**RISC-V Class Project Phase 7 – ALU/Branch/Jump Schematic + CA**

Phase 7 of the Class Project adds the logic to implement all ALU functions, all of the conditional branch instructions and the JAL/JALR instructions to the Schematic and the Codasip implementation created in Phase 6. The schematic update will be described first, followed by the Codasip code update.

1. **ALU Schematic**

Copy the schematic standardname6.xml to standardname7.xml. Alternatively named schematics will not be accepted. Open the schematic in draw.io.

Adding all of the ALU functions requires only two changes to the Schematic, since all of the instructions except LUI are handled simply by expanding the s\_id\_aluop field to provide more function choices to the ALU.

1. Because LUI requires a different immediate field (it is a U-type instruction instead of an I-type instruction), add a new control signal s\_id\_immsel in the ID stage coming from the decoder and going to the IMMGEN control logic. This signal will select between the various immediate values extracted from the instruction. Note that this connection will also handle the B-type and J-type immediates required for the branch and jump instructions later in Phase 7, and the S-type immediate in Phase 8. For readability, it may be better to label the output of the DECODER with a Signal Out symbol, and then label the input to IMMGEN with the same name using a Signal In symbol, rather than drawing a line between them. This is a common schematic implementation.
2. Adding the additional ALU operations will be a major function in the Codasip implementation of Phase 7, and the “alu” function in ia/other/ia\_utils.codal will indicate the functions we need. Since the goal is to minimize the number of ALU operations for performance reasons, it is desirable to implement LUI by adding 0 to the U-type immediate. Add a multiplexor on the source 1 side of the ALU in the EX stage, selecting between the normal input and a constant value of 0, similar to the immediate mux on the source 2 side. Since we will add a third selection for jumps, use a 3-1 multiplexor, and define the select as s\_id\_alusrc1 from the DECODER, pipelined to the EX stage.
3. **Branch Schematic**

Adding the conditional branch instructions follows the logic shown in Figure 4.49 of the textbook, reproduced below as Figure 1. The functions in the three red rectangles must be added.

* 1. **PC Input Multiplexor**

Add a 2-1 Multiplexor component in the IF stage, which allows the selection of a branch target address (r\_me\_bradd) from the ME stage in place of the (PC + 4) value when a conditional branch is taken. The selection is made by the s\_me\_pcsrc signal from the ME stage. The adder output should remain as s\_if\_nextpc, and the register input will be s\_if\_pcin.

* 1. **Branch Address Creation**

In the EX stage, add the Adder component which adds the PC value to the correct immediate value. The “Shift Left 1” function in Figure 1 is not necessary, as this function is handled in the IMMGEN logic. The output of the adder (s\_ex\_brimm) will be used below.

* 1. **Branch Selector**

Logic in the ME stage determines whether the conditional branch is taken or not. The zero output from the ALU (s\_ex\_zero) is asserted when the ALU output s\_ex\_alu is zero. This signal is pipelined to the ME stage, and is the only signal required to determine whether a conditional branch is taken or not.

A new control signal s\_id\_brnchop is required from the Decoder in the ID stage to determine how the zero indicator is used. Pipeline this signal to the ME stage. Add the control block BRNCH which creates the s\_me\_pcsrc signal which selects the next PC source, based on brnchop and zero.

