**RISC-V Class Project Phase 5 – ADD + ADDI CA Implementation**

Phase 5 of the Class project will implement a Cycle Accurate (CA) version of the ADD + ADDI instruction function in the Codasip Codal language, based on the schematic created in Phase 4. Once the Codal code is written, the project will be built and a test program will be run to verify the design. The following steps are required for this Phase.

1. **Import the IA Model**

The CA implementation requires a full IA implementation, so Import G:/Information/Phase 5/Phase5\_orig into Codasip. Copy that project to standardname5 for this phase. Do not edit or remove any copyright notices.

Build the Model Compilation (ia), Assembler(ia), Disassembler (ia) and Simulator (ia). This will produce a working IA model that can be used to build and debug the software. Do not use your existing Phase 2 IA Assembler.

1. **Print the Schematic from Phase 4**

The best way to effectively create the Codasip code is to print Pages 1-5 of the schematic created in Phase 4, and then mark each element as it is implemented in Codal.

1. **Add the Required Code to Each File**

Eleven new files are included in a CA model, and several of them must be modified in order to complete Phase 5. The updates to each file are described in the sections below. Note that some changes require edits to several files at once, so read the instructions carefully. Good commenting is very valuable and will save lots of time in subsequent phases. Make all file changes before attempting to build the hardware design, as any partially updated set will cause many compile errors.

Much of the documentation below contains descriptions of how the various functions work, which is critical information for implementing the project. Sections which describe specific changes which must be made to the files are denoted by **[ACTION]**.

* 1. **ca/resources/ca\_resources.codal**

The file ca/resources/ca\_resources.codal defines every signal and register used in the design. The existing file includes examples of:

1. The stage definitions created in the “pipeline” event, which is a special Codasip function. This creates the five stage names which are used by other Codasip functions throughout the design.
2. The stage divider comments. It is valuable to keep the signals and registers divided by stage.
3. The pipeline register r\_pc was used in the IA Model and is defined in share/resources/arch.codal, so there is no definition for it in ca\_resources.codal.
4. A pipeline register definition for r\_id\_pc. Because this is a pipeline register, it must include the pipeline assignment with the correct pipeline specifier. The pipeline function allows Codasip to manage stalling and clearing pipeline registers in a simple and straightforward way. The number in brackets after “bit” is the number of bits in the register. We have defined ADDR\_W in the share/include/config.hcodal file as the width of any address (such as the PC) to be 32. Using defines instead of constant values makes the code more portable and is good engineering practice. Note that the pipeline registers are defined in the stage where they are pipelined (e.g. r\_id\_pc in the ID stage) but are shown in the schematic page of the previous stage (e.g. r\_id\_pc is on the IF stage schematic page).
5. A signal definition for s\_id\_instr, which is of width INSTR\_W (also 32 bits). Note that a signal cannot have a pipeline assignment.
6. A signal definition for s\_id\_aluop, and a corresponding register definition for r\_ex\_aluop. These signals have the special width ALUOP\_W. This is not defined in share/include/config.hcodal, but is automatically calculated in ca/define/defines.hcodal which is created in section 3.3 below.
7. A set of special pipeline control signals, which are all of width 1. These signals are used with special Codasip pipeline control functions which are not shown in the schematic, and are described in more detail in section 3.6 below. There is one special signal r\_id\_clear which is a register instead of a signal. This is a register which is initialized to a value of 1 (true) whereas all other registers initialize to a value of 0. The usage of this signal is described in section 3.8.1 below.

**[ACTION]** Add every signal and register from the schematics into the appropriate stage area. Data path signals and registers like s\_ex\_alu and r\_ex\_reg1 will use the width WORD\_W. Control signals like s\_id\_regwrite and s\_id\_alusrc2 will have their widths defined in section 3.3 below, so these can be left blank for now. Note that additional signals will be added to this file in subsequent sections of this document. Note that every signal must have the correct width as shown by the bitfield definitions in the schematics, or as full 32-bit signals if bitfields are not specified. Any necessary constant will be defined in share/include/ca\_resources.hcodal.

* 1. **ca/events/ca\_main\_reset.codal**

The file ca/events/ca\_main\_reset.codal contains the main code executed by Codasip, which calls all of the other code. Two special Codasip events are defined here – “reset” and “main”. The “reset” function is executed when Codasip starts or restarts, and simply prints a message if the info variable is defined as INFO\_GENERAL. The “main” function is then executed, and it performs the following operations:

1. Execute the code specified in main, which calls a function for each pipeline stage (the functions wb(), me(), ex(), id() and fe() defined in the pipe stage files defined below) and then a final function pipeline\_control which handles stalling and clearing of the pipeline. NOTE: the event for the IF stage is called “fe” because “if” is a reserved word in Codasip. As the code in each pipeline stage is executed, the value for each signal is updated whenever it is assigned a value by the code, and the next state of each register is calculated but the register value itself is not updated. The five pipeline stages must be executed in the reverse order (wb -> me -> ex -> id -> fe) so that control signals can be correctly passed backward (right to left) in the pipeline. Note that only register values can be passed forward (left to right). Remove the comments to enable the pipeline functions to be called.
2. After all of the pipeline stages are executed, the pipeline control() event is executed, which can modify the inputs to any of the pipeline registers using special Codasip functions.
3. Once all of the code in main has been executed, all of the registers (whether they are specified to be in a pipeline or not) are updated simultaneously with their next state value. This is the completion of one clock cycle.
4. The tool then loops back to the top of the main routine and executes the next clock cycle.

