

Mini-Project: An Introduction to RTL Design

(revised, with extra credit option)

This homework is due on **Monday, December 12 (last day of classes), at 4pm** (submission details to be announced soon).

In this problem, you will design and optimize an entire subsystem, top down, using an RTL design flow. In particular, you will choose one of two problems, each targeted to a useful real-world custom hardware unit: (i) a *hybrid floating-point adder*, or (ii) a *fault-tolerant channel interface for an on-chip router node*.

This is a *written problem only*. You are not required to do simulation or VHDL for this problem. However, for *extra credit*, you can optionally do VHDL modelling and simulation.

Introduction. This homework is an introduction to top-level digital system design. In each of the 2 problems, you are given a verbal description of a custom chip (ASIC) or subsystem. You will then write an algorithm for its behavior in pseudo-code, formalize the specification as a generalized ASM, and then derive a complete microarchitecture, including both the datapath components and a controller (i.e. FSM). You will then work to further optimize your design. *You will pick only one of the two problems below, and do a complete and optimized solution.*

Grading. The mini-project is *worth 18%* of your final grade.

Note: Handout #39a, to be released soon, includes a short additional assignment, which is an introduction to asynchronous burst-mode controllers using the Minimalist CAD tool. This asynchronous assignment is *worth 2%* of your final grade, and should take you only a few hours to complete.

Working in Groups. You are allowed to do this homework in a group-of-two. You both get the same grade. However, solo homeworks are also allowed.

Extra Credit. The required part is to a written-only design. The extra credit option is to do two extra steps: making VHDL models for all components, then assembling them and simulating them. This option is worth 3% of your final grade.

Note: We will not be too lenient on partial credit for this extra credit problem. You need to successfully complete a significant portion of this VHDL modelling and simulation to receive any credit. (We will give some partial credit, but not if your solution is incomplete or has major problems.)

Also note: I treat this as true extra credit. If you do not do this part, and everyone else does, your grade will not suffer. The extra credit points can only help you.

Overview. In particular, for problems #1 and #2 below, you will follow the first 7 *steps* of Handout #34. (Steps #8-9 are for extra credit only.)

- 1 Write an **algorithm** (in pseudo-code) for the system's behavior;
- 2 Write a formal *specification* for the algorithm in the form of a **generalized ASM** (*Moore style*);
- 3 Do **resource allocation** of datapath components, i.e. select datapath blocks needed (including any necessary MUXes, hardwiring of inputs, and optimizations as in Handout #34);
- 4 Then, **identify status signals** (outputs of datapath, which are inputs to control), and allocate any additional datapath blocks required to generate these status signals;

- 5 Draw the final **micro-architecture** of the system, showing all datapath blocks, a single block for the control, and including all wiring between blocks (external inputs/outputs, control and status signals);
- 6 Derive a **Moore controller ASM specification**, using simple (B/V-style control-only) ASM;
- 7 Generate the corresponding **Moore state diagram** for the controller specification.
- 8 *OPTIONAL: extra credit only.* Generate a **VHDL model** of your system. You are allowed to use structural modelling throughout, to get a nice hierarchical view. For individual components, you should use *dataflow model* for each combinational component, and an appropriate *sequential (i.e. behavioral) model* for each sequential datapath block. As with Homework #5, use an appropriate top-level *sequential (i.e. behavioral) model* for the symbolic Moore state diagram, following one of the B/V templates.
- 9 *OPTIONAL: extra credit only.* **Simulate** your VHDL model, using the Altera tool environment, generating appropriate test vector sequences to thoroughly test the design, and hand in the result. *You will also need to send the VHDL models and simulation results electronically to the TA; details will be provided in the next few days.*

That is, you will simply follow Steps #1-7 of Handout #34, but applied to one of two new design problems. For extra credit, then map to VHDL models, and simulate the resulting microarchitecture. More details on Steps #8-9 will be provided shortly: modelling templates, what to hand in, etc.

Note: *Modelling the Moore Machine.* Do not design a gate-level implementation of the Moore state diagram. Just create the symbolic Moore state diagram which is the FSM specification.