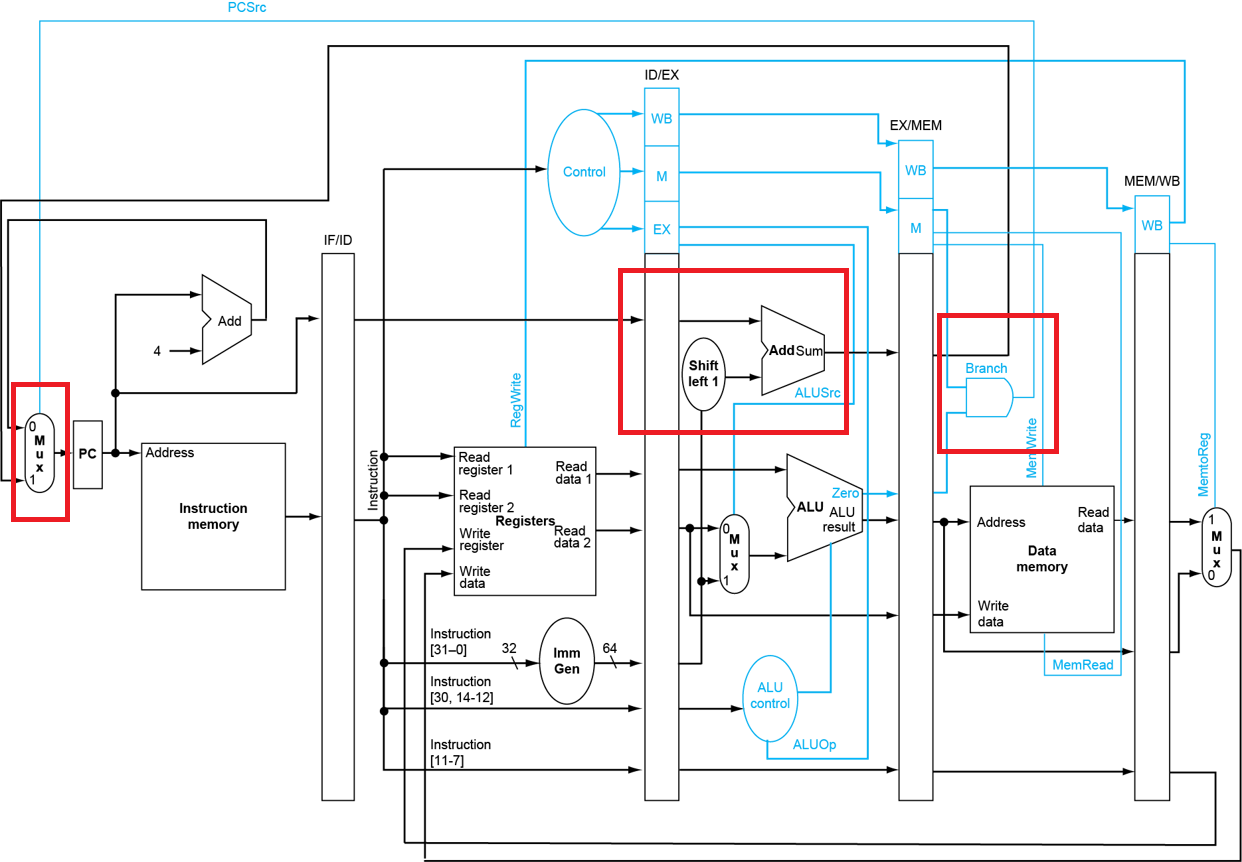


Figure 1

1. **Jump Schematic**

The jump logic is not included in any of the block diagrams in the textbook because the mini RISC-V processor of the book doesn’t have these instructions, so you will need to develop this from the functions of the instructions. There are several functions to be implemented.

* 1. **Write the PC Back**

Both JAL and JALR write the current PC value plus 4 into the destination register of the instruction. Implementing this requires a multiplexor selecting the write data to the Register File. Add a 3-1 Multiplexor component in the WB stage. One input is the current write data r\_wb\_alu. Add an Adder component which adds 4 to the current PC value r\_wb\_pc and connect it to the second input of the 3-1 mux. The third input can be left unconnected for now but will be used in Phase 8.

The Register File write data is now a signal (use s\_wb\_result) which is the output of the multiplexor. Note that this is the value which must be used in any forwarding case, so change this anywhere it is used in the ID or EX stages.

Add a new Decoder output s\_id\_rfwtsel to control the write data select. Pipeline this signal to the 3-1 mux select.

* 1. **Generate the Jump Address for JAL**

The JAL instruction jumps to the address which is a J-type immediate added to the current PC. This path already exists in the schematic for the branch instructions, although IMMGEN will create a different immediate value based on s\_id\_immsel. No schematic changes are necessary.

* 1. **Generate the Jump Address for JALR**

The JALR instruction jumps to the address which is the sum of a register and an I-type immediate. The ALU is used to produce the address for this operation since it already has all of the correct connections. Add a 2-1 multiplexor in the EX stage which selects between the output of the address adder and the output of the ALU and produces the final s\_ex\_bradd, which is pipelined to the ME stage. Add a new Decoder output s\_id\_brnchsel to control this mux and pipeline it to the EX stage.

