**RISC-V Class Project Phase 9 – ECC Instructions + Test**

Phase 9 of the Class Project adds two new instructions to the RISC-V instruction set which implement an Error Correcting Code function. This will demonstrate how easy it is to add new instructions to the RISC-V architecture. No schematic changes are necessary in Phase 9. Copy your standardname8 project to standardname9.

1. **New Instruction Definitions**

We will create two new R-type instructions called ENC (encode) and DEC (decode).

ENC takes the lower 16 bits of the data in the register selected by rs1, computes the 6 parity bits necessary to perform single bit correction/double bit detection on the 16-bit input data, places these bits in bits [21:16] of the src1 input and writes the result to the register selected by the rd field. The rs2 input is not used. Figure 1 Shows the process for ENC.

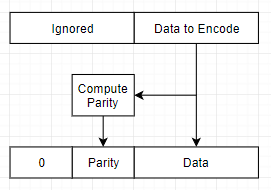


Figure 1

DEC reads a data word (with 22 bits of information) produced by ENC from the register selected by rs1 and performs the error check on it, computing the check bits and the Syndrome. The register selected by rs2 is not used. If no errors are detected, the lower 16 bits of the input word are written to the register selected by rd (i.e., XORed with all zeroes). If a single error is detected, it is corrected by XORing the bit in error with a 1 (and all other bits with a 0), the corrected 16-bit value is written to the register selected by rd, and bit 30 of the output word is set. If multiple errors are detected, the value 0x80000000 (bit 31 set) is written to the register selected by rd. Figure 2 shows this process. Bits 30 and 31 are referred to as the Signal bits, so that software can determine if an error was detected or corrected.

A close up of text on a white background

Description automatically generated

Figure 2

1. **Add the Instructions to the IA Model**

The ENC and DEC instructions must first be added to the IA model so that the Assembler will recognize them. As with the SUB instruction, these instructions will use the existing FN3 values for AND (ENC) and OR (DEC), but change the FN7 value to 0b0100000.

Add the new instructions to the opcodes.hcodal file, following the pattern for other instructions.

Modify the isa.codal file by adding decodes for the new instructions. Add the new opc values to the opc\_alu set. For this exercise we will not implement the actual instruction functions in the IA model, so these instructions will pass the opcode to the alu() function in ia\_utils.codal which will produce a zero output.

At this point build all of the IA functions in the Task window – Model Compilation (ia), Assembler (ia), Disassembler (ia) and Simulator (ia). The IA SDK standardname9.ia now exists and will be used to build software for this phase.

1. **Add New ALU Enums in ca\_defines.codal**

Each of the new instructions will have a new ALU function, so add enums for each of them in ca\_defines.

1. **Add the New Functions to the Decoder**

Add the new ALU functions into ca\_decoder.codal.

1. **Implement the ALU Operations**

We are using a Hamming Code to implement error correction and detection. The textbook describes this process starting on page 412 in Section 5.5, but the example there shows encoding for an 8-bit word and we want to implement a 16-bit version. The “Hamming Code” section in Wikipedia has a lot of useful information. Figure 3 shows a 16-bit implementation which we will use in Phase 9, with the bits numbered from 0 to 15 and the double bit correcting parity bit p32 included.

Table

Description automatically generated

Figure 3

A Hamming Code is implemented by making the computed parity across the various groups of bits indicated by X’s be even. Parity is calculated by XORing each of the bits in a group, so it is a 0 if the number of 1 bits in the group is even, and 1 if the number of 1 bits in a group is odd. Thus the Parity Bits p1, p2, etc. are each computed as the XOR of all the other bits in that row. Note that the bits are defined in the “Encoded Data Bits” row of Figure 3, with their position in the encoded word specified in the “Bit Position” row. For example:

p1 = d0 ^ d1 ^ d3 ^ d4 ^ d6 ^ d8 ^ d10 ^ d11 ^ d13 ^ d15

The Compute Parity block calculates the five Parity Bits, and places p1 in bit 16 of the output word, p2 in bit 17, etc. It also computes parity across ALL of the input bits AND the Parity Bits (21 total bits) and places that result in bit 21 of the output word. This overall parity function allows us to detect all double bit errors. This adds the 6th row to Figure 3 in which all of the bits are checked, and the 22nd column for p32 which has only the last row checked.

Implement this function for the ALU operation of ENC. I recommend creating six new 1-bit signals s\_ex\_ecc0 through s\_ex\_ecc5, one for each Parity Bit p1/p2/p4/p6/p16/p32. It is fine to use these signals only in the new functions and not in every case, since they are not outputs. These can then be concatenated with the input data (use “::” in Codasip) to produce the output result. Figure 4 shows the implementation of one of the Parity Bits (the one called p16 in Figure 3). Remember that the Codasip syntax for a bit field is [A..B], and both A and B must be included even if they are the same.



Figure 4

The DEC function initially implements almost exactly the same parity functions, except that the Parity Bit is included in each group as shown in Figure 5.



Figure 5

In this case we are computing the Syndrome, which consists of the 6 parity bits we generate. Concatenate the 6 bits into a single 6-bit word s\_ex\_ecc, with p1 as the LSB and the overall parity check as bit 5 – this is the Syndrome. Then implement a switch statement based on this numerical value, which will define how to modify the input data in order to correct errors.

