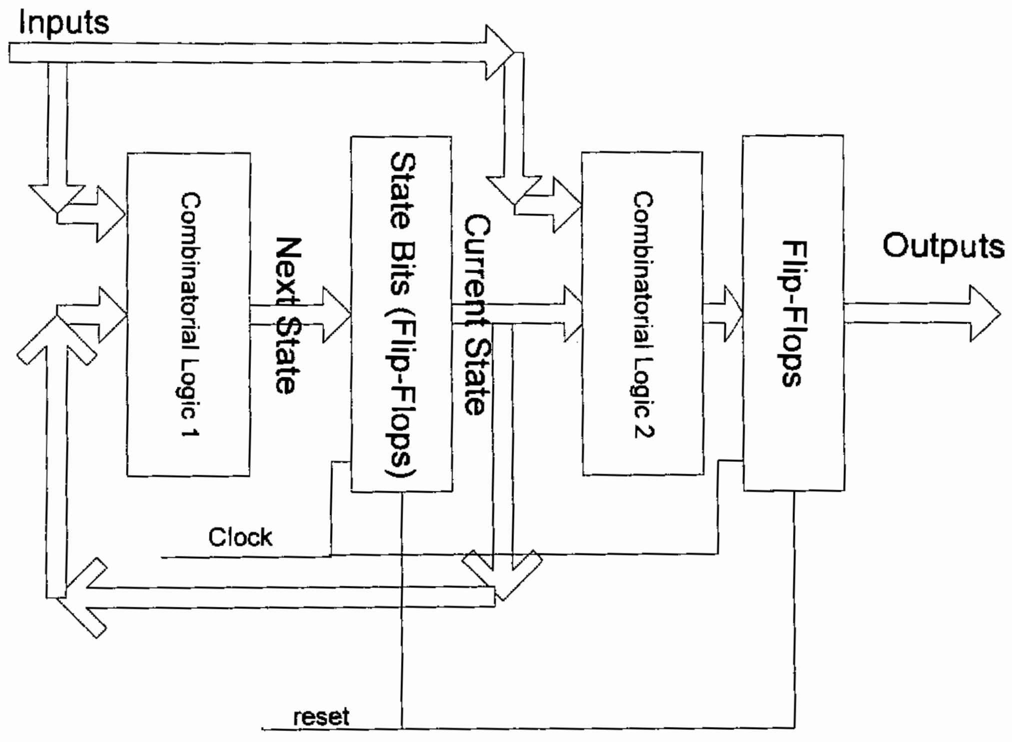
The above conclusion will shortly be seen to define specific conditions that dictate what can be present in the block labeled Combinatorial Logic 2.

Solution 1: Eliminate Combinatorial Logic 2!!! (=make it wires only!)

This can be seen to be EQUIVALENT to registering all of the outputs from the state machine:



Exercise: Show that Eliminating Combinatorial Logic 2 is EQUIVALENT to registering all the outputs, as shown here:



Hint: Add the output flip-flops to the State Bits and Combinatorial Logic 2 to Combinatorial Logic 1, and create a new state machine that does not have Combinatorial Logic 2 or output flip-flops

Indeed, registering all of the outputs is the most common method to eliminate glitches. It does, however, have several drawbacks:

1.There is a 1 clock delay added to the outputs. Apart from the delay itself, this often adds confusion to the designer's understanding of the state machine operation

2.There can be no direct dependence of outputs on inputs

3. Each addition of outputs needs a corresponding number of registers to be added. This causes an inflation (and sometimes an explosion) of registers needed for the implementation

Nonetheless, registering the outputs is often the fastest "quick-fix" for glitchy state machines.

A more efficient method defines the limitations on Combinatorial Logic 2 as follows, in a matter that eliminates the need to register every output:
An output can be:

1. Driven directly from a state bit

2. Driven by the NOT of a state bit

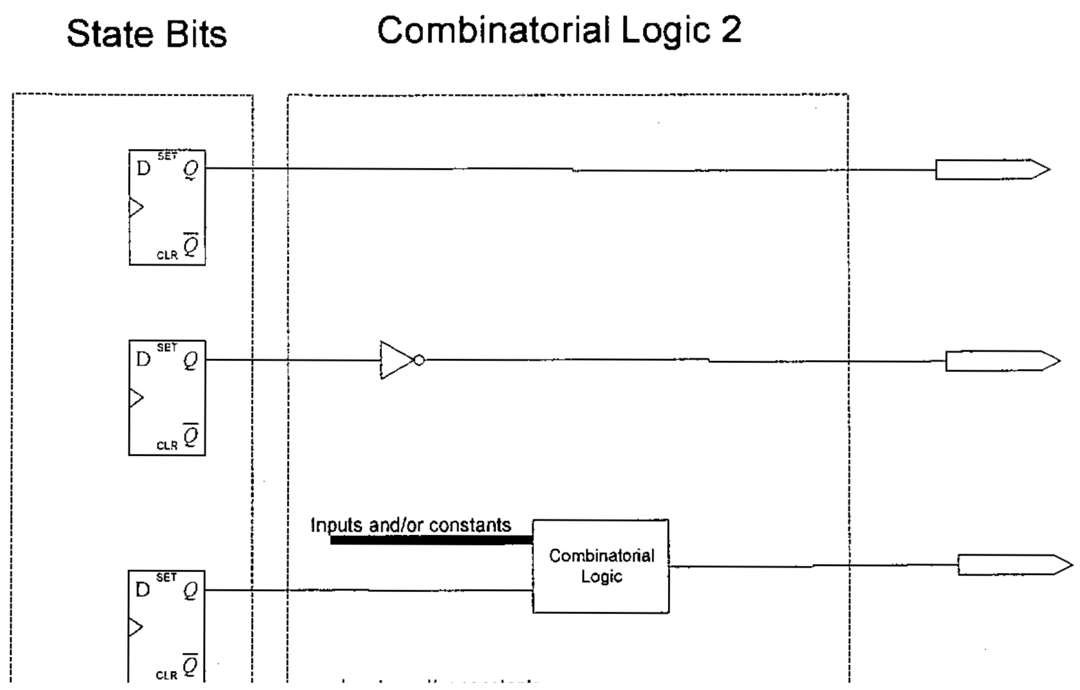
3. Driven by combinatorial logic that depends on a **single** state bit, but can depend on other inputs or constants

Explanation:

It is obvious why cases (1) and (2) are glitch free.

In case (3) glitches are not generated as a result of the state bits, because at least 2 are needed to generate a glitch. However glitches may be generated if the inputs change and the resulting output is the same, just as glitches are generated by the state bits. The meaning in case (3) is generally for constant inputs or quasi-constant inputs (that is - constant during the operation of the state machine - e.g. the arguments passed to the machine from a higher level module which remain constant throughout the state machine's operation).

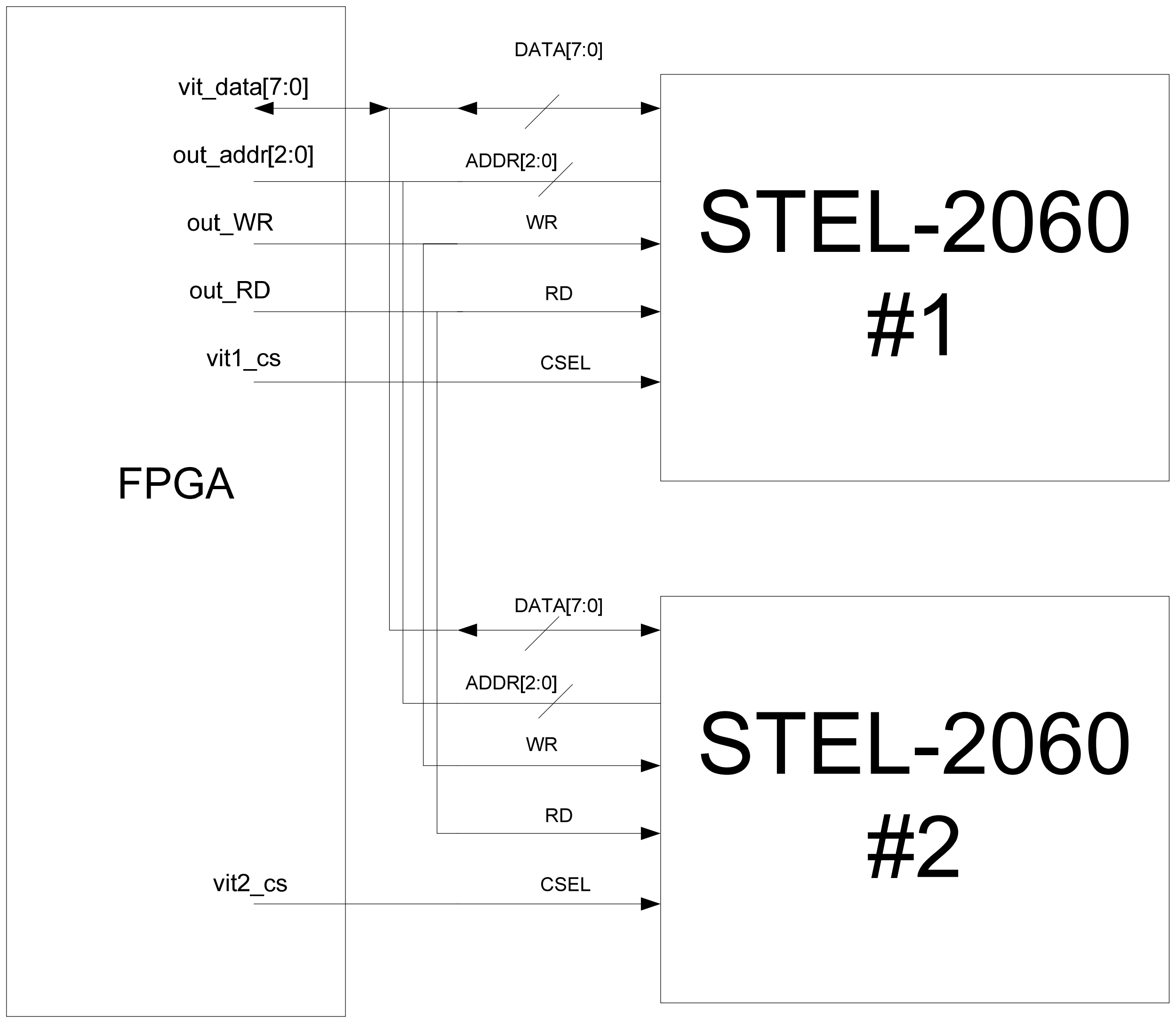
The allowed constructs are shown in the following page:



<u>There can be many variations on the theme, but it is important to always keep in mind why glitches are caused and how to avoid them!!!</u>

Mission: to interface with the STEL-2060 Viterbi Decoder chip in order to write and read from it according to the datasheet in the next page.