* 1. **Create the AUIPC Path**

The AUIPC instruction adds a U-type immediate to the PC and writes it to a register in the Register File. The ALU is used for this, so connect the third input of the source 1 multiplexor on the RS1 side of the ALU in the EX stage to the current PC. The select for this mux already exists as s\_id\_alusrc1.

1. **Examples**

Figure 2 shows some rough sketches of what the IF Stage through WB Stage schematic pages should look like.

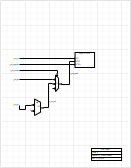
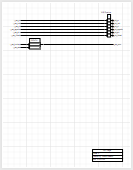
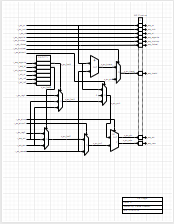
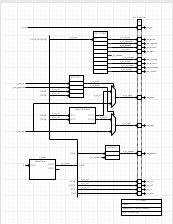
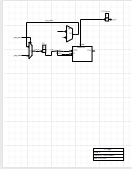


Figure 2

1. **Submit the Updated Schematic**

Although not required, it is recommended that you submit the updated schematic as soon as it is complete, as standardname7.xml. This will allow for feedback prior to creating the Codasip code. The scoring will reward early submission of the schematic.

1. **Update the Codasip Files**

Changes will be made to several of the stages, in each case requiring updates to ca\_resources.codal and ca\_defines.hcodal.

* 1. **Adding ALU Functions**

New functions must be added to the ALU to handle all of the required cases. The alu function defined in ia/other/ia\_utils.codal includes all of these functions (there are 11 total). Update the following files:

* + 1. **ca\_defines.codal**

Add function defines for all the required functions in the aluop enum. There must be an enum for each function which is used in the “alu” function in ia/others/ia\_utils/codal. Note that as more choices are added, the width of s\_id\_aluop (ALUOP\_W) is automatically adjusted to include enough bits to encode all of the choices.

Add the selections for the source 1 multiplexor, which will be very similar to the source 2 mux.

Add the enum selections for the IMMED function in the ID stage, as IMM\_xTYPE. It is recommended to add selections for all 5 immediate types at this point, even though only four (I-type, B-type, J-type and U-type) will be used in this phase.

* + 1. **ca\_resources.codal**

Note that no changes are required to ca\_resources.codal to handle the width change for aluop, but there are other new signals.

1. Add the new control signal s\_id\_alusrc1 and pipeline it to the EX stage.
2. Add the new data signals required because of the source 1 mux and the immediate selection s\_id\_immsel in IMMGEN.
3. Add five new definitions for the 5 immediate types s\_id\_immedX, where X is i, b, u, s and j. Note that the I-type immediate already exists as s\_id\_immed and the name should be changed. Note also that these signals are not explicitly shown in the schematics – we also need to create signals for anything required inside a functional block like IMMGEN.
   * 1. **ca\_pipe\_state3\_ex.codal**

Add the logic for the source 1 multiplexor, which will be very similar to the existing source 2 multiplexor. Note that this will change the signal which is the input to the ALU on the source 1 side, so update the functions in the ALU switch statement.

Add the function for each ALU operation choices to the ALU switch statement. The “alu” functions in ia\_utils.codal will provide guidance here. Some (but not all) functions are shown in Figure 3.

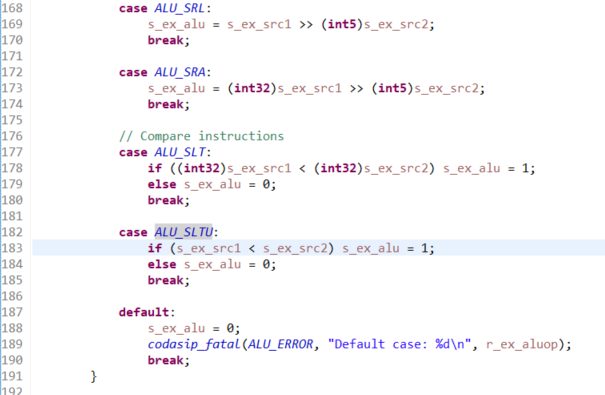


Figure 3

* + 1. **ca\_pipe\_stage2\_id.codal**

Add the generation of the other four immediate types and rename the I-type immediate. The I-type immediate can be used can be used as a template for the other immediates. If the immediate involves the “Shift Left 1” function, create the value and then shift it with “<< 1”. Note that the concatenation and field extraction functions will be used in many of these immediates – see the assignment to s\_id\_opc in this file for a good example. Remember that a single bit field still requires both a starting and an ending bit, e.g. s\_id\_instr[7..7]. See Figure 4 for an example.



