CU Computer Organization
5-stage RISCV32I Processor
Phase 5: Data-path
Fall 2022

Objective: A processor comprises of a data-path which transports instructions, operands, and results while control-signals determine the flow of the data-path and how the information in the data-path is manipulated by processing elements such as the Arithmetic Logic Unit (ALU). In this phase, you will be creating the data-path for a 5-stage RISCV32I processor.

To implement the data-path, you will add the resources defined in your phase 4, Block Diagram, to an imported project. These resources will be added to the ca_resource.codal file located in the "project"/model/ca/resources folder.

> Signals: Are connections within the design that transport data or control-signals from one part of the processor to another such as from the Instruction Decoder to a multiplexer (mux) to select which immediate operand to convey to the Execute stage. Signals contain no state, they do not retain their value between clock cycles.
>
> Registers: Are sequential elements that update their value on a clock edge and retain their state until the next clock edge. Some registers, such as the pipeline registers have additional functionality such as a clear function and a clock enable. If the clear input is asserted (value true) upon the clock edge, the value of the pipeline register will become zero and insert a NOP bubble instead of the register's input value. If the clock enable is deasserted (value false) the value of the register will remain unchanged causing a pipeline to stall.
>
> - Note: In this project, all registers are clocked on the rising (positive) edge of the CPU clock

After creating the resources, the next objective is to use the phase 4 Block Diagrams to add the resources in ca_resources.codal to their individual pipeline description files and make their connections. These files are located in "project"/model/ca/pipelines.

- ca_pipe1_if.codal: Instruction Fetch stage (output pipeline: pipe.IF)
  - The Program Counter (PC) is the input register to the Instruction Fetch stage
- ca_pipe2_id.codal: Decode stage (output pipeline: pipe.ID)
- ca_pipe3_ex.codal: Execute stage (output pipeline: pipe.EX)
- ca_pipe4_me.codal: Memory stage (output pipeline: pipe.ME)
- ca_pipe5_wb: codal: Write Back stage
  - The Register File (RF) is the output register of the Write Back stage

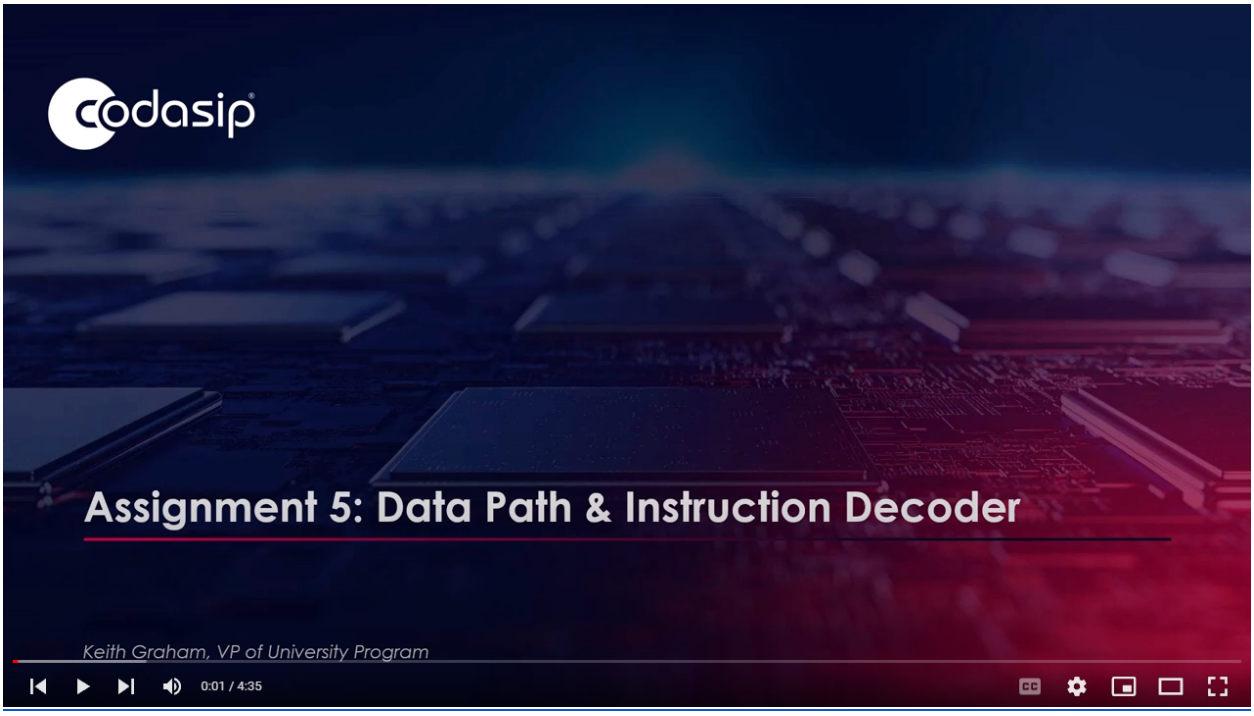
Once the resources are defined and connected, the data-path is ready to be validated against a given test file.