The general structure of ca\_main\_reset.codal will be used in the pipeline files. The event main, for example, calls a number of events defined in other places, so they must be referenced with the “use” function before being called.

**[ACTION]** Remove the comments in ca\_main\_reset.codal so that all of the pipeline functions are called. Make sure to preserve the order of the pipeline calls.

If any functions or variables are used in a function but defined in another file, that file must be included with a #include statement. A common cause of a build failure is a missing include file, especially one of the header files in share/include or ca/include.

This file is the CA equivalent of the ia/events/ia\_main\_reset.codal file.

* 1. **ca/include/ca\_defines.hcodal**

The file ca/include/ca\_defines.hcodal creates enumerated values for many of the control signals. Using enumerated values instead of constants makes the Codal code much more readable, and enables the easy creation of additional functions as the processor architecture evolves. There are two components for each control signal, which are included as an example for the aluop control field.

1. **[ACTION]** Create the enumeration for the signal in the appropriate stage area of the Enumerations section. The enum directive defines the name of the control signal, and then each possible value for the control signal is listed. The example shows the field “aluop”, and in Phase 5 we need only the ALU\_ADD and ALU\_NOP choices (more will be added in subsequent phases). Codasip will assign the actual values in the order the enums are specified, beginning with 0 and incrementing for each subsequent value. It is important that the ALU\_NOP choice is first so that it receives the value 0, since this is used for clearing the pipeline registers. Make sure the default selection is first in all cases.
2. **[ACTION]** Define the width of the signal (in this case ALUOP\_W) by defining it in the Signal Size section. The width is created using the special Codasip function bitsizeof, which is the log2 of the number of entries in the enum. In this case ALUOP\_W will be 1 since there are 2 enum entries. The width ALUOP\_W is used in ca\_resources to define the control signal (see section 3.1).

The advantage of creating this type of structure is that additional aluop selections may be created simply by adding more choices to the enum. The width of the control signal will be adjusted automatically.

**[ACTION]** Add an enumeration and size selection for each of the other control signals produced by the DECODER on Page-2 of your schematic. Name the enums so that they clearly identify the function to be performed. It is recommended to use a common string for each control group (like ALU), followed by an underscore and the specific selection.

Note that ca\_defines.hcodal has some auxiliary definitions at the end, which define some constants used in the pipeline. ca\_defines.hcodal must include the #ifndef, #define and #endif declarations which are in the example file.

* 1. **ca/decoders/ca\_decoder.codal**

Because the instruction decoding operation is so critical to any processor architecture, a special file ca/decoders/ca\_decoder.codal is used to create the DECODER. This file implements the instruction group decoding which is defined in the share/isa/isa.codal file, so the two files will have a very similar structure. There should be an element in ca\_decoder.codal corresponding to each group element in isa.codal. Since all names must be unique, the ca\_decoder.codal element should be named the same as the isa.codal element with “\_hw” added after the initial i. For example, i\_alu in isa.codal corresponds to i\_hw\_alu in ca\_decoder.codal.

In Phase 5, which only implements the ADD, ADDI and HALT instructions, only three elements are required, corresponding to i\_alu, i\_alu\_i and i\_halt in isa.codal. Examples of i\_hw\_halt and i\_hw\_alu are included in the original project, so add i\_hw\_alu\_i which will be very similar to i\_hw\_alu. Several things to note about these structures, which will be expanded significantly in subsequent Phases:

1. **[ACTION]** Every signal which is an output of the DECODER must be assigned in EVERY element of ca\_decoder.codal, even if the output value is DONT\_CARE. Each time a new decoder output is created, it must be assigned a value in every element, and a new element must assign every decoder output.
2. **[ACTION]** The example includes constants for the control signal values. If different names were assigned in ca\_defines.codal, use them here.
3. The DECODER is passed the value “opc” which includes all of the instruction fields necessary to decode any instruction. Each element uses a subset of the opcodes, and the subsets (such as opc\_alu) are defined in isa.codal.
4. **[ACTION]** Each element must have the assembler and binary function calls, which always have the parameter opc.
5. The signals are assigned values in the semantics section. Note that when an element uses more than one instruction, a switch statement on opc is used for signals which get different values for different instructions.
6. Each switch should include a “default” case which calls the codasip\_fatal function, stopping the simulation and printing an error message including the first parameter (ALU\_ERROR in the i\_hw\_alu example for s\_id\_aluop). The first parameter value is printed as a number, and the value of ALU\_ERROR is defined in debug.hcodal. Assigning different values to each default case will make it easier to determine what has failed.
7. The top level function of the DECODER is “dec”, which must be set to include all of the implemented groups. This corresponds to the riscv\_isa function in the IA model.