Figure 4

Add the 5-1 multiplexor (which is in the IMMGEN control block) selecting the correct immediate choice based on s\_id\_immsel, using the enums created in Section 6.1.1. Note that this multiplexor must be AFTER the call to dec(), since the value of s\_id\_immsel is assigned in that function. The first part of this mux is shown in Figure 5. Assign 0 for IMM\_NONE, and add a new error code IMM\_ERROR in debug.hcodal since immediate errors are not uncommon and use that in the default of the switch statement.

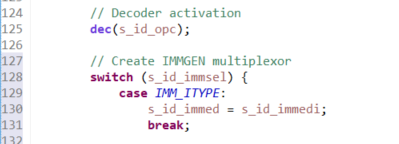


Figure 5

Add pipeline registers for any other new control signals like s\_id\_alusrc1.

* + 1. **ca\_decoder.codal**

Since new control signals from the DECODER have been added, they must be added to all of the instruction groups in ca\_decoder.codal. In addition, the correct choice for s\_id\_aluop must be added for all of the instructions in a group. Some examples for the i\_hw\_alu group are shown in Figure 6. The functions added to i\_hw\_alu\_i will be very similar, but note that the immediate shift instructions are not in i\_hw\_alu\_i.

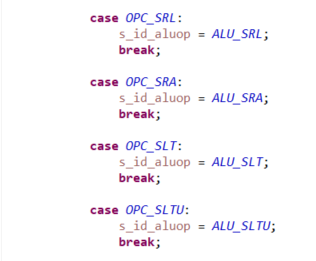
****

Figure 6

Since every control signal must be defined in every group, s\_id\_immsel must be assigned one of the immediate select values in each group. Only IMM\_ITYPE (and IMM\_NONE when no immediate is needed) will be used at this point, but others will be added later in Phase 7 and in later phases.

* 1. **Adding the LUI Instruction**

The LUI instruction is handled differently from other ALU operations because it has different control signals. The U-type immediate and the source 1 multiplexor option of 0 were added in section 6.1, so all that is needed is a new instruction group in ca\_decoder.codal.

The i\_upper\_imm group in isa.codal implements LUI, so create a new group i\_hw\_upper\_imm in ca\_decoder.codal. Copy i\_hw\_alu\_i (which is somewhat similar to LUI) to create this group. Note that this group also includes the AUIPC instruction. Use the correct opc function from isa.codal to define opc.

For each control signal, select the proper value to execute LUI. The ALU operation is always ADD so the switch statement is not required.

Add the new group to the “set dec =” at the top of ca\_decoder.codal. Omitting this will result in undefined instruction errors.

* 1. **Adding the Immediate Shift Instructions**

In a similar fashion to LUI, copy the group i\_hw\_alu\_i to a new group i\_hw\_shift, based on the i\_shift group in isa.codal. This will execute the three immediate shift instructions SLLI, SRLI and SRAI. This will look very much like the normal immediate instruction group.

* 1. **Adding the Conditional Branch Instructions**

Adding the 6 conditional branch instructions requires several changes, as can be determined from the schematics.

* + 1. **ca\_defines.codal**

Add the decode selections for the new control signal s\_id\_brnchop, as shown in Figure 7, and create BRNCH\_W as the field width. The four values implement branches as follows:

BRNCH\_FALSE – never branch (which must be the default and thus first in the list)

BRNCH\_TRUE – always branch (used for jump instructions in a later phase)

BRNCH\_COND\_TRUE – branch if the ALU output is zero (for conditional branches)

BRNCH\_COND\_FALSE – branch if the ALU output is not zero (for conditional branches)

This provides flexibility for the functions and enables a very simple hardware structure.