Ground Rules, Assumptions: Problems #1-2. Use the following design guidelines. Any further updates will be announced on the class web page or in class.

- **Datapath Blocks.** Handouts #37-38 provide you with the library of components which you are allowed to use in your implementation. Try to *only* use datapath blocks that are given in Gajski, in chapter 5 (Handout #37) and in chapter 7.1-7.6 (Handout #38). *If you have a very compelling reason to use other blocks, I may permit it; but you must justify to me in advance for permission.*
You can feel free to generalize any smaller block (e.g. 4- or 8-bit) in Gajski up to 64 bits, or conversely to take an Gajski 32- or 64-bit block and assume a smaller one. For example, you can allow 32- or 64-bit shift registers, counters, etc. But be sure to add appropriate number of additional control signals, to scale to the larger designs. In general, *you are not allowed to scale any component beyond 64-bits.*
- **ROMs/RAMs.** For the given problems, you *should not use* any ROMs or RAMs, unless explicitly allowed.
- **Barrel shifters/rotators.** These can be quite useful. If you need them, see description in Gajski combinational handout.
- **Register Files vs. Separate Registers.** In general, since you are designing a custom ASIC, it is convenient to use a set of independent registers. This allows you the option to write or read arbitrary numbers of them in the same clock cycle, hence support more parallelism and complex and varying operations. However, an alternative more structured use of registers is also allowed: using a register file (see Gajski sequential handout). Register files allow indexed access, which may simplify some designs, but limit your access to them. In particular, you may only use a register file with *single write port* and *dual read ports*, i.e. which can perform at most 1 write and 2 reads per cycle. You can also use a combination of both free registers and a register file. However, you may not need any register files.
- **Combinational Integer Multipliers.** You are allowed to use combinational integer multipliers, as presented in the class. I.e. you can assume either the unoptimized or optimized array multipliers of Handout #20. You may assume any size multiplier up to 64 bits. You may assume that the multiplication always completes in less than 1 clock cycle.

- **Input and Output Bus Operations: Timing Assumptions.**

Bus Reads/Writes. When reading a value from an input bus, follow the same strategy as in Handout #34: do *not* do any operations directly using an operand on the input bus, it *must be written to a register or other storage unit first*. Likewise, when writing a value to an output bus, assume that this write operation takes an entire clock cycle. Do *not concatenate* or combine a combinational operation (e.g. to generate a result) followed by placing its result on the output bus, all in the same clock cycle.

Input Bus Validity. How long should data be assumed to be valid on the input bus? In Handout #34, we assume it is valid in the same cycle as “Start” is asserted, and remains valid for 1 extra cycle (enough time for it to be stored by the datapath). Use the same assumption for your solution; if you have compelling reasons to change this assumption, you must get permission of the instructor.

- **Number of Operations Per Clock Cycle.** You must use reasonable assumptions as to the clock speed. For example, you can assume that every basic combinational function block (Handout #37) completes in less than 1 clock cycle, such as adders, combinational multipliers, barrel shifters, comparators, etc.

However, in general, *do not concatenate two large combinational blocks within a given clock cycle*. For example, you should not have a multiplication followed by an addition all within the same clock cycle; the multiplication should be in one cycle and the addition in the next clock cycle. Otherwise, you would be requiring a slow clock rate.

However, there are useful exceptions to this rule. Small components like MUXes, etc. should be considered as having very little delay, so you *can* have a MUX concatenated with a larger block, and assume that together they always compute within the same clock cycle. Also, you *can* have a *barrel shifter or other combinational shifter*, attached to inputs or outputs of a combinational function block (adder, multiplier, ALU), where the combined shift + operation fits into one clock cycle. This is a common combined structure used in many processors’ “EX” stages, for example. Likewise, a pair of fast combinational operations such as “decode” followed by “add” (or a logical bitwise operation) are also allowed in the same clock cycle. Use good judgment, and follow the basic types of decisions you see in Handout #34.

The use of two consecutive or serial combinational operations in the same cycle is called **chaining**. The resulting hardware is a *combined structure* with one combinational block feeding as input to the next block, forming a single custom unit. You must explicitly designate such a combined unit in your “datapath allocation” step, if you use chaining.