Note that Codasip is very helpful in identifying typos and missing elements in the files. Some examples from the initial ca\_decoder.codal file:

1. In the set function, i\_hw\_alu is initially in a bold blue font, indicating that its element exists. i\_hw\_alu\_i is in a normal black font, indicating that this element does not exist. Once the i\_hw\_alu\_i element is added, the font will switch to bold blue.
2. Signals and registers which have been defined in ca\_resources.codal are indicated in a reddish brown font, while undefined signals and registers are in a normal black font.
3. Enumerated constants which have been defined in ca\_defines.hcodal or debug.hcodal are in a blue italic font (like ALU\_NOP, ALU\_ERROR and ALU\_ADD), while undefined constants like WE\_TRUE and WE\_FALSE are in a normal black font. This indicates that these enums need to be added to ca\_defines.hcodal.
4. Many types of syntax errors cause a red bomb or a yellow question mark to be displayed in the gray area to the left of a line number. Hovering over the symbol gives a cryptic message about the error.
   1. **ca/others/ca\_utils.codal**

The file ca/others/ca\_utils.codal contains the print\_pipeline function, which is very handy in observing the flow in the pipeline. This function is called in the pipeline\_control function, and prints the current instruction and PC value in each of the pipeline stages ID, EX, ME and WB (in the IF stage the instruction has not been fetched yet so that stage is not interesting). The print is enabled by setting info to INFO\_PIPE (which is 6) – see section 6.6 for information on the info variable.

No changes need to be made to the ca\_utils.codal file.

* 1. **ca/pipelines/ca\_pipe\_control.codal**

The remainder of the files to be modified define the pipeline itself. ca/pipelines/ca\_pipe\_control.codal defines the function pipeline\_control() which includes operations which are applied to all of the pipeline register inputs after the actual pipeline code is executed, just before the register values are updated from the inputs at the end of the cycle. The code in ca\_pipe\_control.codal implements two important pipeline functions:

1. Stalling, which causes a pipeline register to retain its current value and not be updated with the next value.
2. Clearing, which causes a pipeline register to load zeroes and not the next value.

The RISC-V architecture uses stalling and clearing in several places, and the automatic Codasip structure allows these functions to be easily implemented. Note that the stall and clear functions are not shown in the schematics.

These functions are controlled by two signals, s\_STG\_stall and s\_STG\_clear (note that only stall is used in the IF stage). If one of these signals is asserted, the special Codasip functions pipe.STG.stall and/or pipe.STG.clear are called. The functions operate independently for each pipeline stage.

**[ACTION]** Create pipeline\_control by first entering the data in Figure 1 below the copyright notice. This calls the print\_pipeline function defined in ca\_utils.codal (see section 3.5) which can selectively print the pipeline contents. The next function checks the s\_if\_stall signal and stalls the IF pipeline stage if it is asserted.

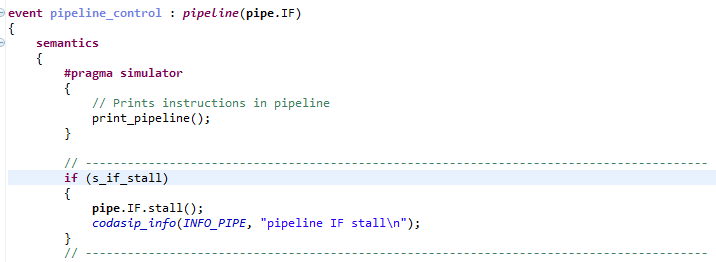


Figure 1

**[ACTION]** Add the code shown in Figure 2. This implements both stall and clear for the ID stage. Copy this code for the EX, ME and WB stages, changing the signal name, pipeline selection and message as appropriate.

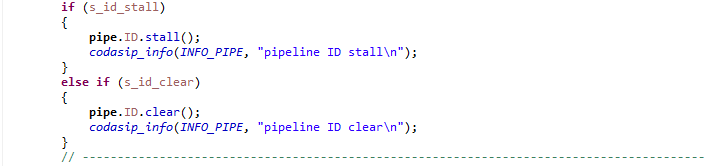


Figure 2

**[ACTION]** Add the required terminating braces at the end of the file, each with a semicolon.

* 1. **ca/pipelines/ca\_pipe\_stage1\_if.codal**

The file ca/pipelines/ca\_pipe\_stage1\_if.codal contains the logic for the IF pipeline stage, which should be taken from schematic page 1. The code should contain functions corresponding to each component of the schematic, plus some additional functions specific to special Codasip operations.

The code for each pipeline stage defines an overall event for the stage, in this case the event fe (because “if” is a reserved word in Codasip). The overall event implements all of the combinational functions within the stage. This then calls one other function: STG\_output (if\_output in this case), which assigns values to all of the registers in the stage (with register names in the next stage).

The schematic should include three components, each of which must be implemented in the semantics section of the code.

Note that Codasip executes all of its code in the order in which it is called. Within any individual file, FORWARD REFERENCES (using a value which is assigned in a later statement) MUST BE AVOIDED or unexpected behavior will occur. The example below shows a forward reference – the value of s\_ex\_signal2 used in the first statement will be the value BEFORE the second statement is executed. This is different from the behavior in Verilog and VHDL.

s\_ex\_signal1 = s\_ex\_signal2 + s\_ex\_signal3;

s\_ex\_signal2 = s\_ex\_signal4;

* + 1. **Instruction Memory Address Part**

**[ACTION]** Add the Instruction Memory address part as shown in Figure 3. The Instruction Memory is defined as if\_code in share/resources/interface.codal, which is why that label was added to the schematic. The “transport” function is the standard Codasip way to access this component. There are six parameters to this call:

1. The part of the memory being accessed (Address or Data). Two predefined Codasip constants CP\_PHS\_ADDRESS and CP\_PHS\_DATA are provided. Since this is the Address part, use CP\_PHS\_ADDRESS.
2. The status value returned. Since status is not used at this point, we define a 2-bit variable “status” to receive the result.
3. The command or operation. Three predefined Codasip constants CP\_CMD\_READ, CP\_CMD\_WRITE and CP\_CMD\_NONE are provided. The Instruction Memory is read only, so CP\_CMD\_READ is used in this case.
4. The base address, which must be aligned on a 4-byte (word) boundary. In this case the address is the PC register r\_pc, which is naturally aligned to the correct boundary.
5. The byte offset from the base address, which in this case must be 0 since all instructions reads are aligned to a word boundary.
6. The length of the access in bytes. Use the constant INSTR\_SIZE (which is assigned to 4 in config.hcodal).