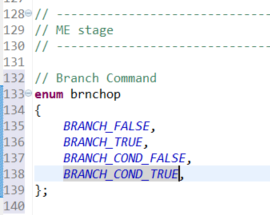


Figure 7

Create the enum and width defines for the WB stage multiplexor.

Create the enum and width defines for the EX stage multiplexor.

Add the third source 1 multiplexor select as ALU1\_PC. Note that the width of the control signal is handled automatically.

* + 1. **ca\_resources.codal**

Add definitions for all of the new signals added to the schematic, including the branch operation, the ALU zero output and the branch address. Note that several branch signals must be pipelined to the ME stage. s\_me\_pcsrc and s\_ex\_zero are 1-bit control signals and can be defined with bit[1]. Add the data path signals in the WB stage for the PC and the write result to the Register File.

* + 1. **ca\_pipe\_stage1\_if.codal**

Add the 2-1 multiplexor which selects the branch address r\_me\_bradd or the next PC value as the input to the PC. Because the select is a simple control signal s\_me\_pcsrc, the structure in Figure 8 can be used. Note that the input to the PC register must change to adapt to this.



Figure 8

One of the critical functions in RISC-V branches is that the pipeline must be flushed when a branch is taken, since three instructions which must not be executed have been fetched and are in the pipeline. This will require the first use of the special Codasip “clear” functions which are created in ca\_pipe\_control.codal. A pipeline stage is cleared by asserting the STG\_clear function for that stage. The three instructions to be flushed are in the IF, ID and EX stages, so the clear signals to the next stages (ID, EX and ME) must be asserted when a branch is taken. The signal s\_me\_pcsrc is asserted on a taken branch and therefore is ideal to create the flush. This is implemented by changing the inputs to all of the clear signals except s\_wb\_clear from 0 to s\_me\_pcsrc (r\_id\_clear must also be changed). Do not change the input to any “stall” signals.

* + 1. **ca\_pipe\_stage2\_id.codal**

Pipeline the new control signals from the DECODER.

* + 1. **ca\_pipe\_stage3\_ex.codal**

Add the adder which creates the branch address s\_ex\_bradd, for both branches and jumps. Also add the multiplexor which selects among the branch address choices.

Add the generation of s\_ex\_zero, which is asserted when the ALU output s\_ex\_alu is zero.

Pipeline all required signals in ex\_output.

* + 1. **ca\_pipe\_stage4\_me.codal**

The only new function in the ME stage is the creation of the BRNCH control function. This is a switch statement using the choices defined in Figure 7. Select the correct value for s\_me\_pcsrc for each of the four brnchop values. Note that some cases will depend on s\_me\_zero and some will not.

Pipeline any new control signals.

* + 1. **ca\_pipe\_stage5\_wb.codal**

Add the adder to increment the PC by 4.

Add the multiplexor for the Register File write result. Create a new error code RFWT\_ERROR in debug.hcodal for use in the codasip\_fatal in the default.

Change the write data input to the Register File to the correct signal. In addition, this change must be made everywhere that the previous write input r\_wb\_alu is used.

* + 1. **ca\_decoder.codal**

As has been done before, the first step in updating ca\_decoder.codal is to add the assignment of any new control signals to every existing instruction group. Since none of the existing groups ever branch, assign BRANCH\_FALSE for s\_id\_brnchop in each case.

Next copy the i\_hw\_alu group and rename it i\_hw\_branch to match the i\_branch group in isa.codal. The i\_hw\_alu is a good choice because the conditional branch instructions use the two Register File outputs as the ALU inputs. The switch statement will have cases which are the 6 conditional branch instructions. In this case, both the ALU operation and the branch operation must be selected separately for each instruction. The ALU function can be determined from the alu() function in ia\_utils.codal. The branch operation will be one of the conditional ones based on the conditional in isa.codal.

Add the functions for AUIPC into the i\_hw\_upper\_imm group. This should only change the s\_id\_alusrc1 control signal, which requires different values for LUI and AUIPC.

Copy the i\_hw\_alu\_i group to a new group corresponding to the i\_jalr group in isa.codal. The switch statement on opc is not necessary since this group includes only one instruction (JALR). Set all control signals appropriately. Note that JALR contains an I-type immediate. Use the proper value as opc.