(Viterbi coding/decoding is used in many modern communications systems, a fascinating topic to study)



There are two Viterbi chips on the card connected to the same data and address bus, each with an individual chip select.

A single module is required to be designed that will interface with the two Viterbis and write to them or read from them, according to arguments passed to it. If the operation is a read, it will return the value read to the calling (higher level) module.

STEL-2060C/CR

Data Sheet

STEL-2060C/CR

45 Mbps

Viterbi Decoder

intel.

FEATURES

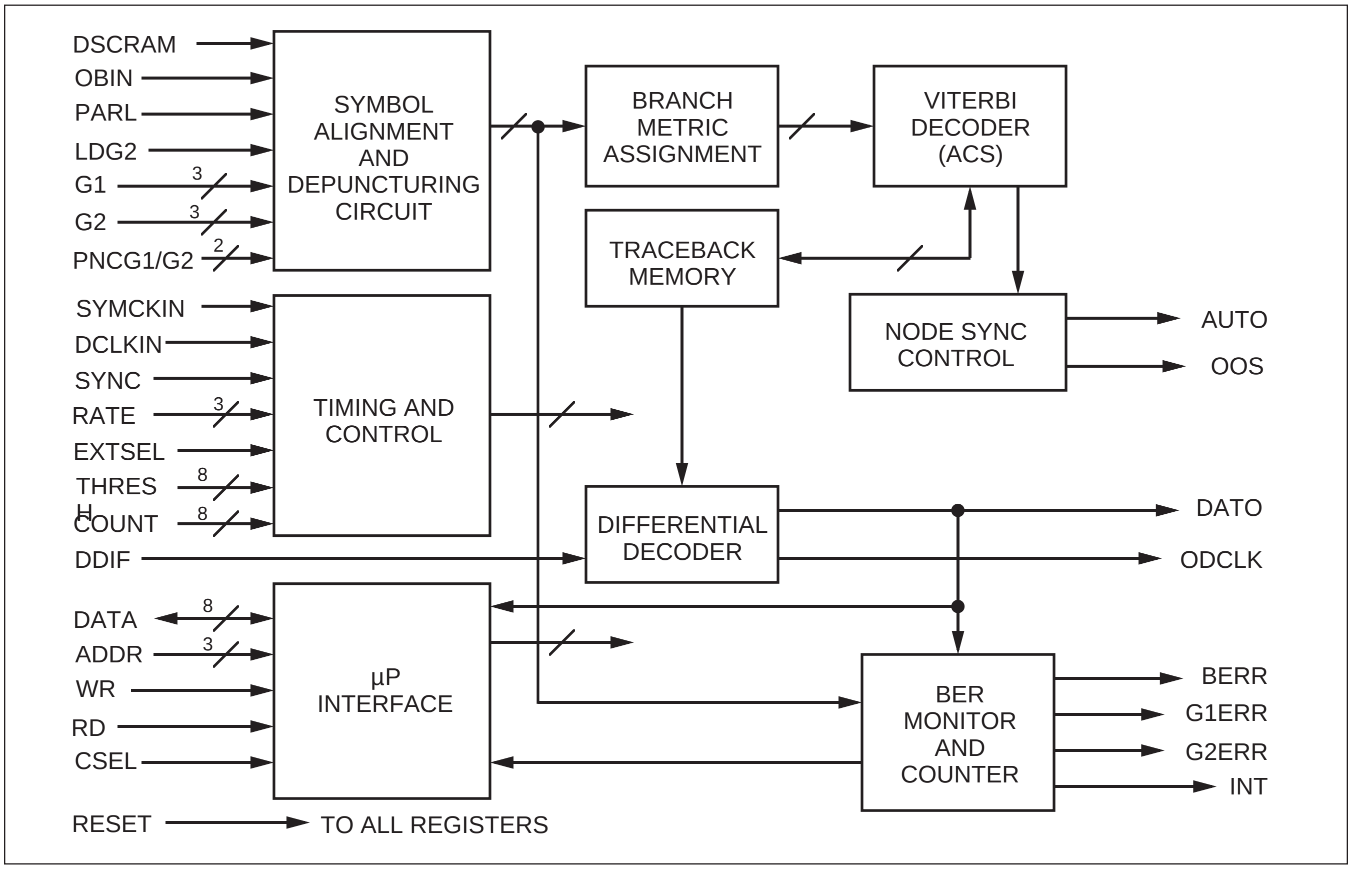
- **45 Mbps Operating Rate**
- **Constraint Length K = 7** **$G_1 = 171_8$ $G_2 = 133_8$**
- **Multiple Rates: Rate $^1/_2$ as well as Punctured codes at Rates $^2/_3$ through $^7/_8$**
- **Internal Depuncturing Capability at Rates $^2/_3$, $^3/_4$ and $^7/_8$**
- **Multiple Devices can be Multiplexed to Give Higher Data Rates**
- **Optimized Interface to Operate with BPSK and QPSK Demodulators**
- **Auto Node Sync Capability**
- **Differential Decoder**
- **"Invert G2" Descrambler**
- **Internal BER Monitor and BER Measurement Circuit**
- **5.2 dB Coding Gain @10^{-5} BER (R = $^1/_2$)**
- **100-pin PQFP Package**
- **0.5 Micron CMOS Technology**

FUNCTIONAL DESCRIPTION

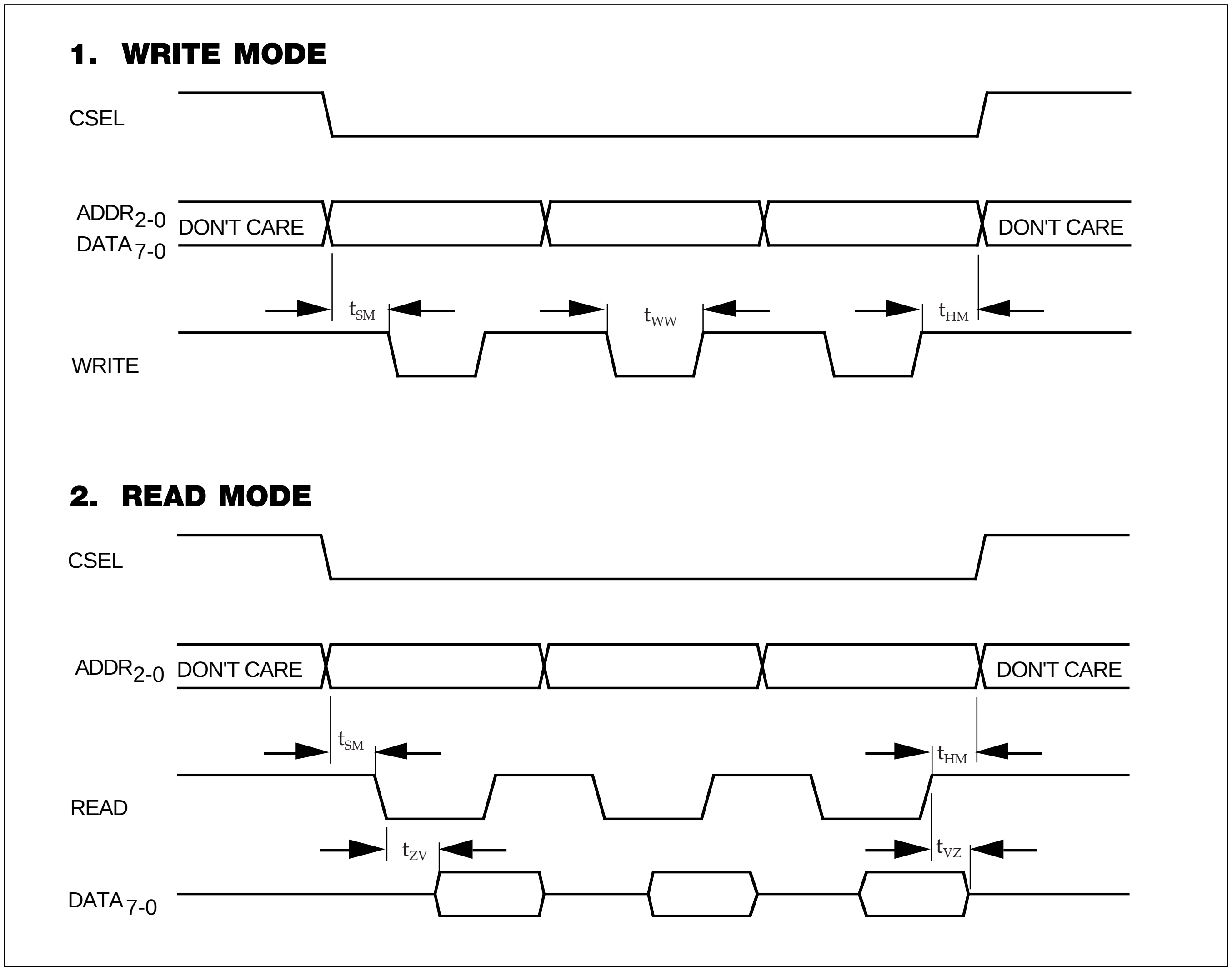
Convolutional encoding and Viterbi decoding are used to provide forward error correction (FEC) which improves digital communication performance over a noisy link. The STEL-2060C is a specialized product designed to perform this specific communications related function. At the encoder a stream of symbols is created which introduces a high degree of redundancy. This enables accurate decoding of the information despite a high symbol error rate resulting from an impaired communications link.