Key Learning Outcomes of this phase:

- Data-Path: The data-path generally includes transporting instructions, operands, and results but does not include control signals or execution units. The data-path is built from signals that connect pipeline registers to logic blocks to execution units to register files to pipeline registers as well as pipeline registers to pass the data-path from one stage to the next or to maintain state. An example of maintaining state is when a control signal specifies to a pipeline to “stall” and not to proceed to the next stage.
- Control-Signals: A stage of the processor is composed of combinatorial logic while the state machine of the processor is determined by the control-signals generated by the Instruction Decoder in the Instruction Decode (ID) stage. There are other control-signals generated such as data hazard detection and forwarding which will be implemented in a future phase. The control signals generated by the ID stage will be forwarded to each stage to specify what operations are to be performed on the data-path in the later stages.
- Block diagrams: One planning tool in computer architecture is the block diagram. It is a visual representation of a processor’s data-path, control-signals, and how both the data-path and control-signal interact with logic blocks and execution units. Good planning reduces both the time to implement and debug.

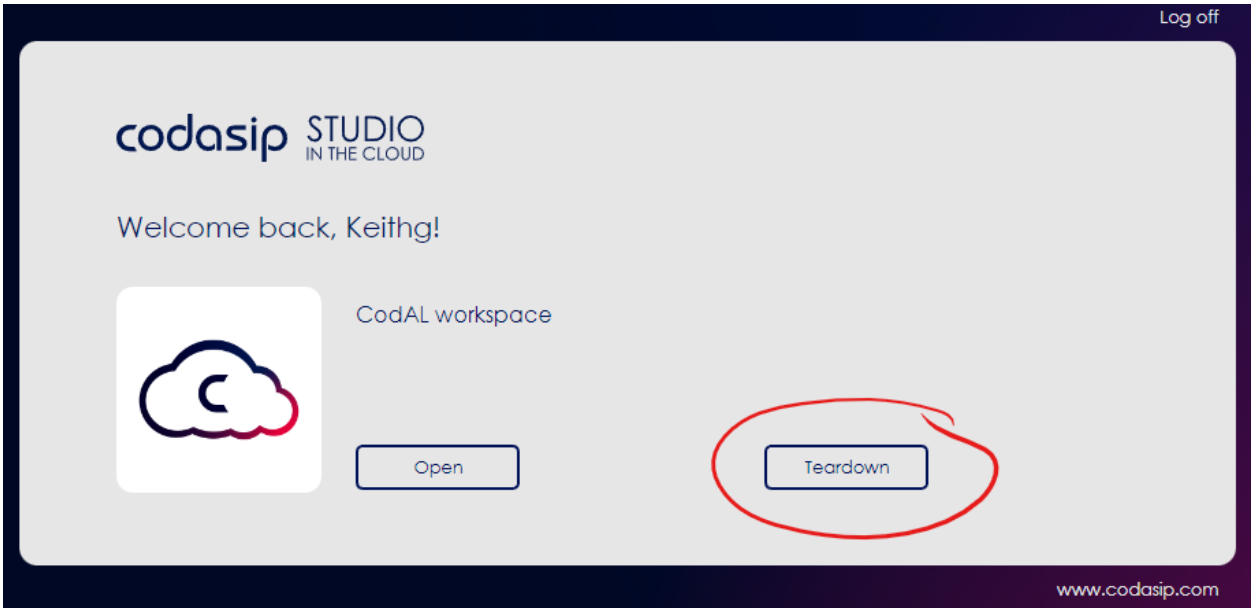
Due Dates:

- This is a 1-week phase
- Due date: Monday October 17th 2022



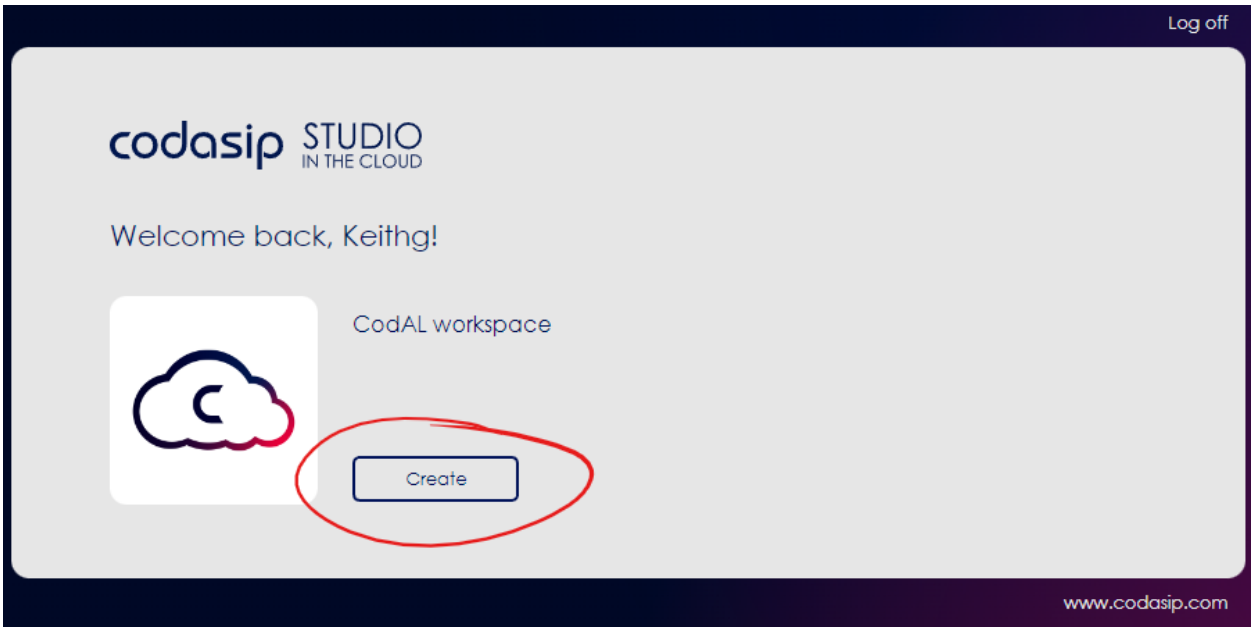
Special Instructions:

- To prevent and remove the 503 error, your workspace needs to be moved onto a new Node group in the cloud. Follow these instructions before you begin your phase:
- Download your regression.s from Phase 3. It is required for this and future phases
- Log into Codasip Studio and choose **Teardown**



- Confirm your **Teardown** request

- Once the Teardown is complete, **Create** a new workspace and click on **Open**



- **You are then ready to proceed to Phase 5 and the 503 error should no longer appear**

Instructions:

- Import a new RISCV processor model, cu_riscv32i_cycle_accurate. You will no longer be using the ia_riscv32i model
  - Type in (or copy and paste) the following command in the Terminal Window to import phase_1_prg
    - git clone https://github.com/CompOrg-RISCV/cu_riscv32i_cycle_accurate.git
- Import the assembly program, addi_assembly_test, that you will be using for this phase.
  - Type in (or copy and paste) the following command in the Terminal Window to import phase_1_prg
    - git clone https://github.com/CompOrg-RISCV/addi_assembly_test.git
- You will also be using your regression test from Phase 3. Either:
  - Import your Phase 3 regression test into your Phase 5 workspace
  - Or, import the below blank regression test and copy your Phase 5 regression test into the imported regression.s file
    - git clone https://github.com/CompOrg-RISCV/regression_test.git

- Review of the Cycle Accurate (CA) files to understand how they interact with each other to create a 5-stage pipeline design. Please refer to Figure 1: Cycle Accurate files under model/ca
  - Please note that all names of files must be lowercase with no spaces within the filename
  - The nomenclature used for this project is all files associated with the Cycle Accurate (CA) model are prefixed with ca_