Copy the i\_hw\_upper\_imm group to a new group corresponding to the i\_jal group in isa.codal. The switch statement on opc is not necessary since this group includes only one instruction (JAL). Set all control signals appropriately. Note that the ALU is not used in this case. Use the proper value as opc.

Add all of the new groups to the set of dec.

* 1. **Adding the New - -info Print Function Code**

We want to add a new - -info 11 print function to debug branches, and some adjustments may need to be added to make the ALU - -info 7 function work.

* + 1. **others/ca\_utils.codal**

Uncomment lines 41 and 42 as shown in Figure 9. This will enable the - -info 11 print function.

A screen shot of a social media post

Description automatically generated

Figure 9

Check the signals in line 36 of Figure 9. The first two signals must be the exact signal names of the two ALU inputs, which may have changed since Phase 6. This change will allow - -info 7 to work, which shows the ALU inputs and outputs. This is an extremely useful debug tool for Phase 7.

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 warning about if\_data is still acceptable in this Phase.

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

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 phase7\_test. Import this test program from G:/Information/Phase 7 and build it using your Phase 5 IA SDK. It may be valuable to first run the phase6\_test program from Phase 6, to make sure nothing basic has been broken and you are familiar with its behavior.

Execute Run -> Debug Configurations as in Phase 5/6. The C/C++ Project should be phase7\_test, and the Application should be Debug/phase7\_test.xexe. For the debugger, select standardname7.ca.standardname7-ca-simulator. If this is not a choice, the project build process has not completed successfully. If the test completes successfully, register 10 will have the value 49, register 25 will have the value 1 and the test should stop at line 571.

The running and stepping controls are the same as described in Phase 2. The debugging functions described there will be useful. Additional debug pointers are in section 9.

1. **Debug Information and Examples**

This section contains an additional set of debug information which is particularly relevant to Phase 7 and beyond. There are no design actions in this section.

* 1. **Decoder Error Example 1**

One error which can occur during a debug run is the failure of the DECODER to detect an instruction. This error is shown in Figure 10. When the run stops, the error will appear briefly in the Console window of the Debug Perspective and then disappear. The reason that the message disappears is that Codasip switches the console view. There are several different console views available, which are selected by the console pull down menu at the upper right corner of the Console window as shown in Figure 11. Clicking on the down arrow produces a selection of several console views. The error message can be seen by selecting the “New configuration [Codasip C/C++ Application] gdb (7.4.1)” window, which is then listed as the console in Figure 10. When the simulation is running normally, the console view will be the “New configuration [Codasip C/C++ Application] phaseX\_test.xexe” view, where any print messages will occur. An error message may also occur in that window as shown in Figure 12, but it is not highlighted in red. Note that the info variable was 3 for this run.

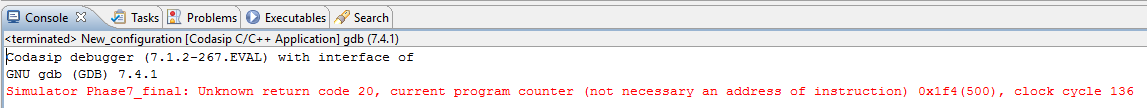


Figure 10

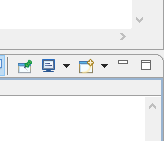


Figure 11

A picture containing bird

Description automatically generated

Figure 12

The notations about “return code 20” and “fatal(20)” indicate that a codasip\_fatal operation has been encountered with a code of 20. Looking at the “fatal\_codes” section in debug.hcodal, we see that 20 indicates ALU\_ERROR, so the codasip\_fatal operation has a first parameter of ALU\_ERROR. The “clock cycle 136” and “@136” notations indicate that the error occurred when decoding the instruction on the 136th clock. Since most programs don’t proceed perfectly linearly due to branches, this may not be the instruction on line 136 of the test. If info is set to 3 as it was in this case, it can be seen that the error occurs on an instruction after Register File register x3 was written to 0xaa when register x10 (the phase7\_test case) is set to 0x18 (case 24 in decimal).