The STEL-2060C contains a K = 7 Viterbi Decoder. The data inputs can be in offset binary or offset signed-magnitude formats, with 3-bit soft decision. Auto node sync is provided for applications where symbol uncertainty can occur. Rate $^2/_3$, $^3/_4$, $^4/_5$, $^5/_6$, $^6/_7$ and $^7/_8$ punctured signals can be decoded, as well as non-punctured, Rate $^1/_2$, signals. The polynomials and puncturing patterns used are industry standards. Depuncturing logic is incorporated into the decoder to provide automatic depuncturing of received data at rates $^2/_3$, $^3/_4$ and $^7/_8$ when the puncturing patterns supported by the device are used. A BER monitor is also provided in the device, along with a circuit for computing the mean value of the BER over an extended period. These circuits operate with punctured codes as well as unpunctured. The STEL-2060C incorporates a descrambler for signals scrambled with the "Invert G2" algorithm. (With this method the G2 symbols are logically inverted at the encoder. This provides a very effective level of scrambling for the purpose of avoiding long strings of ones or zeroes in the transmitted signal using BPSK modulation.)

BLOCK DIAGRAM



MICROPROCESSOR INTERFACE TIMING



A.C. CHARACTERISTICS

(Operating Conditions: V_{DD} = 5.0 ±5% volts, T_a = 0° to 70° C)

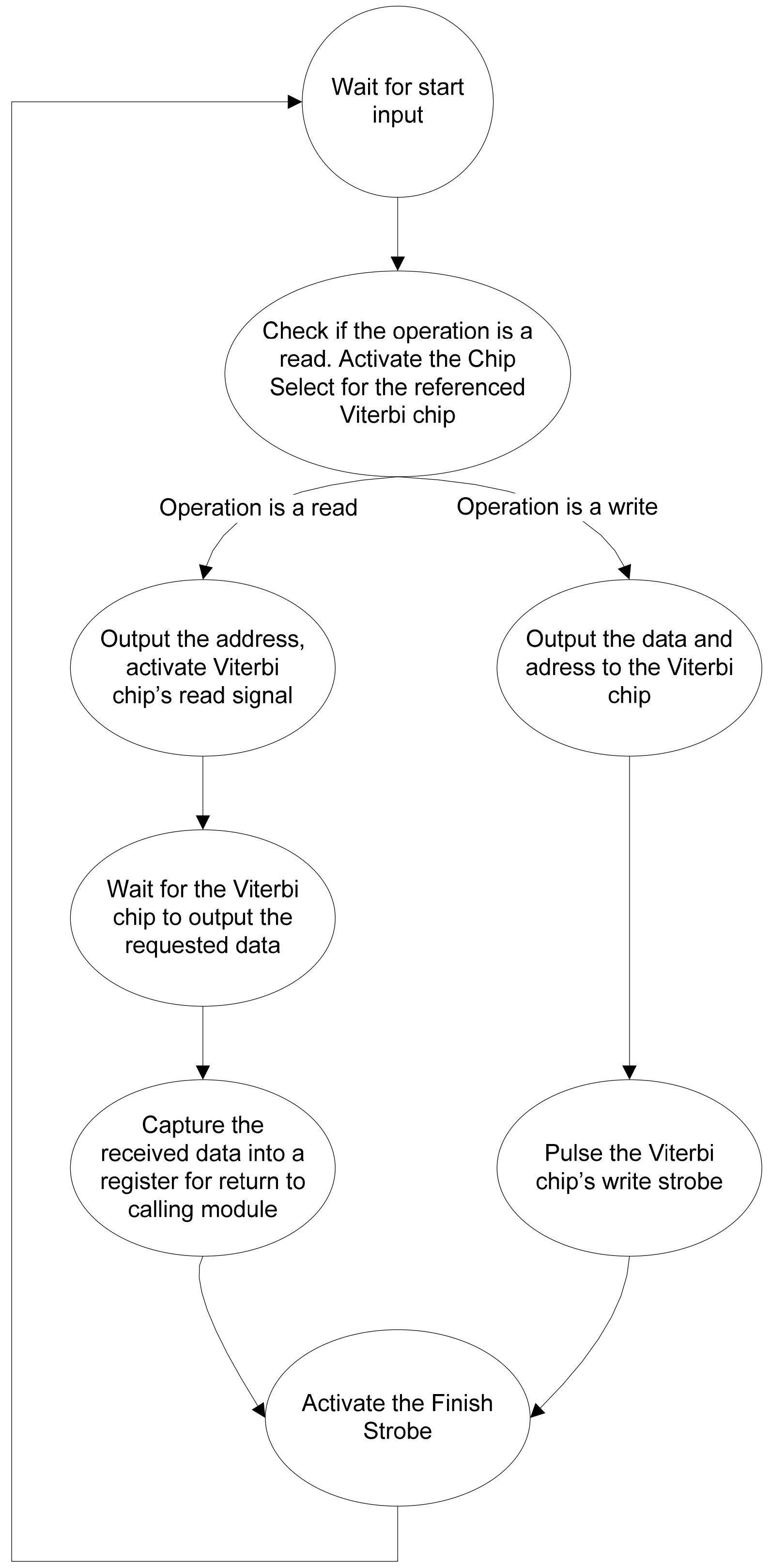
Symbol	Parameter	Min.	Max.	Units
t_{SM}	**CSEL**, **ADDR$_{2-0}$** or **DATA$_{7-0}$** to **WRITE** or **READ** Setup	10		nsecs.
t_{HM}	**CSEL**, **ADDR$_{2-0}$** or **DATA$_{7-0}$** to **WRITE** or **READ** Hold	5		nsecs.
t_{WW}	**WRITE** Pulse width	5		nsecs.
t_{ZV}	**READ** (low) to **DATA$_{7-0}$** Valid		10	nsecs.
t_{VZ}	**READ** (high) to **DATA$_{7-0}$** High-Impedance		10	nsecs.

Step 1: Mission statement, study the datasheet to understand the chip, its pin connections and its timing waveforms (above)

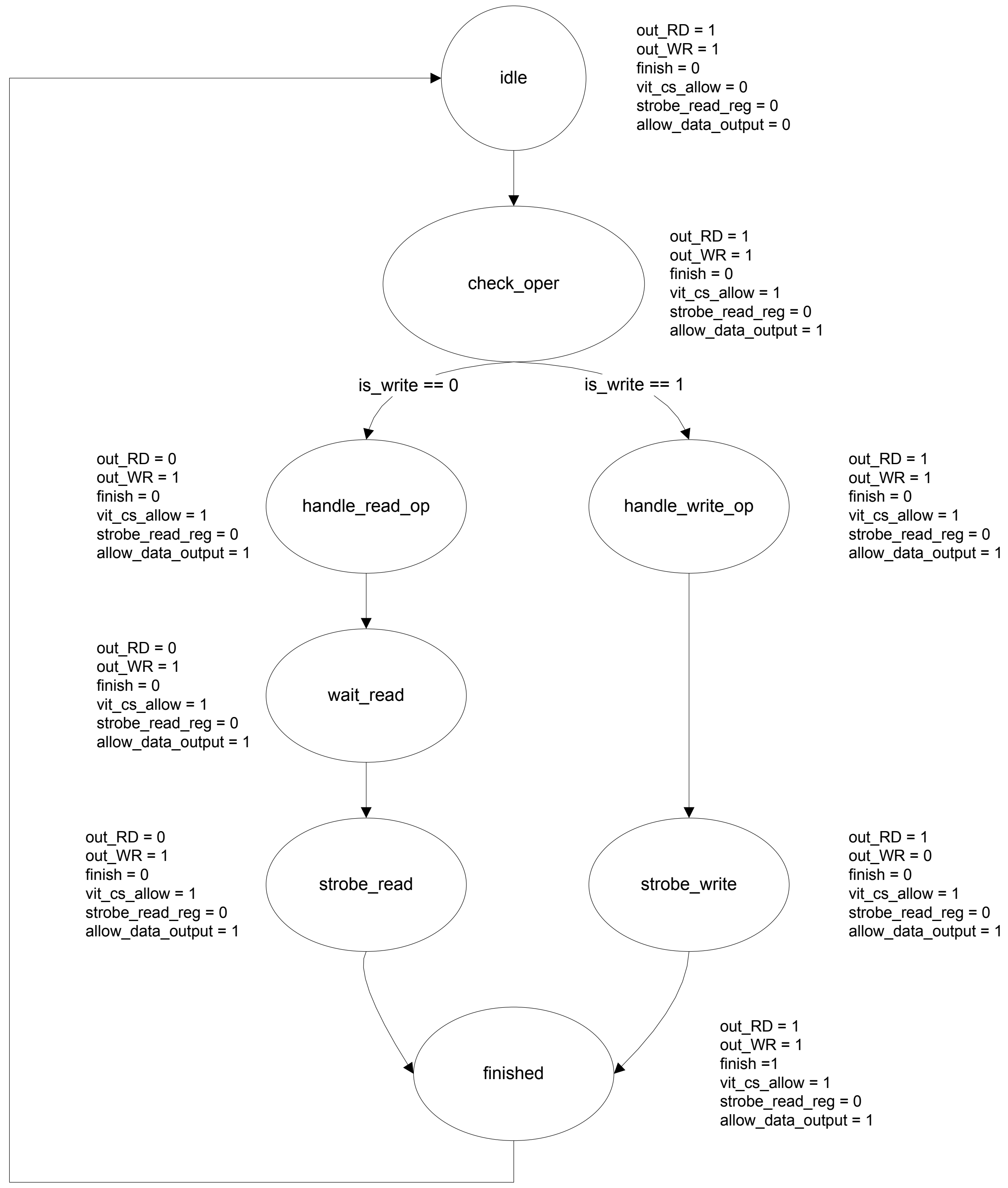
Step 2: Define the inputs and outputs of the module and their purpose