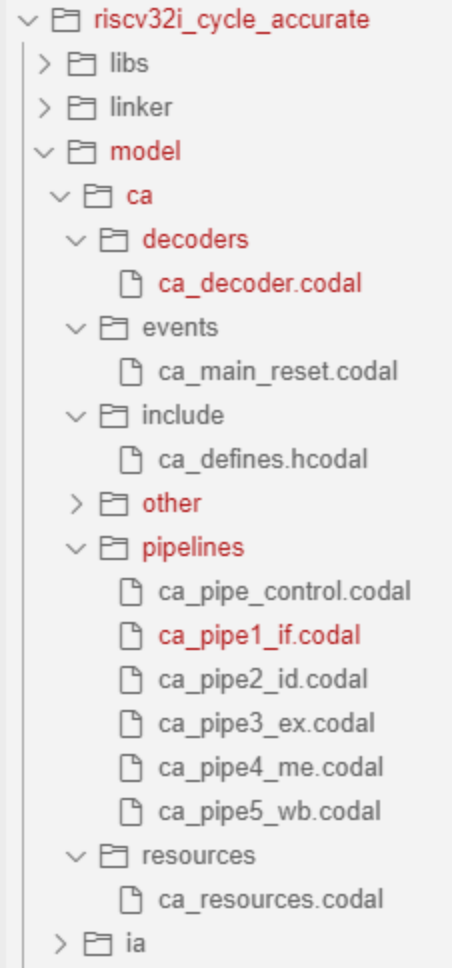


Figure 1: Cycle Accurate files under model/ca

- model/ca/**decoders**
  - ca_decoder.codal is the instruction decoder. If the design is converted into RTL such as Verilog, the code in this file will become combinatorial logic that decodes the instruction opcode to generate the control-signals to implement the instruction throughout the pipelines.
    - You will be updating this file.
- model/ca/**events**
  - ca_main_reset.codal is for simulation purposes only. It informs the simulator the order to execute/simulate the pipeline stages. The simulation starts at the last stage, Write Back, and proceeds to the first stage, Instruction Fetch. This order is critical in that only signals at a later stage can cross pipeline boundaries and only to an earlier stage. A control-signal can never cross pipeline boundaries from an earlier stage to a later stage in that an instruction that has not yet started in the view of a later stage instruction cannot affect it since it has not occurred for this in-order 5-stage pipeline design.

            - For this phase, no changes will be required for this file. This section is for information only.
    - model/ca/**include**
        - ca_defines.hcodal is a header file for the .codal program files. You will add #defines and enums to this file to aid in the programming and development of your CA model
            - You will be updating this file
    - model/ca/pipelines
        - ca_pipe_control.codal handles the logic to specify which pipelines need to be stalled, updated or retain state, due to an earlier instruction in a later stage of the pipeline. These actions require additional cycles to complete its operations. The operations include clearing the pipeline or adding a NOP bubble to a pipeline stage. These are inserted due to an earlier instruction later in the pipeline altering the program flow. Instructions that commonly cause program flow changes are branch or jump instructions.
            - For this phase, no changes will be required for this file. You will be updating this file in the phase to add branch/jump instructions as well as the phase to add load and store instructions.
        - ca_pipe1_if.codal is the Instruction Fetch (IF) pipeline stage. In this stage, the appropriate PC address will be sent to the instruction memory to retrieve its associated instruction.
            - For this phase, no changes will be required for this file. You will be updating this file in the phase to add branch/jump instructions as well as the phase to add load and store instructions.
        - ca_pipe2_id.codal is the Instruction Decode (ID) pipeline stage. In this stage, the instruction will arrive from memory and will be decoded to set all the appropriate control lines for the ID stage and all later stages. This stage will also start the data-path for the source operands for the Arithmetic Logic Unit (ALU) to operate on.
            - You will be updating this file
        - ca_pipe3_ex.codal is the Execute (EX) pipeline stage. This stage will perform the operation requested by the instruction.
            - In later phases, this stage will also compute the address to branch or jump and the address to request a load or store operation from memory.
            - You will be updating this file
        - ca_pipe4_me.codal is the Memory (MEM) pipeline stage. In this stage, the load instruction will be complete. This phase will only be adding the required data-path and control-signals to be passed to the Write Back stage.
            - You will be updating this file
        - ca_pipe5_wb.codal is the Write Back (WB) pipeline stage. The WB is the last stage in the 5-stage pipeline design. Its function is to write the ALU