Ask the instructor for permission on cases of chaining that you want to use. In general, you should largely *avoid chaining* except for rare simple cases.

Notating legal chaining operations in your generalized RTL specification: For cases where chaining is allowed, such as the above example of add-followed-by-shift, you need to notate the combined operation in a single RTL statement in your specification. For example, to note “add R2 + R3, then shift-right-by-1, then write to R5” in your RTL, you would use the notation:

$$R5 \leq [(R2 + R3) \gg 1].$$

- **Hamming Error Correction Operation and RTL Design.** If you select problem #2, you may be implementing a Hamming error correction operation. See Handout #27 for details. Given a non-0 syndrome for a received codeword, error correction is typically implemented by a decoder followed by a simple array of XOR2 gates (one gate per bit). Hence there are two combinational operations in sequence: “decode” followed by “bitwise XOR”. Given that this is small and fast hardware, you are allowed to chain both of these operations into 1 clock cycle. At the RTL level, you are allowed to write a simple statement using an operator such as “Hamming-correct(...)”. At the hardware block level, however, you must explicitly show the two chained blocks, together forming a custom unit, and their interconnection, to implement the 1 cycle correction operation, as part of the datapath allocation step.

- **Moore vs. Mealy Generalized ASM.** Use a Moore-style generalized ASM, as in Handout #34 (not Mealy).

- **Initialization, Reset.** Your design should follow the same basic assumptions as in the one presented in Handout #34. In particular, for problems #1-2, (a) assume a *Start* external input which activates the algorithm; (b) when done, unless stated otherwise, generate a *Done* external output for 1 clock cycle, as well as place the result on an *Output* data bus for that same one cycle. However, for problem #2, where multiple rows of a product code are placed in sequence on the output bus, follow the directions on how two control outputs are used to flag the start and end of the output transmission.
- **Optimizing Your Design: Guidelines and Suggestions.** Each of the problems below can be solved with many different generalized ASM specifications and microarchitectural implementations.

In general, as a design strategy, first focus on a correct and simple design. Do not make it either very sub-optimal or over-optimized/complex. Instead, work out a “clean” fairly efficient initial solution which works correctly and has an easy-to-follow flow.

Then, for your final version, focus on optimizing your specification (generalized ASM) and its resulting micro-architecture. *Most of your optimization should be captured in your generalized ASM specification*, i.e., optimizing the underlying algorithm specification to allow for greater parallelism, etc. This ASM should *still translate fairly directly* to your microarchitecture. That is, you should not make a lot of hand optimizations on the micro-architecture, beyond the small types done in Handout #34; you want your specification to translate cleanly directly into the optimized hardware.

The primary optimization you should focus on is to improve overall performance. Increased performance often comes at the expense of increased area, and you should use judgment if the increase is reasonable. *Increasing your design by a factor of 10 area is not reasonable!* A good rule of thumb, that we will generally enforce, is that *you may consider the use of up to 4 parallel function blocks of the same type within the same clock cycle* if you can support them in a correct algorithm with clear pseudo-code, and so that they can handle all input scenarios. After optimizing for improved performance, secondarily, improvements to area will be considered. There are a number of creative ways to increase parallelism for these problems, and thereby to optimize the critical path. Other optimizations include reducing unnecessary hardware. Points are also given for elegance of solution.

Above all, even if your design is heavily optimized, the top-level generalized-ASM specification should be clean and understandable.

- **Optimizing Your Design: Grading.** For Problem #1 below (*hybrid floating-point adder unit*), a basic slow but correct solution will already obtain *80% of perfect grade*. An improved design with moderate optimization will obtain *90% of perfect grade*. The remaining *10%* of the grade will depend on the quality and/or performance of your solution.

For Problem #2 below (*fault-tolerant channel interface unit*), a basic slow but correct solution will obtain *80% of perfect grade*. An improved design with moderate optimization will obtain *90% of perfect grade*. The remaining *10%* of the grade will depend on having a well-structured and more efficient design.

Problem #1: Hybrid Floating-Point Integer Unit

In this problem, you will design a specialized floating-point adder: it takes an *integer* and a *floating-point number* as operands, adds them, and produces the resulting *floating-point number* as the resulting sum. It is useful for specialized applications, where floating-point addition is needed and one operand is always an integer.