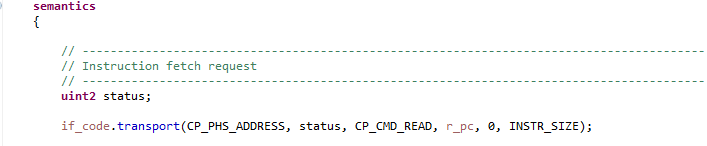


Figure 3

* + 1. **PC Adder**

**[ACTION]** Add the function which adds 4 to the PC to produce the next PC value as shown in Figure 4. Note that the constant INSTR\_SIZE is used here for flexibility. This shows the standard format for assigning a value to a signal like s\_if\_nextpc. Assign s\_if\_nextpc to s\_if\_pcin in another statement, as we will later insert logic between them.

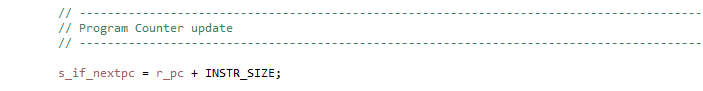


Figure 4

* + 1. **Register Updates**

**[ACTION]** In the semantic section of the if\_output event, add assignments for all of the registers as shown in Figure 5.

A screenshot of a cell phone

Description automatically generated

Figure 5

* + 1. **Pipeline Control Functions**

The example includes a set of signal assignments which are used for pipeline control. The clear functions for all the stages are included in the IF stage for easier usage later. The stall function for each stage is included in that stage. Note that the one clear signal (r\_id\_clear) which is a register is set in the if\_output event (see section 3.7.3).

* 1. **ca/pipelines/ca\_pipe\_stage2\_id.codal**

The file ca/pipelines/ca\_pipe\_stage2\_id.codal contains the logic for the ID pipeline stage, which should be taken from schematic page 2. The code should contain functions corresponding to each component of the schematic, plus some additional functions specific to special Codasip operations.

* + 1. **Instruction Memory Data Part**

**[ACTION]** Add the Instruction Memory data part as shown in Figure 6. This uses the if\_code.transport function just like the address part (see section 3.7.1), but as this is the data phase there are only three parameters:

1. The data part is selected by CP\_PHS\_DATA.
2. The response output is not used, so create a 3-bit variable response to receive the result.
3. The output data is the third parameter, which according to your schematic should be s\_id\_instr. However, due to the specific behavior of the Codasip memory model we must add a special function to create this.

The variable id\_instr\_temp is created to receive the memory output. This will be the desired instruction in every case except for the first cycle, which will be the default memory value 0 because the address cycle has not been completed due to pipelining. In the RISC-V architecture an instruction of all zeroes is illegal, so we force the instruction output to a real NOP, which in RISC-V is typically the instruction addi x0, x0, 0.

The value r\_id\_clear is initialized to 1 (see section 1 item 7)) and is normally set to 0 in each cycle, so it is 1 in the first cycle. The constant NOP\_INSTRUCTION (which is addi x0, x0, 0) is selected for the real instruction signal s\_id\_instr if r\_id\_clear is 1, otherwise the memory output will be used for s\_id\_instr.

This code uses one common multiplexor structure which is A = B ? C : D. This function means that if the value B is true (or 1) the signal A will get the value from C, otherwise it will get the value from D.

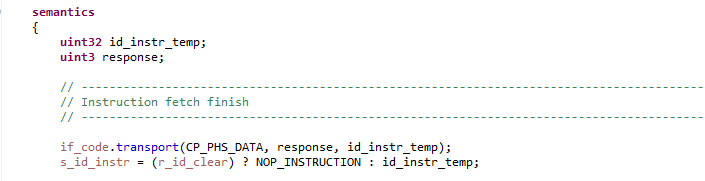


Figure 6

* + 1. **Instruction Parsing**

**[ACTION]** One key function of the ID stage is to extract a number of important fields from the instruction s\_id\_instr. The schematic shows several of these fields, so add these signals as shown in Figure 7. Several interesting elements of the Codal syntax are shown here.

1. The extraction of a bit field is indicated by the double period (NOT by a colon as in C). Note that if a 1-bit field is to be selected, the bit must be repeated (e.g. s\_id\_instr[7..7]). A single index will result in an array syntax error.
2. Concatenation of fields to a signal field is indicated by the double colon. The s\_id\_opc assignment concatentates the three instruction fields (FN7, FN3 and OPCODE) into a single opc value.
3. The s\_id\_immed assignment shows how sign extension of an immediate field is implemented. s\_id\_instr[31..20] extracts the 12-bit I-type immediate field as an unsigned integer, (int12) converts that to a signed 12-bit value, and (int32) then expands that to a 32-bit signed value which is required for use as an ALU input. This is the initial implementation of the IMMGEN function, which will be expanded in later Phases.