Port Name	Type	Connects To	Purpose	Additional Comments
in_data [2:0]	input	Calling module	The address to be written inside the referenced Viterbi chip	
in_data[7:0]	input	Calling module	The data to be written to the referenced Viterbi chip if the operation is a write	
is_write	input	Calling module	1 if the operation is a write, 0 otherwise	
vit_num	input	Calling module	0 for operation on Viterbi Chip #1 1 for operation on Viterbi Chip #2	
start	input	Calling module	Tells the module to begin operation	
return_data[7:0]	output	Calling module	Returns the data read from the Viterbi chip if the operation was a read	
finish	output	Calling module	Tells the calling module that the operation has been completed	
vit1_cs	output	Viterbi chip #1	Viterbi #1 Chip Select	Active Low
vit2_cs	output	Viterbi chip #2	Viterbi #2 Chip Select	Active Low
out_addr[2:0]	output	Both Viterbi chips	Connects to the Viterbi chips' ADDR pins.	
vit_data[7:0]	inout (bidirectional)	Both Viterbi chips	Connects to the Viterbi chips' DATA pins. Is an output for write, an input for read.	
out_RD	output	Both Viterbi chips	Connects to the RD pin on the Viterbi chips	Active Low
out_WR	output	Both Viterbi chips	Connects to the WR pin on the Viterbi chips	Active Low

<u>Step 3</u>: Draw a flow diagram, in plain English, detailing the operation of the state machine


Wait for start input
Check if the operation is a read. Activate the Chip Select for the referenced Viterbi chip
Operation is a read
Operation is a write
Output the address, activate Viterbi chip's read signal
Output the data and adress to the Viterbi chip
Wait for the Viterbi chip to output the requested data
Capture the received data into a register for return to calling module
Pulse the Viterbi chip's write strobe
Activate the Finish Strobe

Step 4: Draw a flow chart detailing the state transitions and outputs. Decide on which state bits go out to the outputs directly, and which through other logic. Give names to the states and to the state bits:



<u>Step 5</u>: Set the state encodings as follows, adding additional state bits (here bits 7-10) to make the states unique (bit 6 is always 0 and is a placeholder for future expansion)

```
                                    // 0987_6543210
parameter idle                = 11'b0000_0000000;
parameter check_oper          = 11'b0010_0101000;
parameter handle_write_op     = 11'b0011_0101000;
parameter strobe_write        = 11'b0100_0101010;
parameter handle_read_op      = 11'b0101_0101001;
parameter wait_read           = 11'b0110_0101001;
parameter strobe_read         = 11'b0111_0111001;
parameter finished            = 11'b1000_0101100;

assign out_RD            = ~state[0];
assign out_WR            = ~state[1];
assign finish            = state[2];
assign vit_cs_allow      = state[3];
assign strobe_read_reg   = state[4];
assign allow_data_output = state[5];
```

Step 6: Write the output equations for the outputs that are not driven directly by a state bit, paying close attention that the constructs are those allowed that do not cause glitches!!!

```
assign vit1_cs = !((vit_num == vit1_code) && (vit_cs_allow));
assign vit2_cs = !((vit_num == vit2_code) && (vit_cs_allow));

assign out_addr = in_addr;

assign vit_data = (allow_data_output && is_write) ? in_data[7:0] : 8'bz;

assign return_data = read_reg;

always_ff @(posedge strobe_read_reg or negedge reset_all)
begin
       if (~reset_all)
         read_reg <= 7'b0;
       else
         read_reg <= vit_data[7:0];
end
```

Note that `strobe_read_reg` is strictly speaking an output of the state machine directly driven by a state bit, but it is the clock of another register that captures the data received from the Viterbi chip. What could have happened if `strobe_read_reg` had glitches?

Step 7: Write the state transition equations

```
always_ff @(posedge clk or negedge reset_all)
begin
     if (~reset_all)
           state <= idle;
      else
           case(state) /* synthesis full_case */
           idle :begin
                      if (start) state <= check_oper;
                  end
           check_oper : begin
                             if (is_write)
                                   state <= handle_write_op;
                                 else
                                   state <= handle_read_op;

                           end

           handle_write_op: state <= strobe_write;

           strobe_write: state <= finished;

           handle_read_op: state <= wait_read;

           wait_read : state <= strobe_read;

           strobe_read: state <= finished;

           finished : state <= idle;

           endcase
end
```

```
module viterbi_ctrl  (
                      clk,
                      reset_all,
                      in_addr,
                      in_data,
                      is_write,
                      start,
                      vit_num,
                      vit1_cs,
                      vit2_cs,
                      out_addr,
                      vit_data,
                      out_RD,
                      out_WR,
                      return_data,
                      finish
                    );

input clk, reset_all;
input[7:0] in_data;
input[2:0] in_addr;
input is_write;
input start;
input vit_num;

output vit1_cs;
output vit2_cs;
output[2:0] out_addr;
inout[7:0] vit_data;
output out_RD,out_WR,finish;
output[7:0] return_data;
wire[7:0] return_data;

parameter vit1_code = 1'b0;
parameter vit2_code = 1'b1;

reg[10:0] state;
reg[7:0] read_reg;
wire vit_cs_allow;
wire strobe_read_reg;
wire allow_data_output;
                               // 0987_6543210
parameter idle               = 11'b0000_0000000;
parameter check_oper         = 11'b0010_0101000;
parameter handle_write_op    = 11'b0011_0101000;
parameter strobe_write       = 11'b0100_0101010;
parameter handle_read_op     = 11'b0101_0101001;
parameter wait_read          = 11'b0110_0101001;
parameter strobe_read        = 11'b0111_0111001;
parameter finished           = 11'b1000_0101100;

assign out_RD            = ~state[0];
assign out_WR            = ~state[1];
assign finish            = state[2];
assign vit_cs_allow      = state[3];
assign strobe_read_reg   = state[4];
assign allow_data_output = state[5];

assign vit1_cs = !((vit_num == vit1_code) && (vit_cs_allow));
assign vit2_cs = !((vit_num == vit2_code) && (vit_cs_allow));

assign out_addr = in_addr;

assign vit_data = (allow_data_output && is_write) ? in_data[7:0] : 8'bz;

assign return_data = read_reg;

always_ff @(posedge clk or negedge reset_all)
begin
     if (~reset_all)
         state <= idle;
     else
         case(state) /* synthesis full_case */
         idle :begin
                    if (start) state <= check_oper;
               end
         check_oper : begin
                             if (is_write)
                                  state <= handle_write_op;
                              else
                                 state <= handle_read_op;

                      end

         handle_write_op: state <= strobe_write;

         strobe_write: state <= finished;

         handle_read_op: state <= wait_read;

         wait_read : state <= strobe_read;

         strobe_read: state <= finished;

         finished : state <= idle;

         endcase
end

always_ff @(posedge strobe_read_reg or negedge reset_all)
begin
     if (~reset_all)
       read_reg <= 7'b0;
     else
       read_reg <= vit_data[7:0];
end

endmodule
```

Advantages of the design method:

1.Glitch free!!!

2.Universal design methodology for all state machines

3.Single, verified "always" clause for state transitions – you KNOW how it is synthesized (no asynchronous surprises because of complex always structure), and you can use the "full_case" directive to minimize unneccessary logic

4.Very readable, "clean" code

5. Move from state transition diagram to verilog state transition is immediate

6.You can easily reconstruct a flow diagram from the verilog state transitions

7.From the state encodings you know EXACTLY how the outputs of the state machines behave, and changing them is easy

8.Adding or removing states and outputs is extremely easy, allowing for rapid, safe modification of code

9.Because outputs are encoded directly by state bits, or selected by them, then the 1 clock delay associated with registering the outputs is eliminated

10.Though it might not be immediately apparent, this method of design often causes a noticable and even dramatic reduction in logic required. This is due primarily because the block "Combinatorial Logic 2" is either eliminated or simplified considerably (turned into a MUX which has only 3 combinarotial levels) and there are no output registers where they are not needed, as well as implementing output registers that are needed as an integral part of the state machine encoding.

Additional points:

1. All inputs that influence the state transitions MUST be synchronized to the state machine clock. This has NOTHING TO DO with metastability (though metastability may occur, it is not the main problem but one of its symptoms). Rather, the main problem is that if the inputs are not synchronized they may cause a setup time violation on one or more of the state bits, causing the state machine to move into an illegal state.