More precisely, given two operands, (i) an arbitrary *32-bit integer* (in 2's complement notation), and (ii) a *32-bit floating-point number* in IEEE 754 single-precision floating format, your unit will add them, and (iii) output the resulting sum as a *32-bit floating-point number*, also in IEEE 754 single-precision floating format.

Similar to Handout #34, assume that the two input operands, *Input-I* and *Input-FP*, arrive in the same cycle as a control input *Start=1* (asserted high for one clock cycle), each on a separate input bus. Also, assume that the final result should be generated on the output bus while simultaneously asserting the control output *Done* high for one clock cycle.

Handling Special Values: You should allow the special floating point values of “signed zero” (both positive and negative) and “signed infinity” (both positive and negative). These values should be correctly used in operations.

Rounding Mode: You should support a basic rounding mode, “Banker’s Rounding”: round to nearest number, ties to even.

Background and Related Reading: The basic IEEE standard floating point representation is covered in B/V ch. 5.7.2 (3rd edition), which will be assigned. Details of conversion between integer and FP representation, and FP addition and multiplication, are presented in the book, Patterson/Hennessy “Computer Organization and Design”, in section 4.8, in an older edition (2nd edition); *several copies are on reserve for the class in the Engineering Library*. I will also be covering basics of floating point in class lecture shortly.

Other resources include the Wiki page on ‘floating point’ and other books and online documents.

Hints: A good basic strategy in this specialized addition problem is to follow basics of a typical floating-point addition, but to see which steps can be simplified or omitted, because of the constraint that one of the operands is a 32-bit integer.

Optimizations: See the above two sections on “Optimizing Your Design.”

Problem #2: Fault-Tolerant Channel Interface Unit for On-Chip Router Node

Introduction. This problem is a nice practical application of the class material on error correction and detection, using product (i.e. two-dimensional) codes. The focus is on designing a unit to manage the communication on a single channel of an on-chip router node. The unit will send and receive packets on the channel, and also provide fault-tolerance.

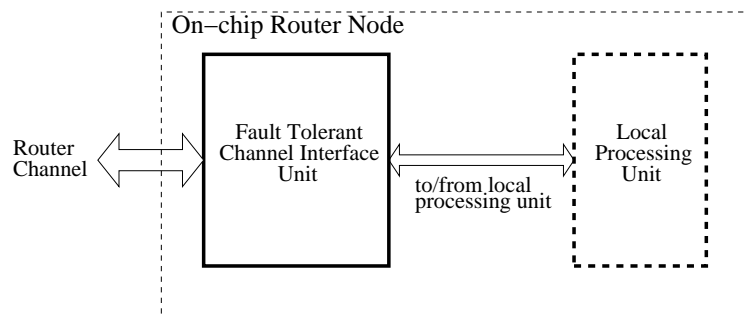
In particular, the unit will handle up to 3 errors per packet. It will provide complete error correction for all 1- and 2-bit errors. In the case of a 3-bit error, the unit will not be able to correct the errors; instead, it will detect the errors, and a retransmission will be performed, until a correct, or correctable, packet has been sent or received.

Note: While this writeup is longer than that of Problem #1, the problem is not more difficult, but is more “control-oriented”, involving flow control and protocol specification. Hence, it requires more detailed presentation of the problem.

Problem Overview. The target of this problem is to design an interface unit to manage a single communication channel attached to a router node. Packets will be sent and received on this channel. To ensure robustness, data will be encoded using a *product code*. The channel interface unit will transmit and receive the packets and also provide error handling.

The target application is for an *on-chip routing network*, for use in either an embedded system (i.e. network-on-chip, interconnecting various components) or a parallel processor (i.e. for interfacing cores and memories in a multicore system). The details are not very important for this problem! However, it is useful to know that your component can be used as a building block to construct router nodes for these applications.

A block diagram of the router node is shown below, showing a single communication channel, the channel interface unit, and a local processing unit. You will be designing the channel interface unit.



In *transmitter mode*, the channel interface unit receives a packet from the local processing unit, “wraps” it with a product code, and outputs the encoded packet on the router channel.

