**RISC-V Class Project Phase 2 – Full Instruction Accurate Model**

In this Phase of the Class Project you will expand the Instruction Accurate model of the RISC-V processor created in Phase 1 to the full instruction set. This model simulates the processor’s behavior on an instruction-by-instruction basis, with no assumptions about the actual hardware implementation. Each instruction executes completely and updates the register file or memory before the next instruction begins. This creates a useful environment for understanding the instruction set behavior and creating the tools necessary for building any test software.

This Phase starts with the model files created in Phase 1, which implements three RISC-V instructions – ADD, ADDI and HALT. Several files are then modified in order to create the full instruction set version. A simple test program will be provided to verify that the changes are basically correct.

1. **Create the New Project**

Copy the “standardname1” project from the completion of Phase 1 to a new project named “standardname2”. It is valuable to retain the Phase 1 project for comparison until Phase 2 is completely working. The files in the model/ia and model/share folders will be modified to create the Instruction Accurate model.

1. **Understand the RISC-V Instruction Set**

The IA Model implements the basic RV32I instruction set, so a clear understanding of this is critically important. Chapter 2 of the textbook contains a complete description, and Figure 1 shows the instruction set summary from the RISC-V Instruction Set Manual. The 37 instructions shown will all be implemented. The instruction set will be described in several class sessions.

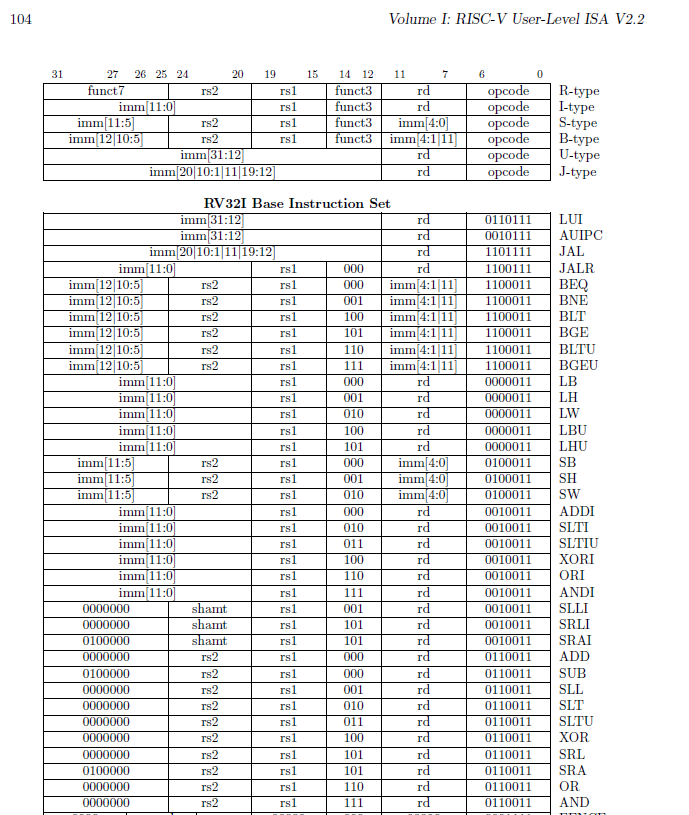


Figure 1

1. **Add Instructions to the Files**

We will now edit each of the files necessary for the IA model to add the remainder of the instructions. Each file will be described in this section, although some files will not be modified at this point. Note that nothing can be built until all of the files have been modified, as they often depend on each other.

Each of the files used by the process is described in detail in the sections below. This provides a handy reference if information is needed about these files, as the Codasip documentation is extremely difficult to navigate and contains a huge amount of information which is not relevant to this project. Note that there are descriptions for several files which will not be modified in Phase 2 – this will be indicated at the end of each section. It is important to understand the function of each file even if it won’t be modified in Phase 2.

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

This file includes two processes, main and reset. The reset process is executed when a simulation is initiated or reset. The main process is the main execution process of Codasip. The register “r\_pc” is the program counter (PC). The instruction at the program counter is fetched into r\_instruction\_buffer from the if\_code memory, which will have been loaded with the test program when the simulation was initiated. Some debug statements allow for printing information about the instruction fetched. The PC is incremented by the instruction size in bytes, which is 4 in this case. The instruction is then passed to the process riscv\_isa which executes it (and which may modify r\_pc). The process then loops back to the beginning and fetches the instruction at the new value of r\_pc. This process continues until a halt or error is detected.

The statement “codasip\_print(INFO\_GENERAL, …)” will print the information in the second parameter if the info\_level matches the first parameter (see Section 3.3 below). The printed line follows normal C formatting, and variables may be included and referenced with %(format).