2. To avoid too many synchronizations in the connections between modules, all state machines should run on the same clock. It is HIGHLY RECOMMENDED that a global clock used for the state machine clock. This is to avoid any problems that may be caused by skew, and to reduce the routing resources used (if all state machines use the same clock, using a global clock can cause major routing reductions)

3. It is highly recommended that the first state have an encoding of all state bits 0. This is always possible, since you can always use the NOT of a state bit if it turns out that you need a state bit with value 1 at the initial state. FPGAs usually wake up with all Flip-Flops at 0, unless you specifically tell them otherwise (and this is not always possible and sometimes the syntax to do so is vendor-specific). To avoid this complexity, it is highly recommended that the state encodings be adjusted so that the first state has an all 0 encoding.

4. In the design process shown above, the steps of course are not done in strict succession but are sometimes done concurrently, depending on the design goal and the degree of designer proficiency. The steps shown are just a general outline of the thought process around the design.

5. Some design tools think that they are smarter than you and allow themselves to reencode the state machine states during synthesis. Disable this feature in your FPGA design tool suite (e.g. synthesis options in Altera Quartus).

Hierarchical Design and the Start-Finish Protocol

Hierarchical design is one of the fundamental concepts of any design. Hierarchical design has many advantages, among them:

1. Separates code into tractable units

2. Allows different modules to be implemented by different designers

3. Allows modules to be easily re-used

What I am going to add to the concept above is the thought that the interface between modules that contain state machines be standardized, so that you can "call" state machines just like you would call subroutines in software. This simplifies the design process greatly because you don't have to devise ad-hoc methods to interface with submodules.

Consider the following analogy:

- Designing without hierarchy is like designing software without subroutines

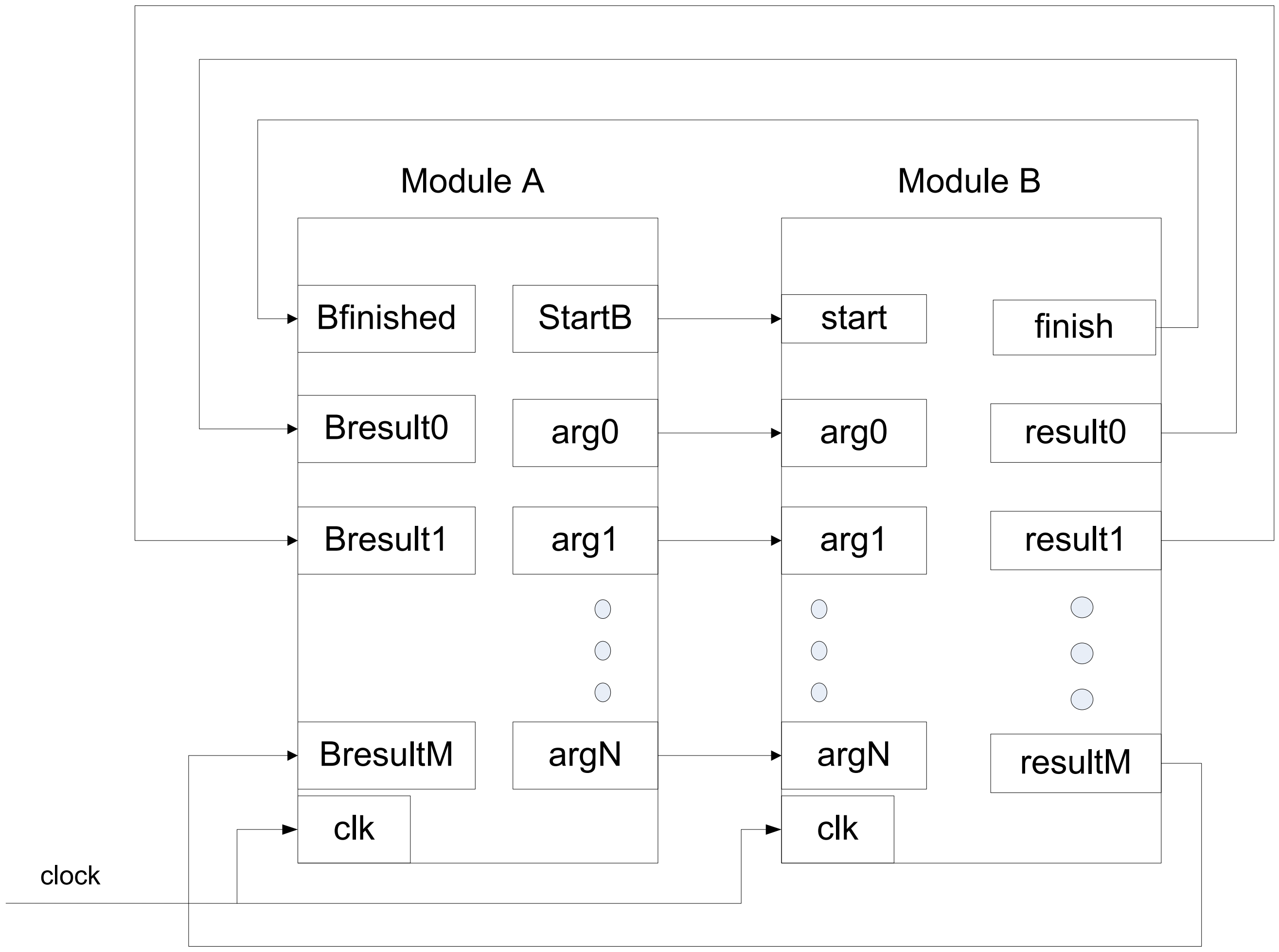
- Designing without a standardized protocol to interface with submodules is to the software designer like re-inventing the way to call and pass arguments to a subroutine for every new subroutine

Standardizing the interface between modules allows:

1. Easier modularization of the design
2. Easier integration of modules designed by different people.
3. Perhaps most important, once an interface has been proven to work reliably, it will work reliably for all new modules. This is much better than debugging a new interface method for every new module.
4. As will be seen, a smart protocol will allow the same module (="procedure") to be "called" by different top-level modules, thus avoiding needless logic replication.

Suggestion: The Start-Finish Protocol:

Let A be a module that calls module B, with arguments arg0,arg1 ,..., argN, and module B returns results as result0, result1, ... , resultM.
The connection of the modules will be as follows:



The operation is as follows:

1. Module A calls Module B in much the same way that a procedure is called in software: It gives module B arguments via B's inputs, and gives a "start" to the start input of B.

2. Module B is by default in a state labeled "idle".

3. When B receives a pulse from its "start" input, it starts operation according to the arguments present at its inputs.

4. When module B finishes its job, it sends a pulse in its "finish" output.

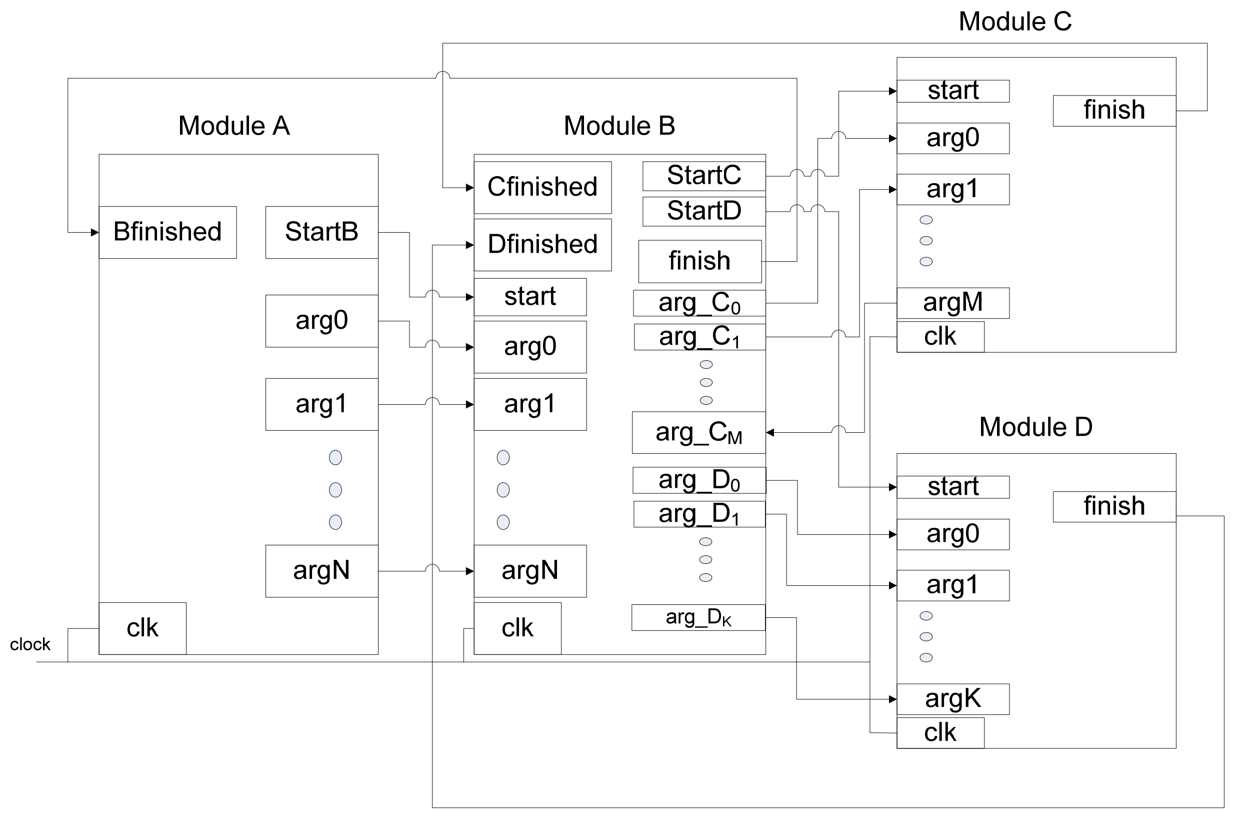
5. While module B proceeds through steps (2),(3) and (4), module A is in a waiting state. When module A senses that module B is asserting its "finish" output, module A knows that module B has finished, and module A continues to its next operation.