In *receiver mode*, the channel interface unit receives an encoded packet from the communication channel, “unwraps” it, checks for errors and does appropriate processing, then forwards it to the local processing unit.

The channel interface unit has partly serial and partly parallel operation. In transmitter mode, the input data word is received from the local processing unit in a single clock cycle, and the resulting product codeword is output *row by row* over several consecutive clock cycles. In receiver mode, the product codeword is received *row by row* over several consecutive clock cycles, and the resulting correct data word is sent to the local processing unit in a single clock cycle.

Error Handling. Packets sent and received on the communication channel will be encoded using product codes (see details below). The goal is to provide (i) *error correction* for some types of errors (1- and 2-bit), and (ii) *error detection* for more severe types of errors (3-bit). For case (i), your unit will identify and perform error correction, so no additional communication is needed. For case (ii), your unit will identify but not correct errors; in this case, the unit will *redo the transaction*: either requesting a repeat of the packet

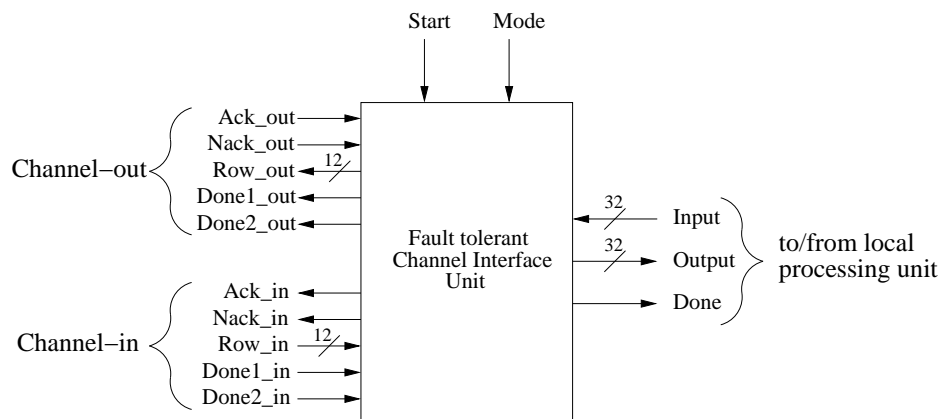
(in receiver mode) or provide a repeated transmission (in transmitter mode). This protocol will repeat, as needed, until finally a correct or correctable packet has been communicated.

Product Code: Basics. For this problem, you will use a variant of the product codes covered in Hand-out #27 and in class lectures. Assume a 32-bit data word, divided into 4 bytes, with 1 data byte (i.e. 8 data bits) per row.

To obtain better error coverage, you will use a *hybrid product code*: (i) for each row, a standard minimum-distance-3 *Hamming code* will be used, by including an appended check field (providing 1-bit correction and 2-bit correction); and (ii) for each column, an *even parity code* will be used, by including an appended check bit. For the final (i.e. 5th) row of parity bits, a Hamming check field is also be appended. As a result, there are 5 rows, where each row contains a byte (i.e. 8 bits) and a 4-bit check field for a total of 12 bits per row.

This hybrid code has *minimum-distance-6*. With careful analysis of scenarios, assuming no more than 3 errors ever arrive per packet, you will be able to identify and correct all 1- and 2-bit errors, and also detect all 3-bit errors.

Block Structure and Operation: Top-Level. The figure below shows a detailed block for the interface unit.



There are two signals common to both modes: the control input *Start* and the mode selection input *Mode*. When *Mode* is set to 1, the unit enters transmitter mode; when *Mode* is set to 0, the unit enters receiver mode. The unit mostly has separate inputs and outputs to handle each of the two modes. This will simplify your design.

(i) *Transmitter Mode.* In transmitter mode, a packet is received from the local processing unit and then output on the communication channel. In particular, data is received on a 32-bit input bus, *Input*, from the local processing unit. The unit will construct the corresponding product code, or “packet”. As the product code is constructed, each row is transmitted on a 12-bit data output bus, *Row-out*. There are two distinct “done” control outputs: *Done1_out* is asserted high when transmitting the first row of the codeword, and *Done2_out* is asserted high when transmitting the last row of the codeword.