The statement “codasip\_disassembler(INFO\_GENERAL, r\_instruction\_buffer, r\_pc)” will print the disassembled instruction in the instruction buffer, with any addresses calculated based on the PC, if the info\_level matches the first parameter (see Section 3.3 below).

For Phase 2, this file does not need any modifications.

* 1. **share/include/config.hcodal**

This header file contains many pre-defined constants used to make the code clearer and more readable. Many of these constants are used in the CA (Cycle Accurate) model only.

For Phase 2, this file does not need any modifications. New constants may be added at the end of the file at any time.

* 1. **share/include/debug.hcodal**

This header file contains some basic debug constants. The “info\_levels” enumerated values are used to control various levels of debug printing. These debug printing functions are enabled by entering “- -info #” (no space between the dashes) in the Simulator Arguments section of the Arguments tab in the Debug Configurations dialog box, where # is the info\_level. The print statements are typically in the form of a codasip\_print – see the example in Section 3.1. The info\_level constants are defined here, so for example when INFO\_GENERAL is the codasip\_print parameter, the print is enabled with - -info 3. Note that only one info\_level can be specified at any one time.

The fatal\_codes section defines a number of codes which are used in debug statements.

For Phase 2 this file does not need any modifications. Additional debug info\_levels may be added here at any time to facilitate debugging, and more standard levels will be added in subsequent Phases.

* 1. **share/include/utils.hcodal**

This header file defines some important functions.

DEF\_OPC – this function defines an instruction opcode in preparation for the instruction definitions in share/isa/isa.codal.

DEF\_XPR – this function defines a register in the general purpose Register File.

For Phase 2, this file does not need any modifications.

* 1. **share/resources/arch.codal**

This file defines the program counter register r\_pc and the general purpose Register File rf\_xpr. It also defines the address space for both the Instruction Memory and the Data Memory. Note that these two memories operate independently because RISC-V is a Harvard architecture, but they share the same address space.

This file also includes the definitions of the Control and Status Registers, which are not used in Phase 2.

For Phase 2, this file does not need any modifications.

* 1. **share/resources/interface.codal**

This file defines the actual interfaces to both the Instruction Memory and the Data Memory. These memories are accessed via a special Codasip structure called the Codasip Local Bus (CLB).

For Phase 2, this file does not need any modifications.

* 1. **share/isa/isa\_ops.codal**

This file specifies several important elements of the processor. The 32 DEF\_XPR calls define the 32 32-bit registers in the Register File. The other elements define the immediate values used in the RISC-V instruction set and some miscellaneous other attributes.

For Phase 2, this file does not need any modifications.

* 1. **share/isa/isa.codal**

This is the main file where the behavior of each instruction is defined. There is a separate element for each group of instructions, which defines several key functions for each instruction.

1. The opc value for each instruction is defined using the DEF\_OPC function.
2. All of the opc values for a particular group are assembled into the group set.
3. The special Codasip “assembler” function in the group element defines the format of an instruction in RISC-V assembly language.
4. The special Codasip “binary” function in the group element defines the actual RISC-V machine language code for the instruction.
5. The special Codasip “semantics” function in the group element describes the behavior of the instruction. This often uses the library functions defined in ia\_utils.codal (see Section 3.8).

In order to create the full isa.codal file, execute the following steps:

1. Uncomment sections by removing all of the global comments (/\* - \*/) except the one set in the Semantic Routines section at the end of the file. Remove ONLY the comments, and not the commented lines.
2. Two of the groups (i\_alu which contains the R-type instructions and i\_alu\_i which contains the I-type instructions) have been partially implemented to handle the ADD and ADDI instructions. In these two groups, remove the comments so that all of the instructions in the group are defined by DEF\_OPC and remove the comments so that the “set” in each case includes all of the instructions in the group.
3. Each of the other groups has an empty semantics section, which is:

semantics

{

ADD SECTION HERE

};

The “Semantic Routines” section at the end of the file contains some numbered sections which each contain the correct semantic function for an instruction group. Based on the required function of each group, copy the correct section to the lines between the braces in the semantics section of each group. Note that there will be leftover sections. The i\_alu and i\_alu\_i groups can be used as examples.

The semantic sections use several functions which are defined in ia\_utils.codal (see section 3.9).

1. Note that any group which has “\_alias” at the end of the name defines a pseudo instruction. These instructions are not real RISC-V instructions, but are mapped to one or more real instructions which makes writing assembly language easier and improves the behavior of the C compiler. Nothing needs to be done for the group definitions other than removing the comments for them.
2. In the “Instruction Set Architecture” section at the beginning of the file, remove the comments so that all of the groups, including the pseudo instructions, are included in the overall riscv\_isa set. Exceptions are i\_csr and i\_csr\_imm – retain the comments on those two lines.

* 1. **ia/other/ia\_utils.codal**

This file defines several special functions used for instruction execution. For each function, some examples are provided and the remainder must be filled in. Each function is described in a section below.