result or memory load value to the proper register file location on the next rising edge of the processor clock.
        - You will be updating this file
    - model/ca/resources
        - ca_resources.codal is the file that you will create the signals, values that are associated with combinatorial logic, and registers which will update its value upon the next clock edge unless it is specified to "stall" or "clear/flush."
            - You will be updating this file

- ca_pipe1_if.codal
    - The first stage, Instruction Fetch, of a 5-stage pipeline design is responsible for:
        - Determining the address of the instruction to request
            - This address could be the current Program Counter (PC), a branch address if taken, or an unconditional jump address. For this phase, the instruction request will be the PC.
        - Initiates the instruction read from memory by preparing the request to memory which will occur at the next CPU clock. Your design is based on **synchronous** memories where the address of a memory request is sent, and latched into the memory on a CPU clock edge. Contrary, asynchronous memories will provide a read result upon the change to the memory independent of a clock. Many computer architecture books are written assuming asynchronous memories while actual implementations are built around **synchronous** buses and memories
        - Pass the Program Counter of the current instruction request to the Instruction Decode (ID) stage through the IF pipeline register
            - Note: This project uses the pipeline register nomenclature where the first set of letters define the input stage to the pipeline register.
                - For example: pipe.ID register takes its inputs from the Instruction Decode(ID) stage.
        - Update the Program Counter to the next instruction, PC + Instruction Word, 4-bytes for a 32-bit instruction word
    - The components of a .codal pipeline file:
        - Required #include files. Instead of a .h file for a c-program, these files have the .hcodal suffix (lines 27 & 28 in the ca_pipe1_if.codal figure)

```
26
27  #include "config.hcodal"
28  #include "ca_defines.hcodal"
29
30  event fe : pipeline(pipe.IFID)
31  {
32      use if_output;
33
34      semantics
35      {
36          uint2 if_status;
37          uint4 cache_hprot;
38
39          // ------------------------------------------------------------------------------------------
40          // Pipeline control
41          // ------------------------------------------------------------------------------------------
42
43          // If branch, the next PC is not the previous pc + 4, but the calculate branch or jump address
44          if (s_me_take_branch) s_if_nextpc = r_exmem_target_address;
45          else s_if_nextpc = r_pc;
```

ca_pipe1_if.codal

- The events specified by the ca_pipe1_if.codal file, such as event fe (line 30), are the calls to simulate the pipeline in the correct order. Events are analogous to functions. In the event **fe**, another event (function), **if_output** (line 56), will be called (line 52) before the **fe** event completes
- For an event to be called, it must be declared by using the "use" statement (line 32) before the semantics section
- The semantic section (line 34-53) is the code that will be used to simulate your design and to generate the Register Transfer Logic (RTL) such as Verilog for a FPGA. Two important points regarding the semantic section:
  - The pipelines are strictly combinatorial logic, no loops are allowed. An IF statement creates combinatorial logic, so it is allowed in the semantic section but while and for loops are not. The state machine nature of a processor is the flow of the control-signals from one pipeline stage to the next.
  - For simulation purposes, the code is executed from top to bottom. This allows a *signal* to be created in an earlier line of code to be used in a later statement such as setting the select signals for a mux or storing its value in a pipeline register.

- For this phase, you will not be modifying the IF pipeline stage

- ca_decoder.codal
  - The imported project has two instructions enabled in the decoder, a "halt" and "addi," add immediate.
    - The add immediate instruction will be used to validate your data-path since through its immediate field, you can load a value into the register file confirming that the write to the destination register, rd, and its data-path are implemented correctly as well as the arithmetic immediate operand generation and its data-path through the ALU. A second add

immediate can be used to confirm the value was written correctly in the register file as well as the rs1 operand is correctly read from the register file and routed to the src1 operand of the ALU.