Specific implementation issues:

- Both A and B operate using the same clock. Otherwise it would be necessary to synchronize the Start and Finish signals to the receiving module. If modules A and B have to use different clocks, module B must synchronize the Start pulse to conform to its clock (like "trapping an asynchronous signal"), and A must do the same for B's finish signal.

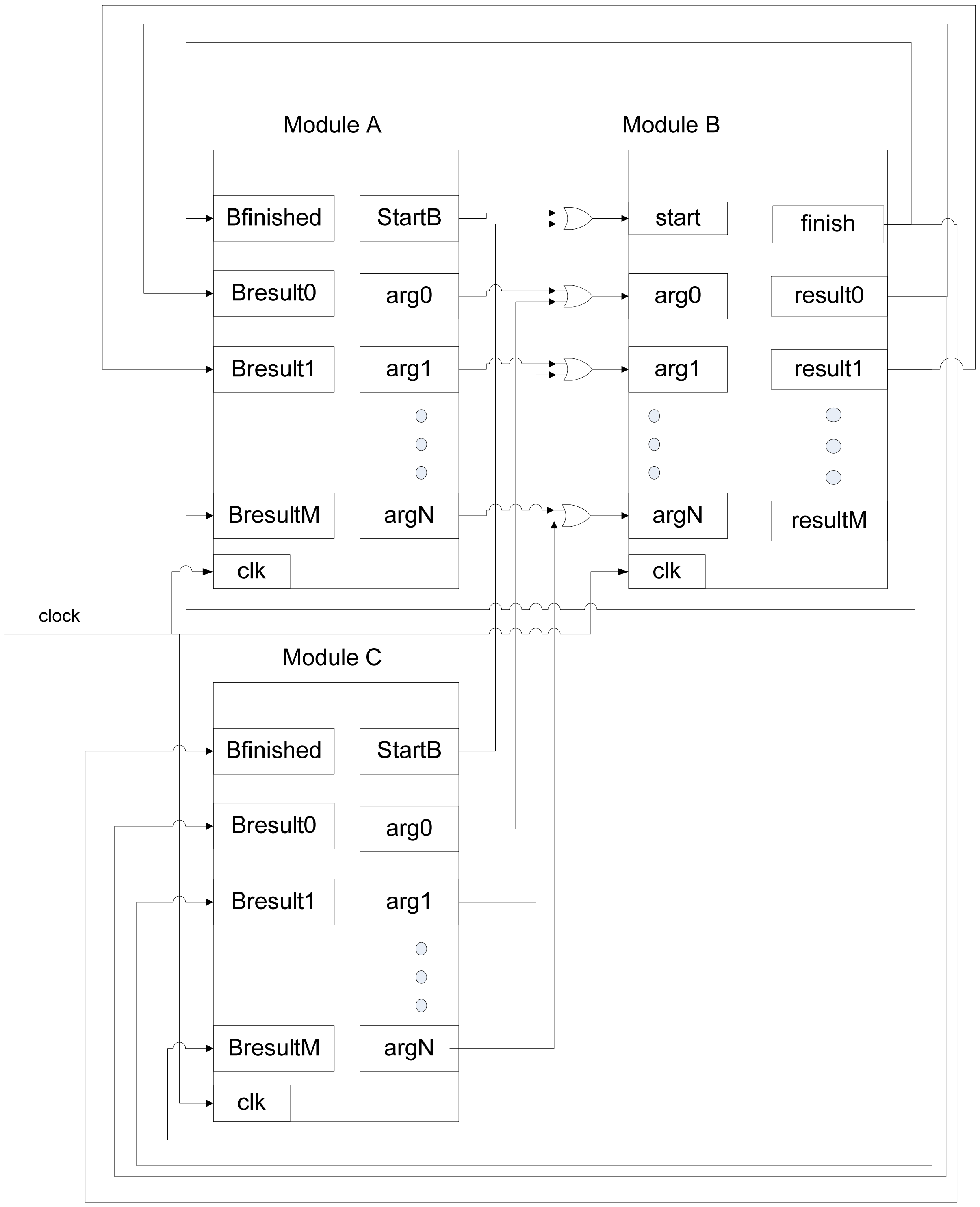
- Inputs to module B must remain valid from when start is asserted by the calling module (=module A) until finish is asserted by the called module (=module B). This is done to minimize flip-flop usage, which would have been required to register the inputs. Furthermore, since module A is in a waiting state ("busy waiting") it does not require any additional logic complexity to hold A's outputs during this waiting period.

This protocol easily allows hierarchical calling, and calling of multiple modules, as seen here. Note that for clarity the paths of returning results to the calling modules have been omitted from the following diagram.



The protocol outlined allows calling modules like you would call procedures in software. Just like in software, each procedure can call other procedures, and those procedures can call other procedures, etcetera. So this method of design helps the hardware designer because he/she can break down complex algorithms into subprocedures and use them in an analogous way to how a software programmer uses procedures.

The start-finish protocol allows the use of one module by many other top-level modules, thus avoiding needless logic replication. This is done simply by OR-ing the start signals of the calling modules and the arguments. This is shown in the following schematic:



The restrictions on calling a module from multiple modules are:

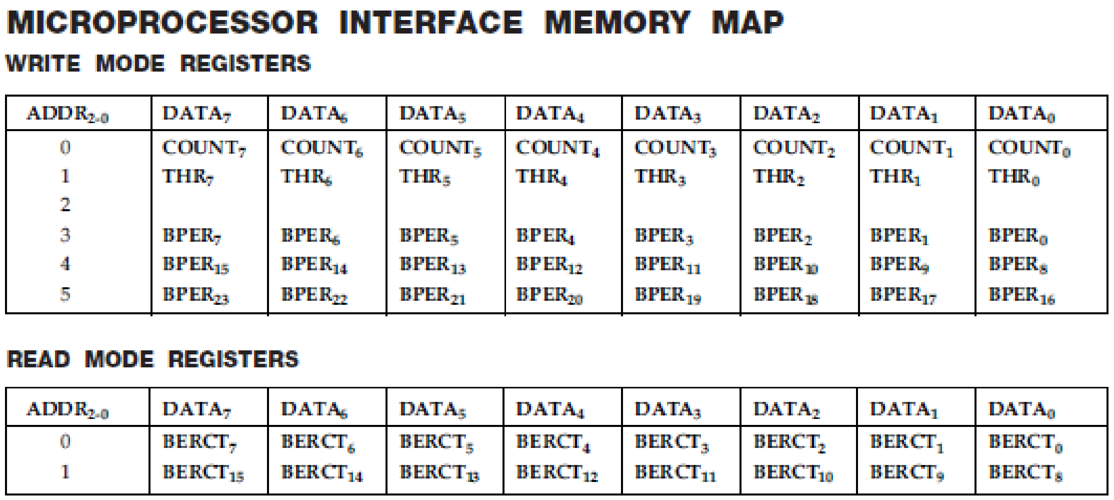
1. There is never a time that both A and C attempt to call module B at the same time (it is the designer's responsibility to ensure that)

2. When a A or C are not in the middle of calling module B, their arguments and start output are at logical 0, so as not to interfere with any other module from calling module B.

There is the issue of where the results are registered (module B or in each of the modules A and C?). In general it is more economical to register the result in B, because this uses half the flip-flops in comparison to latching the results in both modules A and C. In the case that module B latches the results, it is assumed that the results retain their correct value until B is called again (i.e. until it receives another start pulse).

Example: calling the `viterbi_ctrl` module

In order to illustrate the Start-Finish protocol, let's design a few modules to read and write certain registers in the Viterbi chips according to the address map outlined in its datasheet.



MICROPROCESSOR INTERFACE MEMORY MAP

WRITE MODE REGISTERS

$ADDR_{2-0}$	$DATA_7$	$DATA_6$	$DATA_5$	$DATA_4$	$DATA_3$	$DATA_2$	$DATA_1$	$DATA_0$
0	$COUNT_7$	$COUNT_6$	$COUNT_5$	$COUNT_4$	$COUNT_3$	$COUNT_2$	$COUNT_1$	$COUNT_0$
1	THR_7	THR_6	THR_5	THR_4	THR_3	THR_2	THR_1	THR_0
2								
3	$BPER_7$	$BPER_6$	$BPER_5$	$BPER_4$	$BPER_3$	$BPER_2$	$BPER_1$	$BPER_0$
4	$BPER_{15}$	$BPER_{14}$	$BPER_{13}$	$BPER_{12}$	$BPER_{11}$	$BPER_{10}$	$BPER_9$	$BPER_8$
5	$BPER_{23}$	$BPER_{22}$	$BPER_{21}$	$BPER_{20}$	$BPER_{19}$	$BPER_{18}$	$BPER_{17}$	$BPER_{16}$

READ MODE REGISTERS

$ADDR_{2-0}$	$DATA_7$	$DATA_6$	$DATA_5$	$DATA_4$	$DATA_3$	$DATA_2$	$DATA_1$	$DATA_0$
0	$BERCT_7$	$BERCT_6$	$BERCT_5$	$BERCT_4$	$BERCT_3$	$BERCT_2$	$BERCT_1$	$BERCT_0$
1	$BERCT_{15}$	$BERCT_{14}$	$BERCT_{13}$	$BERCT_{12}$	$BERCT_{11}$	$BERCT_{10}$	$BERCT_9$	$BERCT_8$