All global comments (/\* - \*/ pairs) should be removed prior to modifying the groups.

* + 1. **General Purpose Register File Access**

These functions implement reading and writing from/to the Register File. For Phase 2, these functions do not need any modifications.

* + 1. **Branching / Jumping to new address**

This function simply loads the PC with a new value. It is used for branch and jump instructions.

For Phase 2, this function does not need any modifications.

* + 1. **ALU**

This function defines the operation to be performed in the ALU for each RISC-V instruction. The inputs to the “alu” function are the instruction and the values of the rs1 and rs2 sources to the ALU. The function returns the result of the ALU operation.

The operation is a switch statement based on the opc value, which is unique for each instruction. Each opc value is defined by a constant of the form “OPC\_XXX”, where XXX is the instruction. These constant values are defined in share/include/opcodes.hcodal – see Section 3.10. All valid cases are listed. Below the cases are all of the required ALU operations (including the required break), and each case should select the proper operation for that instruction. Multiple cases can select the same function, so each function should only be used once. Note that the add operation is already defined in the example for the ADD and ADDI instructions, and other instructions may also use this function. DO NOT modify any of the functions or add any new ones – the available set is complete. There must be only 11 different functions utilized in this function. You may need to study the instruction functions to determine which function is correct for each one.

* + 1. **Load Function**

This function defines the operation to be performed for a memory load instruction. The inputs are the instruction and the address to be read. The function returns the value read form the address in memory, sign extended to a 32-bit value as specified in the RISC-V architecture.

The operation is a pair of switch statements based on the opc value. The first switch determines the number of bytes to be read from memory, and the actual read is implemented by the if\_data\_read function. The second switch then determines if the result needs to be sign extended to produce the return value.

In each statement, the choices are listed below the instruction select values, so each instruction should execute the correct operation. As described above, multiple case selections can select the same function, so each operation should be included only once.

* + 1. **Store Function**

This function defines the operation to be performed for a memory store instruction. The inputs are the instruction, the address to be written to and the data to be written. The function does not return a value.

The operation is a switch statement based on the opc value, which selects the number of bytes to be written. The actual memory store is implemented by the if\_data\_write function. Each function should be assigned to the correct case statement.

* + 1. **Data Interface Access**

These functions implement the actual read and write operations to the Data Memory. Once the global comments are removed, these functions do not need to be modified.

* 1. **share/include/opcodes.hcodal**

This is the header file where the instruction decoding is defined. A few examples are defined, and the values must be entered for all of the remaining instructions.

The opcode function produces a 17-bit value for each instruction which consists of the FN7 field (instruction bits 31:25), the FN3 field (instruction bits 14:12) and the OPCODE field (instruction bits 6:0). If a particular field for an instruction is not used, it is set to 0. All global comments should be removed before updating the rest of the instructions.

The first two sections create the values for FN7 and FN3 for each instruction which uses them. Based on the ADD and ADDI examples, the 7-bit (for FN7) and 3-bit (for FN3) binary values for each instruction should be entered, so that all of the FUNCT7 and FUNCT3 constants are correctly defined.

The Opcodes definitions sections create the actual instruction opc value. The uj\_type\_opcode enum is for the instructions (U-type and J-type, plus the already defined special instructions HALT and SYSCALL) which use only the RISC-V OPCODE field. Insert the appropriate opcode for each instruction where it is not defined.

The r\_type\_opcode enum is for the R-type instructions and the I-type instructions SLLI, SRLI and SRAI which use all three of OPCODE, FN3 and FN7. Based on the OPC\_ADD example, create all of the other constant values. Note that the OPCODE part of the instruction may be different for different types of instructions.

The isb\_type\_opcode enum is for instructions which use only OPCODE and FN3 (I-type, B-type and S-type). Based on the OPC\_ADDI example, create all of the other constant values. Note that the OPCODE part of the instruction may be different for different types of instructions.

1. **Build the Hardware Project**

Once all of the files have been updated and saved, repeat the build process from Phase 1 (Section 6). If you get error messages in the Console window, you have incorrectly modified one or more files, so correct those errors. Continue this process until the build process completes successfully and there are check marks next to Model Compilation (ia), Assembler (ia), Disassembler (ia) and Simulator (ia). Note that building either Assembler (ia) or Simulator (ia) will automatically build Model Compilation (ia), so that does not have to be built separately.

1. **Import the Test Program**

The next step is to Import the test program into the Project window, which is the phase2\_test project in the G:/Information/Phase 2 folder. Import this as a separate project, NOT into the hardware standardname2 project. DO NOT build the test program at this point – it includes the Binaries and Debug folders which contain a known good test built with a known good Assembler.