If the external receiver successfully receives the packet, it will assert the Ack_{out} signal high for 1 clock cycle, and the operation is complete. However, if the receiver does not successfully receive the packet, it will assert the $Nack_{out}$ (i.e. ‘negative ack’) signal high for 1 clock cycle. In the latter case, the unit will *re-transmit* the encoded packet on the channel, following the above protocol. This process repeats as long as necessary, until it finally receives an asserted Ack_{out} . At this point, the operation is complete.

(ii) *Receiver Mode.* In receiver mode, a packet is received on the communication channel, corrected if necessary, and then forwarded to the local processing unit. In particular, each row of a product codeword is received on a 12-bit data input bus, *Row-in*. The $Done1_{in}$ input is asserted high in the same cycle as the first row, and the $Done2_{in}$ input is asserted high in the same cycle as the last row. In addition, there are a

pair of control outputs, Ack_{in} and $Nack_{in}$, which are used to acknowledge the sender of the product code input.

Once an entire product code is received, if there are 0, 1 or 2 errors, and correction can successfully be performed, then the unit asserts Ack_{in} high on the communication channel, indicating that no re-transmission is needed.

However, if 3 errors detected, correction cannot be performed, hence the unit asserts $Nack_{in}$ (i.e. 'negative ack') high, requesting a re-transmission of the input packet. In this case, the unit will receive a re-transmitted encoded packet on the channel from the sender, following the above protocol. This process repeats as long as necessary, until the unit finally receives a packet with 0 errors (no correction needed) or 1 to 2 errors (correction can be successfully applied).

In both scenarios, either with or without required retransmission, once the final correct 32-bit data word is generated, it is sent on the 32-bit *Output* bus to the local processing unit. The *Done* control output is also asserted high (for 1 clock cycle) in the same clock cycle, and the operation is complete.

Detailed Operation and Timing.

(i) *Transmitter Mode*. Similar to Handout #34, assume that the 32-bit data input, *Input*, arrives in the same cycle as a control input $Start=1$ (asserted high for exactly one clock cycle). Also, in the same clock cycle, the control input $Mode=1$ (also asserted high for only one clock cycle). Your unit will then divide this data word into 4 bytes, from MSB to LSB. Each byte will appear in a consecutive row of the resulting product code. Finally, the unit will output resulting product code, *one row per clock cycle*, in *consecutive clock cycles*, on the 12-bit data output bus, *Row-out*. Control output $Done1_{out}$ is asserted high for exactly one clock cycle simultaneous with the output of the first row, and control output $Done2_{out}$ is asserted high for exactly one clock cycle simultaneous with the output of the last row.

After asserting the output $Done2_{out}$, the unit will receive an acknowledgment, either an Ack_{out} or $Nack_{out}$ input, asserted high for one clock cycle. The acknowledgment may appear several clock cycles later, with unknown timing, depending on network activity. So you should be able to handle it arriving at any number of clock cycles (i.e. 1 or more) after $Done2_{out}$ is asserted.

(ii) *Receiver Mode*. Similar to Handout #34, assume that the control input $Start$ is asserted high for exactly one clock cycle. Also, in the same clock cycle, the control input $Mode=0$. Control input $Done1_{in}$ is asserted high for exactly one clock cycle simultaneous with the input of the first row, and control input $Done2_{in}$ is asserted high for exactly one clock cycle simultaneous with the input of the last row. The 5 rows are received on *consecutive* clock cycles on the channel input bus, *Row-in*. The first row, signalled by $Done1_{in}$, may be received in any clock cycle after the $Start$ pulse is complete.

In receiver mode, the unit will simultaneously support *2-bit correction* and *3-bit detection*. If there are 0, 1 or 2 errors, your unit will correct any errors and place the corrected data word on the *Output* bus. If there are 3 errors, your unit will only detect them, but should not correct them; in this latter case, it will request that the sender re-transmit the packet on the data input bus, *Row-In*. Once a re-transmitted input codeword is received, it is processed again; this protocol can iterate an arbitrary number of times until finally a product codeword is received which has 0, 1, or 2 errors. The unit will then correct any errors, place the correct data word on the *Output* bus and complete its operation.