The display when the error occurs is shown in Figure 13. Because x3 was written with 0xaa in the WB stage (where Register Files writes occur) in cycle 136 (printed after the fatal error print occurs), the bad decode occurred in the ID stage. The “mv x5, x3” instruction is in the ME stage, the “li x4, 0x2A8” instruction is in the EX stage and the “slli x3, x3, 2” instruction is in the ID stage and therefore produced the error.

A screenshot of a social media post

Description automatically generated

Figure 13

Look at ca\_decoder.codal for the decode of SLLI, which should be in the i\_hw\_shift group. The codasip\_fatal code in the default of this switch statement is ALU\_ERROR, indicating that this is the source of the error. The Default case: is reported as 147, which is the hex value of 0x93, which is the FN3 value of SLLI (001) concatenated with the OPCODE value of SLLI (0010011). This error was created by removing the case for OPC\_SLLI in the switch statement.

The error message in Figure 10 indicates that the program counter was 0x1f4. Although the message indicates this may not be the address of an instruction, it almost always is. If the Disassembly window has been enabled as described in Phase 5, the instruction at that address can be determined as shown in Figure 14. If the PC value is 0x1f4, the instruction in the IF stage is at address 0x1f0 and the instruction in the ID stage is at address 0x1ec. This is the SLLI instruction, just as we found by the stepping analysis above.

A screenshot of a cell phone

Description automatically generated

Figure 14

Note that when stepping through instructions, the console view may switch between the gdb (7.4.1) view and the phase7\_test.xexe view. Simply use the console pulldown in Figure 11 to switch back to the desired view. Clicking the monitor (not the pulldown) will select the last Console which was explicitly selected, which is very often the desired choice.

* 1. **Decoder Error Example 2**

A similar DECODER error is shown in Figure 15. This is a different error because there is no error code (and no message in the phase7\_test.xexe view) which indicates that this error was generated directly by Codasip and not by a codasip\_fatal operation. The “instruction decoding failed” message means that the dec() function received an opcode value which was not implemented anywhere in ca\_decoder.codal. The program counter was 0x1f4 as in the example from section 9.1, so the SLLI instruction is also the problem here, but there is no function which decodes it.

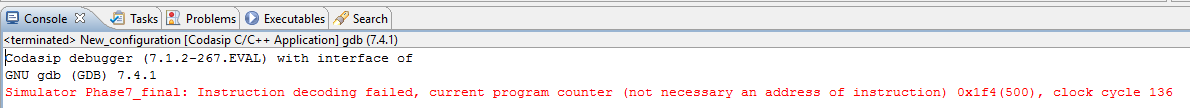


Figure 15

This error was created by removing i\_hw\_shift from the set dec function in ca\_decoders.codal, which causes all of the instructions defined there to be undefined in the DECODER (note that a build warning does occur in this case but it is just a warning). Omitting an instruction group entirely, or using an incorrect definition of the “opc” variable within a group, will also cause this error.

1. **Scoring the Project**

The project should be submitted when the test program phase7\_test passes 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). Unsuccessful submissions will be rejected. The hardware project counts for 80% of the total Phase score.

The score for a successful Project submission will be determined by the time of submission relative to the Target Date, which is Sunday, March 21 at 10:00 PM. A bonus of 1% of the Project score (i.e. 0.8% of the Phase 7 score) will be added for each day earlier than that the successful project is submitted, up to a maximum of 7%). A penalty of 4% of the Project score per day will be imposed after the Target Date.

The score for the schematic will be determined by the time of submission of a correct schematic with deductions for any errors. Error deductions will be similar to those for Phase 4 (1 point for a bad signal name, 2 points for a bad connection, etc.) but the deduction will be from the entire Phase 7 score, not just the 20% which is for the schematic. A bonus of 1% of the schematic score (i.e. 0.2% of the Phase 7 score) will be added for each day earlier than the Target Date a fully correct schematic is received, up to a maximum of 7%. A penalty of 4% of the Schematic score per day will be imposed after the Target Date. Submitting the schematic early not only gains schematic bonus points but produces a correct schematic which aids in creating the actual hardware project.

1. **Exporting the Project**

Once the test program is running, the project should be Exported as standardname7.zip to the G:/Submission folder.