1. **Simulate the Test Program**

The next step is to simulate the test program. Follow the steps from Phase 1 (Section 9), but select the “Disable auto build” radio button in the Build section. The test program has passed if the simulation stops at the instruction after the halt instruction in line 701 (i.e. stops on line 702), Register File register x25 contains the value 1, and register x10 contains the value 57 decimal. If the test fails, the simulation will stop at a different location, and Register File register x10 will contain a value which indicates which test in the program failed. If this happens, there is an error in the file modifications made to the hardware in Section 0 which must be debugged.

1. **Debug the Test Program**

The structure of the phase2\_test program is designed to easily determine what is failing. Open phase2\_test/src/phase2\_test.s, and look at the start of the program. Register x10 is initially loaded with the value 1 (li is a pseudo instruction which loads a 12-bit immediate value into a register), and the code in “case 1” will execute and branch to the label FAIL if the instruction fails. If the instruction passes, x10 is incremented to 2 and “case 2” is executed. This continues until either there is a branch to FAIL or a branch to PASS. If the branch is to FAIL, x10 will contain the case number of the failing case. Note that multiple errors, in particular problems with the branch function, may cause incorrect behavior such that the case number does not actually correspond to the failure point. The warnings shown in Figure 2 can be ignored.

A screenshot of a cell phone

Description automatically generated

Figure 2

One good way to debug errors is to step through the program and observe the results as they are written into the Register File. When the value written is not what is expected (either the wrong value being written or a value written to the wrong register), the instruction just executed is not behaving correctly. The comment with each case statement will provide an indication of which instruction is being exercised in that case.

* 1. **Setting Breakpoints**

Once several of the cases are passing, it may be more efficient to set a breakpoint rather than stepping through the instructions one by one. Breakpoints can only be set if you are using an Assembler which has been created locally, so make sure you have built the Assembler in standardname2. Change the SDK of phase2\_test to standardname2.ia and build phase2\_test. Note that there will be three “Signed relocation” warnings which are OK. To set a breakpoint, go to the desired instruction and double click in the gray bar to the left of the line number. A blue circle will appear indicating that a breakpoint has been set on that instruction. Any number of breakpoints may be set, and they are remembered by Codasip and will be active the next time the program is simulated. Double clicking on a breakpoint will remove it.

One of the tabs in the Display view of the Debug Perspective is the Breakpoints tab, which has an entry for each breakpoint. Each breakpoint can be enabled or disabled independently, which allows setting several breakpoints but only enabling the one or two used for a particular debug operation.

1. **Verifying Memory**

If the test passes, the data in memory must also be correct. Open a Memory Monitor as described in the Memory Debug document which is in Canvas. At the end of the test, the value in memory location 0x7CC (1996 decimal) must be 0x37000000. If this is still 0x00000000 (or some other value) there is an error in the memory (load and store) operations.

1. **Run the Final Test**

The design must be verified by passing a test program which was not built with the created Assembler. Once the test passes the locally assembled version of phase2\_test, rename the phase2\_test project to a different name. Then Import phase2\_test again, but do NOT build it. The phase2\_test.xexe file there has been built with a known correct assembler. Run that test as described above, and if it passes (with the correct value in memory location 0x7CC) Phase 2 should be complete.

If that version of phase2\_test does not pass, there is an error in the implementation of one or more files. Debug as described above, but note that breakpoints cannot be set in this test version, so only stepping through the program will work.

1. **Scoring the Project**

The project score will be a function of when it is correctly submitted, with the correct project name and zip file name. Correctly submitted means that the test program which you Imported without building runs to completion with the correct register results. Incorrect submissions will be declined, with the goal being within 1 day, and they must be corrected and resubmitted to receive credit. There may be deductions for incorrect naming and other procedural errors.

The design elements like ia\_utils.codal and opcodes.hcodal will be checked for correctness, and designs may be rejected if any of the elements are incorrect. The project must be corrected and resubmitted with the correct elements to receive credit. In particular, ia\_utils.codal must implement the correct 11 ALU operations.

The Target Date for Phase 2 is Sunday, February 7 at 10:00 PM. The Bonus is 1%/day up to a maximum of 7%, and the Deduction is 10%/day after the Target Date.

1. **Export the Successful Project**

Once the test program passes, submit it for grading just as in Phase 1. To do this, return to the Codasip Perspective, right click the hardware Project standardname2 in the Project Explorer and select Export. Select General -> Archive File and select Next, which will produce the dialog box shown in 2 of Phase 1. Expand the Project, and uncheck the “work” folder. This must be unchecked to produce an acceptable submission, as otherwise the submitted file will be too large.

In the “To archive file:” box, enter standardname2 which must match the Project name. Incorrect naming will produce an unacceptable submission. Once the correct name is entered, select Finish. This will produce a standardname2.zip file which is then placed in the G:Submission folder for submission.