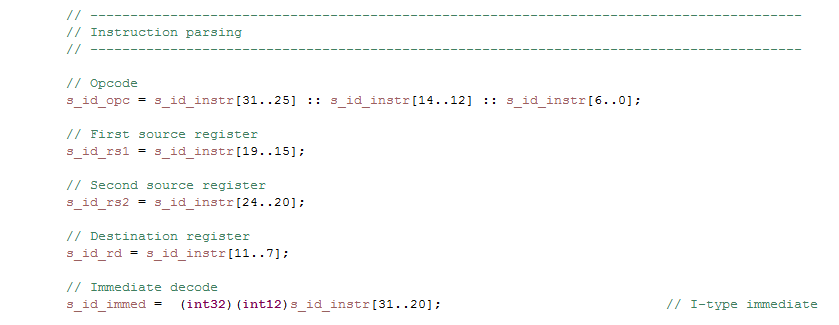


Figure 7

* + 1. **Register File Read**

**[ACTION]** The Register File rf\_xpr is defined in share/resources/arch.codal and is basically an array of 32 32-bit registers, so reading it is simply accessing the array based on the source select fields, as shown in Figure 8.



Figure 8

* + 1. **Access the Decoder**

The decoder is defined in ca\_decoder.codal as the function dec. The example shows how to use this, and it is called with the s\_id\_opc field as the instruction input parameter. All of the decoder control outputs are assigned in the dec function and therefore exist in the ID stage without being explicitly defined there.

* + 1. **Register Assignments**

**[ACTION]** Add all of the register assignments in the semantics section of id\_output event based on the schematic.

* + 1. **Include Files**

**[ACTION]** Some header files will need to be included. This is typically determined by observing undefined references when trying to build the project. Header files include ca\_defines and several files in share/include.

* 1. **ca/pipelines/ca\_pipe\_stage3\_ex.codal**

The file ca/pipelines/ca\_pipe\_stage3\_ex.codal contains the logic for the EX pipeline stage, which should be taken from schematic page 3. The code should contain functions corresponding to each component of the schematic.

* + 1. **Source 2 Multiplexor**

**[ACTION]** Multiplexors are one of the most common elements in any processor, and the RISC-V design will eventually include a number of them. For Phase 5, only a single multiplexor is required, as shown on schematic page 3. This selects between the Register File reg2 output (for ADD) and the I-type immediate value (for ADDI) as the source 2 input to the ALU. The selection is made using the r\_ex\_alusrc2 control signal. There are at least four ways to implement a multiplexor in the Codal language.

* + - 1. **Switch Statement**

The clearest implementation of a multiplexor from a documentation standpoint is with a switch statement as shown in Figure 9. This is the recommended implementation in this case, and it is easily expanded to more than 2 inputs. The selection choices are defined in ca\_defines.hcodal. Note that the codasip\_fatal error code is the special SRC2\_ERROR value.

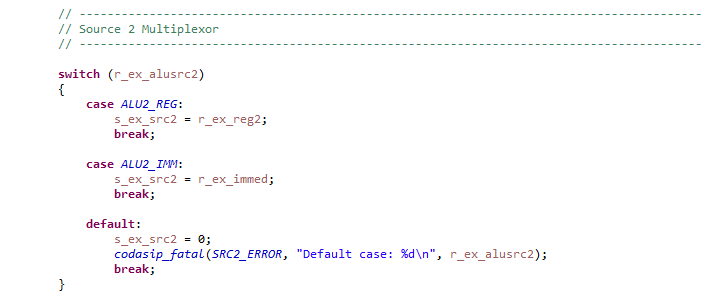


Figure 9

* + - 1. **If/else Structure with Default**

An if/else structure can also be used to implement a multiplexor, as shown in Figure 10. This implementation has all the capability of the version above but isn’t quite as readable.

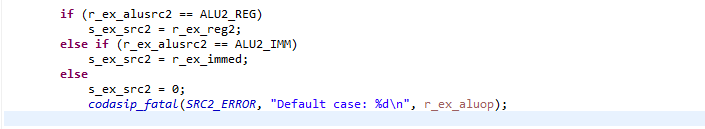


Figure 10

* + - 1. **If/else Structure with Boolean Select**

In some cases, especially when there are only two choices, the select signal can be a Boolean value (true/false or 1/0) which is not assigned selection values in ca\_defines.hcodal (and is not a decoder output). In this case the simpler structure of Figure 11 below may be used (where r\_ex\_alusrc2 is true or false), and there is no need for a default choice.

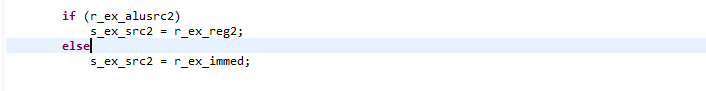


Figure 11

* + - 1. **Decision Structure**

The most compact implementation of a multiplexor is the decision structure shown in Figure 12. This approach also requires that the selection signal is a Boolean.



Figure 12

* + 1. **ALU**

**[ACTION]** The ALU implements a set of operations on two inputs based on the r\_ex\_aluop value. The basic structure is similar to a multiplexor, but in this case the switch implementation is necessary as shown in Figure 13 because several additional functions will be added later.