- ex: addi x5, x0, 25 (addi rd, rs1, immediate)

- The decoder will generate the control-signals that are required by each stage of the pipeline (ID, EX, MEM, WB) to correctly implement the instruction.

riscv32i_cycle_accurate > model > ca > decoders > ca_decoder.codal

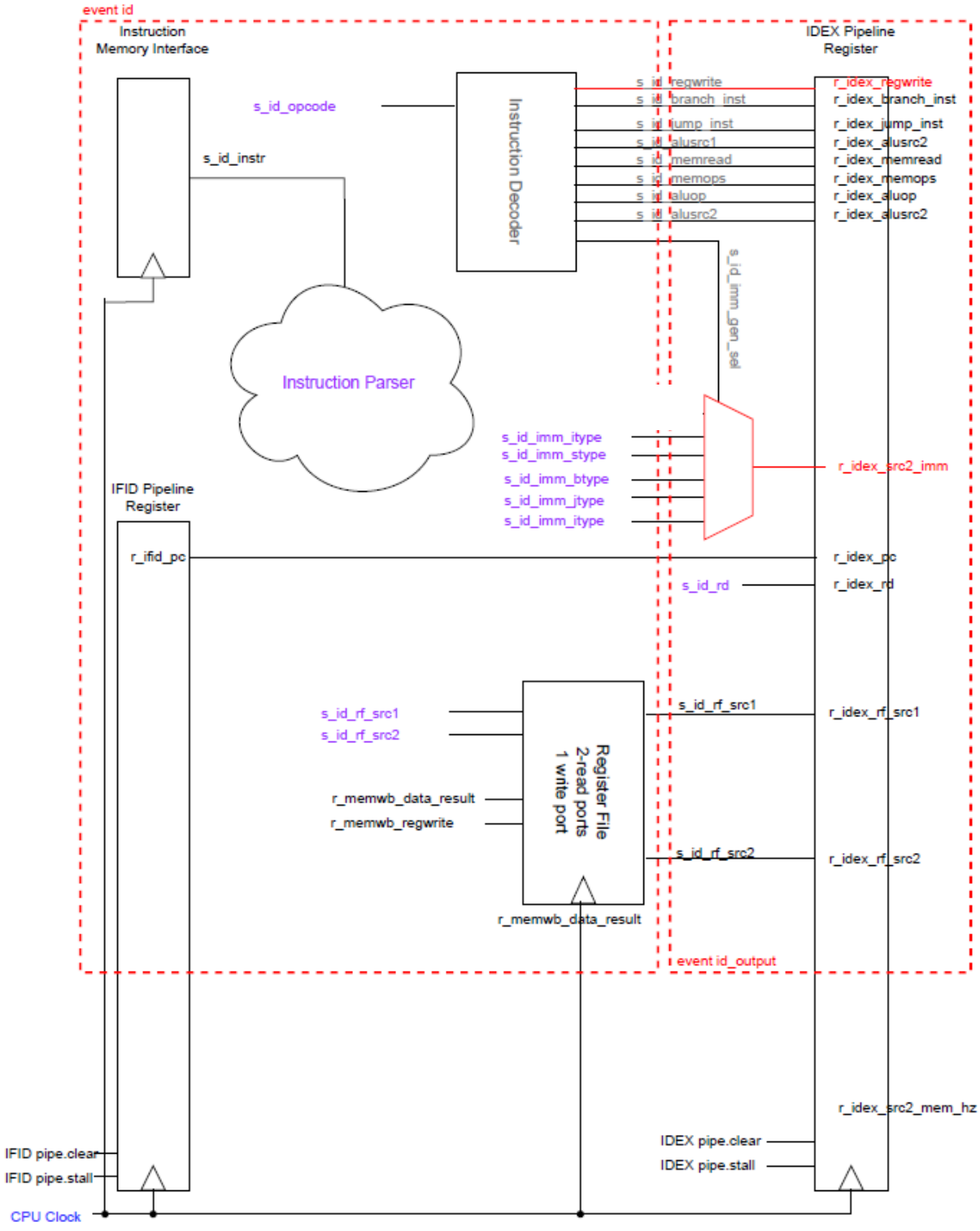
```
element i_hw_itype_alu
{
    use opc_itype_alu as opc;

    assembly { opc };
    binary { opc };

    semantics
    {
        // Register file write enable
        s_id_regwrite = true;

        // This switch statement assigns the alu operation to be performed by this instruction
        switch (opc)
        {
            case ITYPE_ADDI:
                s_id_aluop = ALU_ADD;
                break;
            default:
//                codasip_fatal(ALU_ERROR, "ALU Immediate Decoder case: %d\n", opc);
                break;
        };

        // Operand MUXes
        s_id_alusrc1 = ALU_SRC1_SEL_RF;
        s_id_alusrc2 = ALU_SRC2_SEL_IMM;

        // IMMEDIATE MUX select lines
        s_id_imm_gen_sel = ITYPE_IMM_SEL;

        // Is this a branch instruction to take a branch if branch taken
        s_id_branch_inst = false;
        s_id_jump_inst = false;

        // Load / store control singals
        s_id_mem_ops = MEM_NOP;
        s_id_memread = false;

        // HALT Command
        s_id_halt = false;
    };
};
```