Your unit should process the codeword as it is being received. There are 3 cases: (1) the codeword has 0 errors; (2) the codeword has 1 or 2 errors; or (3) the codeword has 3 errors. *Part of your work on this problem is to identify and distinguish these cases.*

In case (1), no correction is needed. The 32-bit data word is placed on the *Output* bus for one clock cycle. In the same clock cycle, *Done* is asserted high (indicating that the output is valid) and Ack_{in} is asserted high (indicating to the sender that the input was accepted).

In case (2), correction must be performed. Once the final corrected 32-bit data word is obtained, it is placed on the *Output* bus for once clock cycle, and the rest of the case (1) protocol is followed.

In case (3), correction cannot be performed, since this product code in general does not support 3-bit correction. Instead, soon after the entire input packet has been received, the control output $Nack_{in}$ is asserted high for exactly one clock cycle. This event indicates to the sender that the input was not accepted. In this case, note that the control signal $Done$ is *not* asserted high, since the unit has not yet produced a final correct data word. Shortly afterwards (within a small unknown number of clock cycles), the sender transmits a new version of the product codeword on the $Row-in$ data input, row by row in 5 consecutive cycles, following the identical protocol as outlined earlier. This new codeword is processed exactly as the earlier one. If it has 0 errors (case (1)) or 1 or 2 errors (case (2)), the processing is completed as covered in the preceding paragraphs for these two respective cases ((1) and (2)). However, if it again has 3 errors (case (3)), the request for retransmission follows the outline above for case (3), and this process can iterate multiple times, until there are no longer 3 errors.

Note that in each of the above three cases, when a final correct data word is placed on the $Output$ bus, the control outputs $Done$ and Ack_{in} are each asserted high in the same clock cycle.

Control Inputs and Outputs: Default Values. In general, all the control inputs and outputs remain at 0 unless otherwise indicated. When these signals are asserted high, they remain asserted for only 1 clock cycle.

Assumptions and Hints: Handling Errors. Your unit only needs to handle 0 to 3 errors, inclusive. Assume that there will never be more than 3 errors.

A key part of this problem, for “receiver mode”, is to distinguish the 3 different error scenarios: (1) no errors, (2) 1 to 2 errors, and (3) 3 errors. The unit should be able to identify every pattern of 1 or 2 errors that may arise (as observed by row and column check fields), and properly perform error correction in each case. It should also be able to identify every pattern of 3 errors, and *distinguish all such scenarios* from the 1 or 2 bit error cases. The reason is that 3 bit errors must always be handled through detection and a re-transmission request, while 1 or 2 bit errors are always handled simply through correction.

Hint #1 and Requirements: I suggest that you work out a series of error patterns in the product code (i.e. 2 dimensional) array, for all different configurations of 1-, 2- and 3-bit errors. The “error patterns” should indicate the *number of row and column* errors that can be observed, for each pattern of errors. Create a written table (to hand in with the assignment), which identifies which error patterns can occur for each of the 3 types of error scenarios (i.e. (1), (2) and (3)). Use this table to guide the error handling in your interface unit.

Hint #2: For min-distance-3 Hamming codes, which you will use for the rows, remember that these codes can detect all 1-bit errors or detect all 2-bit errors in a given row. This means that the *syndrome* will be non-0 in each of these cases. However, for a 3-bit error, the syndrome may be either 0 or non-0, and detection is not guaranteed for all cases.

Optimizations: In *transmitter mode*, the key performance metric of your design is to minimize the latency (i.e. number of clock cycles) between data input arriving from the local processing unit on the input bus $Input$ and generating the first row on the output channel bus $Row-out$.

In *receiver mode*, the key performance metric of your design is to minimize the latency between receiving the last row of the input codeword on the input channel bus $Row-in$ (i.e., $Done_{in}$ asserted high) and *either* sending the final correct data word to the local processing unit on $Output$ (in cases (1) and (2)) *or* generating a re-transmission request ($Nack_{in}$ asserted high in case (3)). See also above two sections on “Optimizing Your Design.”