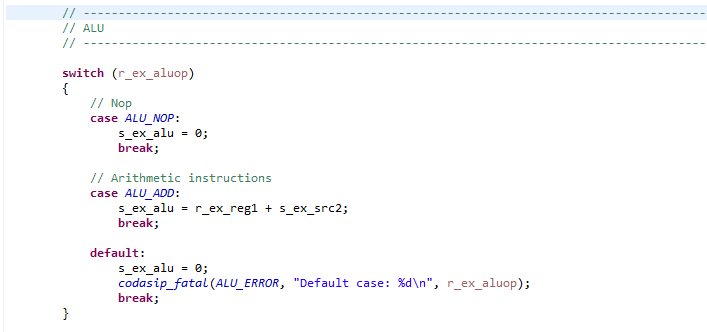


Figure 13

* + 1. **HALT**

The special HALT instruction, which is necessary to stop simulations correctly, is included in the example code. The GPR\_RETVAL register is used by the C Compiler. The HALT function is not shown in the schematic, but the required r\_ex\_halt signal was created in the DECODER in the ID stage.

* + 1. **Register Assignments**

**[ACTION]** Add all of the register assignments in the ex\_output event based on the schematic.

* + 1. **Include Files**

**[ACTION]** Some header files will need to be included. This is typically determined by observing undefined references when trying to build the project.

* 1. **ca/pipelines/ca\_pipe\_stage4\_me.codal**

**[ACTION]** The file ca/pipelines/ca\_pipe\_stage4\_me.codal contains the logic for the ME pipeline stage, which should be taken from schematic page 4. The code should contain functions corresponding to each component of the schematic. In this case only the register assignments and possibly header includes are necessary.

* 1. **ca/pipelines/ca\_pipe\_stage5\_wb.codal**

**[ACTION]** The file ca/pipelines/ca\_pipe\_stage5\_wb.codal contains the logic for the WB pipeline stage, which should be taken from schematic page 5. The code should contain functions corresponding to each component of the schematic. In this case only the Register File write function is required, which is shown in Figure 14. This checks the register file write enable, and writes the result data to the register selected by the rd field. Note that there is no wb\_output event as there are no pipeline registers.

A screenshot of a cell phone

Description automatically generated

Figure 14

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. Note that the warning shown in Figure 15 is not a problem, as we will add the Data Memory in a later Phase. Other warnings are unacceptable and must be corrected.



Figure 15

Build Simulator (ca) in the Task window, and again correct any syntax errors. Note that every time any file in the design is changed, all of the hardware pieces (like Simulator (ca)) must be rebuilt so that they all have the small black check mark. Run -> Debug Configurations will not get an error if this step is omitted – it will simply use the old version of the SDK which is unlikely to be the desired behavior.

Note that many errors and warnings provide an error code in the Console window. Clicking on the error code opens a window into the Codasip Help system describing the error.

Build Assembler (ia) so that you will have a working software SDK.

1. **Run the Test Program**

Once both the Model Compilation (ca) and Simulator (ca) tasks finish successfully, the next step is to Import the test program from G:/Information/Phase 5/phase5\_test, which is very similar to phase1\_test (the one used in Phase 1). Change the SDK to standardname5.ia and build the test.

Execute Run -> Debug Configurations as in Phases 1/2/3. The C/C++ Project should be phase5\_test, and the Application should be Debug/phase5\_test.xexe. For the debugger, select standardname5.ca.standardname5-ca-isimulator. If this is not a choice, the project build process has not completed successfully. Note that this is NOT the ia version of the SDK.

The running and stepping controls are the same as for the IA model described in Phase 1. However, because the hardware now includes the real pipeline stages, the behavior and debug methods are quite different. Section 6 contains a number of descriptions which will aid in debugging the CA Model.

1. **Debugging Information**

This section includes several useful descriptions of Codasip behavior and tips for debugging.

* 1. **Not Combinational Logic**

A frequently occurring error when building Simulator (ca) is HLSW9, with the message that the unit (usually one of the pipeline stages like ex()) should be combinational logic. This means that Codasip has identified a combinational loop, in which a signal feeds back to itself without going through a register, and is often caused by typos of signals. In most cases this is caused because a signal is assigned to a value in more than one statement, which is not allowed. Copy-paste without correctly modifying the signal names is a frequent source of this error.

* 1. **Additional Display Window Functions**

New options now exist in the Display window. In the Registers tab, in addition to the Register File registers which are observable under Architectural Registers, every register declared in ca\_resources.codal is now visible under

Microarchitectural Registers/codasip\_top\_level/standardname5.

Unfortunately the order of the registers is selected by Codasip and cannot be changed. The Number Format can be set independently for each register.

Every signal declared in ca\_resources.codal is visible in the Signals tab, in codasip\_top\_level/standardname5. As with the Registers, the order of display is selected by Codasip. An additional issue is that signals may be displayed only in decimal format, although if a signal is selected all of its formats are shown in the lower part of the window.

* 1. **Hardware Breakpoints**

In addition to the software breakpoints which can be set as described in Phase 2, the CA Model also supports hardware breakpoints. To enable this function, double click on the codal.conf file which is the very last file in the project. The codal.conf file will be opened in the Editor window. Select Simulator in the left column, and check the box to the left of CodAL debugging. NOTE: this box may already be checked. If it is, you don’t need to do anything else.

Setting a hardware breakpoint is exactly the same as setting a software breakpoint. Go to the desired line in one of the hardware files and double click in the gray area to the left of the line numbers. A small blue circle with a check mark indicates that a breakpoint is set. If run is clicked, the simulation will stop at the first breakpoint reached and display the Codal file containing the breakpoint. At that point, the current value of any signal or register may be observed simply by hovering over it with the mouse. Clicking a signal also displays the value.