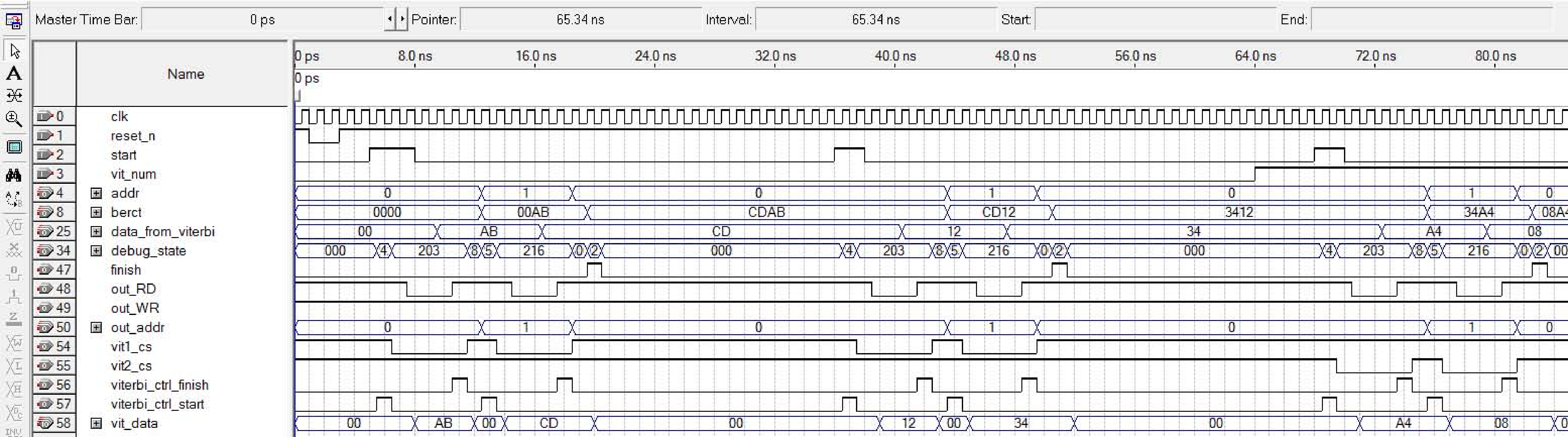
The first module we will design is the `read_berct` module, which is an FSM that will read the Bit Error Rate Count (BERCT) from the read-mode registers and assemble it as 16-bit word. It will call `viterbi_ctrl` to do the heavy lifting of reading from the chip. We shall write a module `test_read_berct` to connect `read_berct` to `viterbi_ctrl` and test the functionality of the overall system.

Next, we will write a module `write_thr` that writes a threshold to the THR register. We will connect `write_thr` and `read_berct` to the same `viterbi_ctrl` module (which makes sense, as there is only one set of actual pins to/from the physical chips). We will write a module `test_read_berct_and_write_thr` that will be the top level module and simulate it to test functionality.

```
`default_nettype none
module read_berct
(
    input clk,
    input reset_n,
    input start,

    output logic viterbi_ctrl_start,
    input  viterbi_ctrl_finish,
    input  [7:0] data_from_viterbi,

    output logic [2:0] addr,
    output logic finish,
    output logic [15:0] berct,
    input  vit_num,
    output logic vit_num_to_viterbi_ctrl,
    output logic [11:0] debug_state
);

    localparam idle                         = 12'b0000_0000_0000;
    localparam start_read_addr_0            = 12'b0010_0100_0010;
    localparam wait_read_addr_0             = 12'b0010_0000_0011;
    localparam transfer_berct_lsb           = 12'b0010_1000_0100;
    localparam start_read_addr_1            = 12'b0010_0101_0101;
    localparam wait_read_addr_1             = 12'b0010_0001_0110;
    localparam transfer_berct_msb           = 12'b0011_0000_0110;
    localparam assert_finish                = 12'b0000_0010_1001;

    assign debug_state = state;

    reg [11:0] state = idle; //idle is the initial value; don't really need to do this
                             //since the value of idle is 0

    logic latch_berct_lsb;
    logic latch_berct_msb;
    logic state_machine_is_active;

    assign addr[2:1]               = 0;
    assign addr[0]                 = state[4];
    assign finish                  = state[5];
    assign viterbi_ctrl_start      = state[6];
    assign latch_berct_lsb         = state[7];
    assign latch_berct_msb         = state[8];
    assign state_machine_is_active = state[9];

//note that vit_num_to_viterbi_ctrl is glitchy, but constant during operation
//of viterbi_ctrl, therefore permissable. Do this sort of thing with care
assign vit_num_to_viterbi_ctrl = state_machine_is_active ? vit_num : 0;

always_ff @(posedge clk)
begin
     if (!reset_n)
     begin
           berct[7:0] <= 0;
     end else
     begin
```

```
            if (latch_berct_lsb)
            begin
                  berct[7:0]  <= data_from_viterbi;
            end
      end
end

always_ff @(posedge clk)
begin
      if (!reset_n)
      begin
            berct[15:8] <= 0;
      end else
      begin
            if (latch_berct_msb)
            begin
                  berct[15:8]  <= data_from_viterbi;
            end
      end
end

 always_ff @ (posedge clk or negedge reset_n)
   begin
         if (!reset_n)
         begin
              state <= idle;
         end else
         begin
                 case (state)
                     idle: begin
                                  if (start)
                                  begin
                                         state <= start_read_addr_0;
                                  end
                            end

                     start_read_addr_0: state <= wait_read_addr_0;
                     wait_read_addr_0    : if (viterbi_ctrl_finish)
                                            begin
                                                  state <= transfer_berct_lsb;
                                            end
                     transfer_berct_lsb  :state <=  start_read_addr_1;
                     start_read_addr_1   : state <= wait_read_addr_1;
                     wait_read_addr_1    : if (viterbi_ctrl_finish)
                                          begin
                                              state <=  transfer_berct_msb;
                                          end
                     transfer_berct_msb  : state <= assert_finish;
                     assert_finish       : state <= idle;
                     endcase
         end
   end

endmodule
`default_nettype wire
```

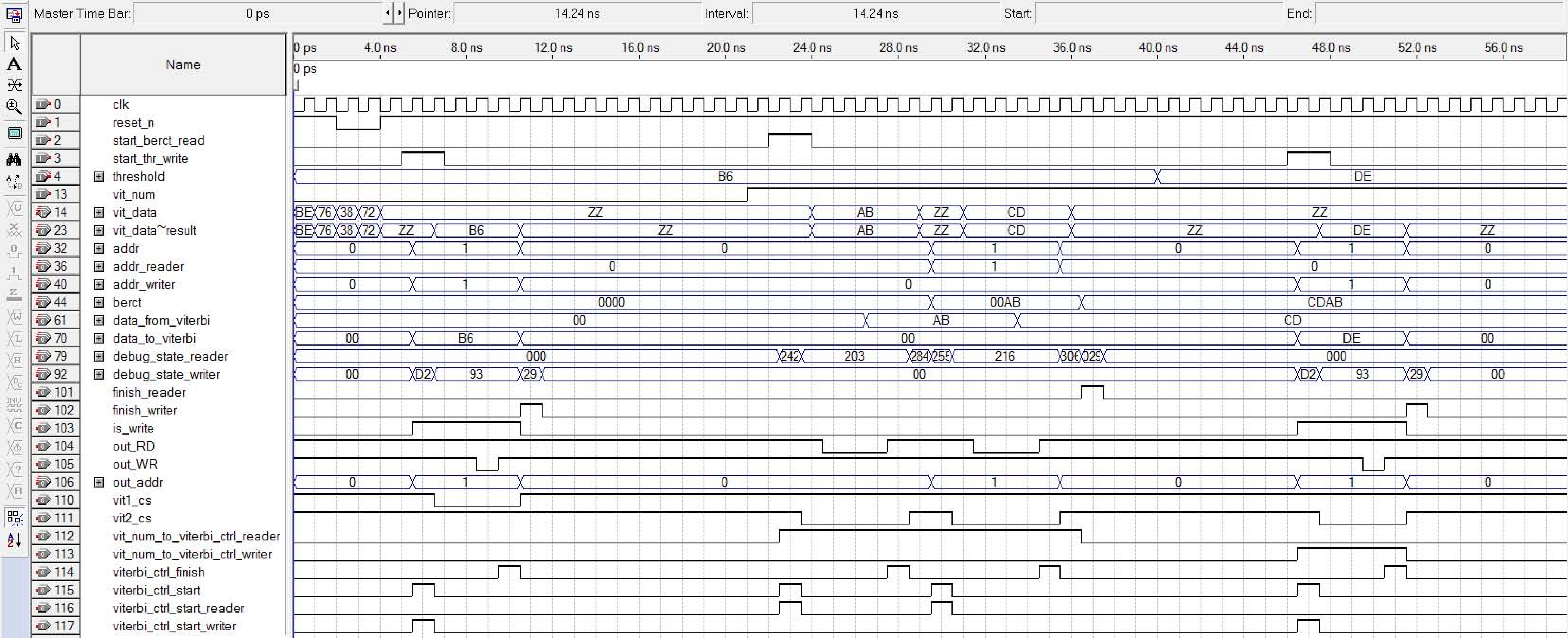
```
`default_nettype none
module test_read_berct
(
    input clk,
    input reset_n,
    input start,

    output logic viterbi_ctrl_start,
    output logic viterbi_ctrl_finish,
    output logic [7:0] data_from_viterbi,

    output logic [2:0] addr,
    output logic finish,
    output logic [15:0] berct,
    output logic [11:0] debug_state,

    input vit_num,

    output vit1_cs,
    output vit2_cs,
    output [2:0] out_addr,
    inout  [7:0] vit_data,
    output  out_RD,
    output  out_WR
);

logic vit_num_to_viterbi_ctrl;

read_berct
read_berct_inst
(
    .clk                     (clk                    ),
    .reset_n                 (reset_n                ),
    .start                   (start                  ),
    .viterbi_ctrl_start      (viterbi_ctrl_start     ),
    .viterbi_ctrl_finish     (viterbi_ctrl_finish    ),
    .data_from_viterbi       (data_from_viterbi      ),
    .addr                    (addr                   ),
    .finish                  (finish                 ),
    .vit_num                 (vit_num                ),
    .vit_num_to_viterbi_ctrl (vit_num_to_viterbi_ctrl),
    .berct                   (berct                  ),
    .debug_state             (debug_state            )
);
viterbi_ctrl
viterbi_ctrl_inst
(
.clk            (clk                            ),
.reset_all      (reset_n                        ),
.in_addr        (addr                           ),
.in_data        (0                              ),
.is_write       (0                              ),
.start          (viterbi_ctrl_start             ),
.vit_num        (vit_num_to_viterbi_ctrl        ),
.vit1_cs        (vit1_cs                        ),
.vit2_cs        (vit2_cs                        ),
.out_addr       (out_addr                        ),
.vit_data       (vit_data                        ),
.out_RD         (out_RD                          ),
.out_WR         (out_WR                          ),
.return_data    (data_from_viterbi               ),
.finish         (viterbi_ctrl_finish             )
);

endmodule

`default_nettype wire
```