The Syndrome affects the behavior of the switch statement in the following way. If it is 0, there are no errors and the input data is passed to the output with the Parity Bits removed. Other 6-bit values are evaluated using Figure 3. For each column, calculate the numeric value if the Xed bits are 1’s and the empty bits are 0’s. Remember that we added a 6th row which is all X’s. So, for example, the numeric value of the first column is (32 + 1) = 33, the second column is 34, the third column is 35, etc. If the Syndrome matches a column, the bit indicated at the top should be inverted by XORing with the proper constant and the result should be ORed with 0x40000000. If the Syndrome doesn’t match any of the column numbers, a multiple-bit, uncorrectable error has occurred, and we simply set the ALU output to 0x80000000. Note that if one of the Parity Bits was corrected, it is still not included in the output – only the lower 16 bits plus bits 30 and 31 should be set in the output.

Figure 6 shows the first part of the switch statement (for the first two data bits d1 and d2) and Figure 7 shows the last part of the switch (for the Parity Bits p8, p16 and p32 and the default case of uncorrectable errors).

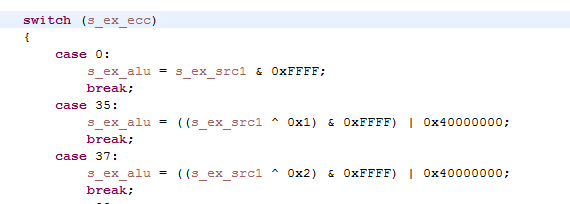


Figure 6

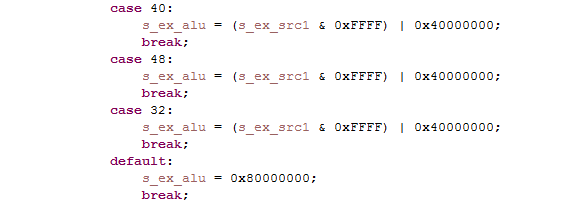


Figure 7

Add any new signals (like s\_ex\_eccN) to ca\_resources.codal, making sure their widths are correct.

1. **Build the Project**

Once all of the CA files are updated, build the project by first double clicking the button to the left of Model Compilation (ca) (NOT Model Compilation (ia)) in the Task window of the Codasip Perspective. Any build errors will appear in the Console window, so correct any missing assignments, syntax errors, etc. Continue until the Model Compilation builds correctly. The previous if\_data warning should now be gone, so there should be no warnings. If there are any warnings left over from Phase 8, they must be fixed.

Build Simulator (ca) in the Task window, and again correct any syntax errors.

1. **Create a Test Program**

Unlike previous Phases, a test program will not be provided so it must be created (as standardname9\_test). Copy phase8\_test and rename both the project and the test source file. Remove all of the code from phase8\_test except for the PASS and FAIL routines. After submission, a thorough test program will be used to evaluate the project for success, so it is imperative that you develop a program which tests all of the functions of ENC and DEC. The thorough test will not be provided to you. The basic process would be to run a number of iterations, each of which does the following:

1. Create some value to be tested.
2. Apply ENC to it to create the result with the check bits included. Write that to a register.
3. Modify that register value by XORing it with a value containing 1 or 2 ones, which effectively inserts 1 or 2 errors into the data.
4. Apply DEC to the modified register value.
5. If 0 errors were applied, verify that the result matches the original value and that bits 30 and 31 are 0.
6. If 1 error was applied, verify that the result matches the original value and that bit 31 is 0 and bit 30 is 1.
7. If 2 errors were applied, verify that the result is 0x80000000.
8. Do not apply more than 2 errors – there is no guarantee that you will not correct an incorrect bit.

Make sure that you exercise single bit errors in all of the bits (including the check bits) and a selection of double bit errors. The test should loop through various pieces, and not simply be a long stretch of inline code.

1. **Run the Test Program**

Once both the Model Compilation (ca) and Simulator (ca) tasks finish successfully, the next step is to run the test program standardname9\_test, and debug any errors. Note that the test program must be built with standardname9.ia in order for the enc and dec instructions to be recognized. In addition, the test program phase8\_test must pass on the Phase 9 hardware.

1. **Scoring**

The project should be submitted when the test programs standardname9\_test and phase8\_test pass when using the CA Model (be sure to select this in Debug Configurations and not the IA model which was used in Phases 2 and 3). Submissions which do not pass the thorough test will be rejected. The hardware project counts for 80% of the total Phase score.

The test standardname9\_test project must also be submitted. It will count for 20% of the total Phase score. Scoring for the test will be subjective based on the thoroughness of the testing, clarity of comments and overall structure. The test should exercise all possible single bit errors, and a selection of double bit errors, which can be inserted into the memory. Unlike the schematics of previous Phases, the test program will not be evaluated until the hardware project passes the thorough test.

The bonus is 1%/day (maximum 7%) for projects successfully completed before the Target Date and the deduction is 4%/day after the Target Date.

1. **Exporting the Project**

Once the test program is running, the hardware project should be Exported as standardname9 and the test program should be exported as standardname9\_test. These must be submitted at the same time.