The hardware breakpoints are listed in the Breakpoints tab of the Display window (along with the software breakpoints), and each breakpoint can be enabled or disabled there just like the software breakpoints.

* 1. **Simulation Behavior**

In order to effectively debug the CA Model, the simulation process must be understood. Codasip executes the CodAL code in order, just as a C program would be executed. Special Codasip functions perform some operations to simplify the simulation. The full process for one clock cycle is executed, and Codasip then repeats the process until a termination command is executed. The sequence is as follows:

1. Execution begins with the special function “main” defined in ca\_main\_reset.codal.
2. main calls a sequence of functions defined in the pipeline files. In each function, the value of any signal is updated when an assignment command to it is executed. For each register, the next state value is updated when an assignment command to it is executed, but the actual register value is unchanged.
   1. wb() in ca\_pipe\_stage5\_wb.codal is executed
   2. me() in ca\_pipe\_stage4\_me.codal is executed
   3. ex() in ca\_pipe\_stage3\_ex.codal is executed
   4. id() in ca\_pipe\_stage2\_id.codal is executed. This function calls:
      1. dec() in ca\_decoder.codal
   5. fe() in ca\_pipe\_stage1\_if.codal is executed
   6. pipe\_control() in ca\_pipe\_control.codal is executed. This function calls:
      1. print\_pipeline() in ca\_utils.codal
3. Once all of the functions in main have been executed, all registers are simultaneously updated from their next state values.
4. main loops back to the top of the file and repeats the process

Understanding this process is crucial when using hardware breakpoints. Because the execution proceeds as described above in order, when a hardware breakpoint is encountered the simulation is part of the way through a cycle, so some signals will have been updated for the current cycle and some will still have the value from the previous cycle. For example, if a breakpoint is set in the EX stage code, signals assigned in the WB and ME stages will have new values, and signals assigned in the ID and IF stages will have old values. In the EX stage, signals assigned above the breakpoint will have new values and signals assigned below the breakpoint will have old values. This behavior allows Codasip to simulate very quickly, but it is different from true hardware simulation environments like Verilog or VHDL where all signals are updated simultaneously.

This behavior also requires that any signal used in an assignment statement must have been set earlier in the cycle or unexpected results will occur. In the code example below, the value of s\_id\_signalA will use the old value of s\_id\_signalC instead of the new value, which is almost always not correct. This is a FORWARD REFERENCE and must be avoided. Unexpected behavior can often be traced to this type of error.

s\_id\_signalA = s\_id\_signalB & s\_id\_signalC;

s\_id\_signalC = s\_id\_signalD | s\_id\_signalB;

Since the pipeline stages are called by main in reverse order (right to left in the pipeline) signal assignments in any stage may use signals generated in any later (to the right) pipeline stage.

* 1. **Software Window View**

When a simulation starts, the Editor window shows the assembly language program, with a green highlight on the currently executing instruction. In the IA Model, after a step that instruction will be executed and the Register File will be updated. However, in the CA Model each step executes one clock cycle, but due to the pipeline each instruction requires 5 clock cycles to execute. This will complicate the debug analysis.

The instruction highlighted in green is the one in the IF stage, so the one before it is in the ID stage, the one before that is in the EX stage, etc. The Register File updates thus occur 5 cycles after the instruction is in the IF stage, and on the step that occurs when the instruction is in the WB stage.

* 1. **The “info” value**

Codasip maintains an internal variable called “info”, which is used to selectively enable a variety of debug print functions. The default info value is 0, which enables no printing. Info may be set to another value in the debug configuration:

1. Execute Run -> Debug Configurations.
2. Select the desired Project, Application and Debugger as defined in Phase 4.
3. Select the (x) = Arguments tab.
4. In the Simulator arguments area, enter “- -info N” (no quotes, no space between the dashes or between the dash and info) where N is a decimal number which is the desired value of info. Multiple values may be specified by separating them with commas (and no spaces), and all of the specified prints will occur.
5. Click Debug to run the simulation in the normal way.
6. The info value will remain as it is set whenever this debug configuration is run.

Figure 16 below shows an example which sets info to 6.

The info value is used by several functions, particularly codasip\_info and codasip print. The first parameter of these functions contains the info value which enables the print operation. The values for info are defined in debug.hcodal, and additional values can be added there as desired.

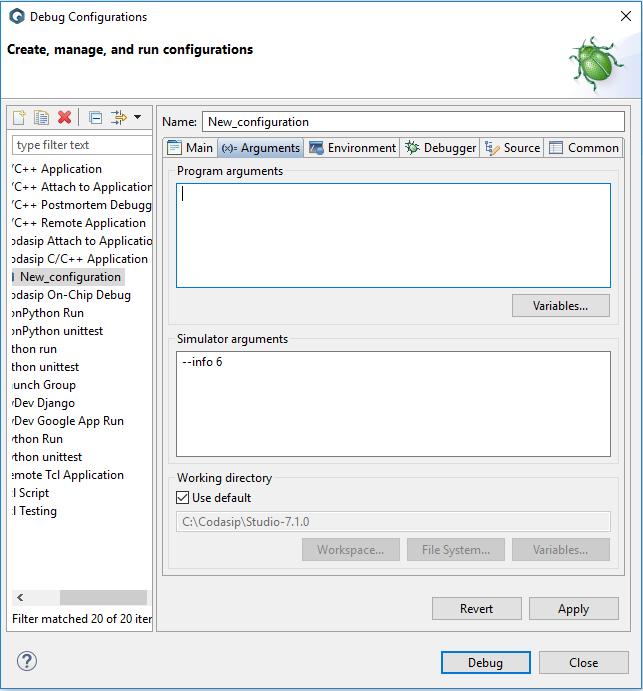


Figure 16

Useful info values in Phase 5 are:

INFO\_PIPE (6) – print the contents of the pipeline on each cycle

INFO\_RF\_WRITE (3) – print each value written to the Register File and its index

INFO\_ALU (7) – print the inputs and outputs of the ALU

See the Document “Debugging Codasip with - -info” for details on each info value.

* 1. **Comparing Files**

In many cases it is valuable to compare two files, especially since most of the project process involves updating files. To compare two files, select one of them in the Project Explorer, then hold down CTRL and select the other file. Right click either file and select Compare With -> Each Other. A new editor window will open which shows both files and highlights the differences. Either file can be edited in this window.

* 1. **Disassembler Display**

In a number of cases it is very valuable to be able to see the actual assembly code which is executed, including the actual address of each instruction, the actual memory offset for memory references and the actual assembly instructions which replace pseudo instructions. This can be achieved by opening the Disassembly Window. This requires the following steps:

1. Build the Disassembler (ia) in the project where the other IA functions are built.
2. Initiate a simulation run, switching to the Debug Perspective.
3. Click on the i-> button as shown in Figure 17.

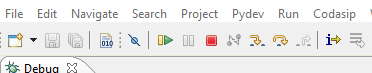


Figure 17

1. The Disassembly window will then open to the right of the Editor window as shown in Figure 18. The left column is the instruction address in hex. Note, for example, that the pseudo instruction “nop” at address 0000000c is displayed as “addi x0, x0, 0”, the actual instruction.

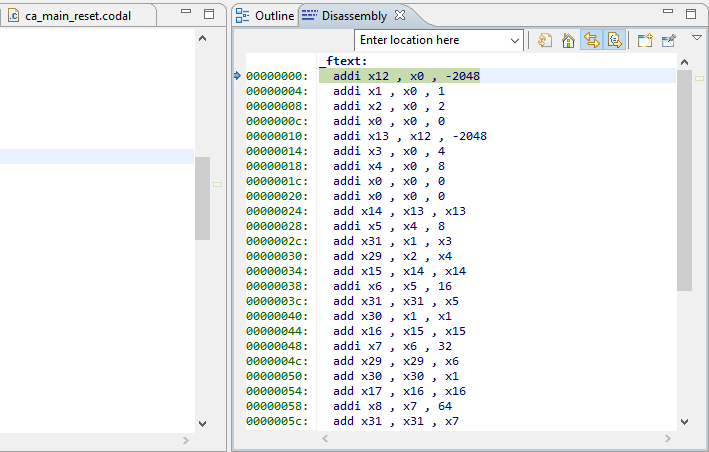


Figure 18

* 1. **Decoder Failure at Initialization**

An error which may occur is a decoder error as shown in Figure 19. 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 20. 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 19. When the simulation is running normally, the console view will be the “New configuration [Codasip C/C++ Application] phaseX\_test.xexe” view, where any normal print messages will occur.

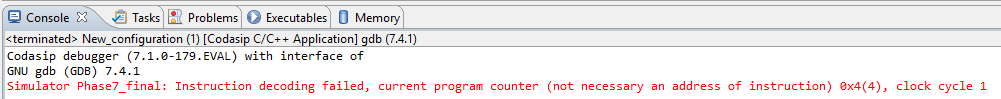


Figure 19

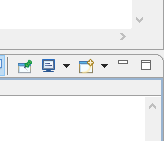


Figure 20

Because the current program counter is 4 and the clock cycle is 1, this error has occurred on the attempted decode of the instruction fetched from the Instruction Memory on the first cycle. This function is defined in section 3.8.1, and this error will occur if NOP\_INSTRUCTION is not correctly defined or the behavior of r\_id\_clear is not correct (i.e. it is not initialized to 1). This will cause id\_instr\_temp (which is 0) to be selected as the instruction, and 0 is an illegal value in the DECODER.

If NOP\_INSTRUCTION and r\_id\_clear are correct, this error can also occur because the definition of the s\_id\_opc value passed to the decoder is not correct, so that the dec() function has received an instruction which does not have a definition within ca\_decoder.codal. One way to debug this error is to look at the signal s\_id\_opc as the simulation is stepped, or to set a breakpoint on the dec(s\_id\_opc) function in ca\_pipe\_stage2\_id.codal.

If this error occurs after the first cycle, the decoder has an error. Determine the s\_id\_opc value which causes the error and then identify where it should be decoder in ca\_decoder.codal.

1. **Scoring the Project**

The project should be submitted when the test program phase5\_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). The test must run completely to the end of phase5\_test with only a single execution of run (the green arrow) and all of the registers must contain the values defined in the comments at the beginning of phase5\_test. Unsuccessful submissions will be rejected. The score for a successful submission will be determined by the time of submission relative to the Target Date.

1. **Exporting the Project**

The project should be Exported and submitted as standardname5.zip, in the folder G:/Submissions.