Master Time Bar: 0 ps Pointer: 65.34 ns Interval: 65.34 ns Start: End:

```
`default_nettype none
module write_thr
(
    input clk,
    input reset_n,
    input start,

    output logic viterbi_ctrl_start,
    input  viterbi_ctrl_finish,
    input  [7:0] threshold,
    output [7:0] data_to_viterbi,

    output logic [2:0]  addr,
    output logic        finish,
    input               vit_num,
    output logic        is_write,
    output logic        vit_num_to_viterbi_ctrl,
    output logic [7:0] debug_state
);

    localparam idle                         = 8'b0000_0000;
    localparam start_write_threshold        = 8'b1101_0010;
    localparam wait_write_threshold         = 8'b1001_0011;
    localparam assert_finish                = 8'b0010_1001;

    assign debug_state = state;

    reg [7:0] state = idle; //idle is the initial value; don't really need to do this
                             //since the value of idle is 0

    logic latch_berct_lsb;
    logic latch_berct_msb;
    logic state_machine_is_active;

    assign addr[2:1]                = 0;
    assign addr[0]                  = state[4];
    assign finish                   = state[5];
    assign viterbi_ctrl_start       = state[6];
    assign is_write                 = state[7];
    assign state_machine_is_active = state[7];

//note that vit_num_to_viterbi_ctrl is glitchy, but constant during operation
//of viterbi_ctrl, therefore permissible. Do this sort of thing with care
assign vit_num_to_viterbi_ctrl = state_machine_is_active ? vit_num : 0;
assign data_to_viterbi = state_machine_is_active ? threshold : 0;

 always_ff @ (posedge clk or negedge reset_n)
   begin
        if (!reset_n)
        begin
             state <= idle;
        end else
        begin
                case (state)
                    idle: begin
                                 if (start)
                                 begin
                                        state <= start_write_threshold;
                                 end
                           end

                    start_write_threshold: state <= wait_write_threshold;
                    wait_write_threshold    : if (viterbi_ctrl_finish)
                                           begin
                                                 state <= assert_finish;
                                           end

                    assert_finish       : state <= idle;
                    endcase
        end
   end

endmodule
`default_nettype wire
```

```
`default_nettype none
module test_read_berct_and_write_thr
(
    input clk,
    input reset_n,
    input start_berct_read,
    input start_thr_write,

    output logic viterbi_ctrl_start_reader,
    output logic viterbi_ctrl_start_writer,
    output logic viterbi_ctrl_start,
    output logic viterbi_ctrl_finish,
    output logic [7:0] data_from_viterbi,
    input  [7:0] threshold,
    output [7:0] data_to_viterbi,
    output is_write,

    output logic [2:0] addr_reader,
    output logic [2:0] addr_writer,
    output logic [2:0] addr,
    output logic finish_reader,
    output logic finish_writer,
    output logic [15:0] berct,
    output logic [11:0] debug_state_reader,
    output logic [7:0]  debug_state_writer,

    input vit_num,
    output vit_num_to_viterbi_ctrl_reader,
    output vit_num_to_viterbi_ctrl_writer,

    output vit1_cs,
    output vit2_cs,
    output [2:0] out_addr,
    inout  [7:0] vit_data,
    output  out_RD,
    output  out_WR
);

logic vit_num_to_viterbi_ctrl;

read_berct
read_berct_inst
(
    .clk                     (clk                            ),
    .reset_n                 (reset_n                        ),
    .start                   (start_berct_read               ),
    .viterbi_ctrl_start      (viterbi_ctrl_start_reader      ),
    .viterbi_ctrl_finish     (viterbi_ctrl_finish            ),
    .data_from_viterbi       (data_from_viterbi              ),
    .addr                    (addr_reader                    ),
    .finish                  (finish_reader                  ),
    .vit_num                 (vit_num                        ),
    .vit_num_to_viterbi_ctrl (vit_num_to_viterbi_ctrl_reader ),
    .berct                   (berct                          ),
    .debug_state             (debug_state_reader             )
);
```

```
write_thr
write_thr_inst
(
    .clk                     (clk                           ),
    .reset_n                 (reset_n                       ),
    .start                   (start_thr_write               ),
    .viterbi_ctrl_start      (viterbi_ctrl_start_writer     ),
    .viterbi_ctrl_finish     (viterbi_ctrl_finish           ),
    .is_write                (is_write                      ),
    .data_to_viterbi         (data_to_viterbi               ),
    .addr                    (addr_writer                   ),
    .finish                  (finish_writer                 ),
    .vit_num                 (vit_num                       ),
    .vit_num_to_viterbi_ctrl (vit_num_to_viterbi_ctrl_writer),
    .threshold               (threshold                     ),
    .debug_state             (debug_state_writer            )
);

assign vit_num_to_viterbi_ctrl = vit_num_to_viterbi_ctrl_writer |
vit_num_to_viterbi_ctrl_reader;
assign viterbi_ctrl_start = viterbi_ctrl_start_writer |  viterbi_ctrl_start_reader;
assign addr = addr_writer | addr_reader;

viterbi_ctrl
viterbi_ctrl_inst
(
.clk           (clk                           ),
.reset_all     (reset_n                       ),
.in_addr       (addr                          ),
.in_data       (data_to_viterbi               ),
.is_write      (is_write                      ),
.start         (viterbi_ctrl_start            ),
.vit_num       (vit_num_to_viterbi_ctrl       ),
.vit1_cs       (vit1_cs                       ),
.vit2_cs       (vit2_cs                       ),
.out_addr      (out_addr                      ),
.vit_data      (vit_data                      ),
.out_RD        (out_RD                        ),
.out_WR        (out_WR                        ),
.return_data   (data_from_viterbi             ),
.finish        (viterbi_ctrl_finish           )
);

endmodule

`default_nettype wire
```


Master Time Bar: 0 ps
Pointer: 14.24 ns
Interval: 14.24 ns
Start:
End:
Name
0 ps
4.0 ns
8.0 ns
12.0 ns
16.0 ns
20.0 ns
24.0 ns
28.0 ns
32.0 ns
36.0 ns
40.0 ns
44.0 ns
48.0 ns
52.0 ns
56.0 ns
0 clk
1 reset_n
2 start_berct_read
3 start_thr_write
4 threshold B6 DE
13 vit_num
14 vit_data BE 76 38 72 ZZ AB ZZ CD ZZ
23 vit_data~result BE 76 38 72 ZZ B6 ZZ AB ZZ CD ZZ DE ZZ
32 addr 0 1 0 1 0 1 0
36 addr_reader 0 1 0
40 addr_writer 0 1 0 1 0
44 berct 0000 00AB CDAB
61 data_from_viterbi 00 AB CD
70 data_to_viterbi 00 B6 00 DE 00
79 debug_state_reader 000 242 203 284 255 216 306 329 000
92 debug_state_writer 00 D2 93 29 00 D2 93 29 00
101 finish_reader
102 finish_writer
103 is_write
104 out_RD
105 out_WR
106 out_addr 0 1 0 1 0 1 0
110 vit1_cs
111 vit2_cs
112 vit_num_to_viterbi_ctrl_reader
113 vit_num_to_viterbi_ctrl_writer
114 viterbi_ctrl_finish
115 viterbi_ctrl_start
116 viterbi_ctrl_start_reader
117 viterbi_ctrl_start_writer

Start Finish Protocol Concluding Remarks

The Start-Finish protocol is just an example of how defining a protocol can aid greatly in simplifying the design process. It is just a suggestion, and you can define your own protocols as you see fit. Of course, the protocols must be reliable and, preferably, as simple as possible.

In general it is advantageous to define a reliable protocol in the very early stages of the design process, as it helps in partitioning the design into submodules and dividing the work between different engineers. This also helps in the subsequent integration of the modules because there is no need for different engineers to invent and debug different ad-hoc interfaces between the modules that they design.

A clear, clean, standardized interface also promotes design reuse, which is a huge advantage. Very few designs are built from scratch - reusing allows systems to be built quickly and reliably, since reused modules have already been proven to work previously. Often the difference between a successful company and a failed one is due mostly or entirely to how efficiently the company is able to reuse code and design efficiently, since this has a decisive effect on time-to-market.