The Decoder's Arithmetic Immediate element

- s_id_regwrite: The result of this instruction will be written to the register file if *true*
- s_id_aluop: The variable to specify to the ALU what operation to perform on its src1 and src2 operands
- s_id_alusrc1 and s_id_alusrc2: Used by the multiplexers in-front of the ALU to select which input source to pass into the ALU to be operated
- s_id_imm_gen_sel: selects which immediate type value to pass into the execute stage based on the type of RISCV instruction
  - itype immediate
  - stype immediate

            - btype immediate
            - utype immediate
            - jtype immediate
        - s_id_branch_inst and s_id_jump_inst: These control lines are used to enable a conditional branch if the result of the ALU operation is *true* or to perform an unconditional jump, s_id_jump_inst.
            - These control lines will not be used for this phase, but they are to be routed to the MEM stage where they will be used in a future phase
            - These control signals are active if *true*
        - s_id_mem_ops: Encoded variable that will inform the load and store units to perform their operations if a load or store instruction is being executed
            - This control variable will not be used for this phase, but it should be routed to the EX and MEM stages for a later phase
            - Setting the variable to 0 results in a memory NOP operation
        - s_id_memread: Specifies to the pipeline that the value to write to the register file will be the result of a load memory operation and not the result from the ALU if *true*
            - This control line will not be used for this phase, but it should be routed to the EX and MEM stages for a later phase
    - At this stage of the phase, you will not be modifying the ca_decoder.codal file. At a later stage of the phase, you will add the remaining arithmetic immediate operations and the arithmetic register-to-register operations.
    - Note: For all the above control signals, they are active if *true*, 1, and disabled if *false*, 0. This definition of control signals is critical to implement a NOP bubble. If all control signals become 0, the functions that are controlled by these control signals will not occur, effectively performing a NOP operation. The clear pipeline register function will set the specified pipeline register values to 0, disabling the control signals, turning the current instruction into a NOP.

- ca_pipe2_id.codal



- The second stage, Instruction Decode (ID), of a 5-stage pipeline design is responsible for:
  - Completing the instruction read from memory

- Parsing the effective read instruction
- Generating the immediate operand, extended to 32-bits if required
- Reading the rs1 and rs2 values from the Register File
- Decoding the instruction
- Setting the values in the ID pipeline register to pass along to the Execute (EX) stage

- Completing the instruction read from memory requested from the previous clock cycle's Instruction Fetch (IF) stage

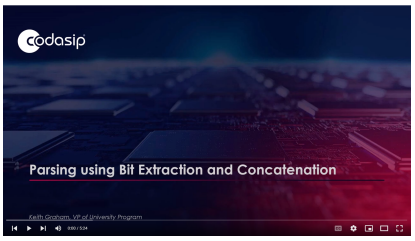
```
48      if_fetch.transport(CP_PHS_DATA, cache_ready, cache_response, id_instr_temp); // Read the instruction from memory request in if-stage
49
50      if (!cache_ready) s_id_icache_stall = true;                    // If i-cache not ready, stall instructions until i-cache is ready
51      else s_id_icache_stall = false;
52
53      instr_control = (uint1)(r_id_buffer_full) :: (uint1)(s_id_icache_stall || !id_instr_temp[0..0]);
54
55      switch (instr_control) {
56          case (uint2) 10:               // instruction buffer full, recovering from ID stage stall
57          case (uint2) 11:               // buffer has priority over instruction from memory or cache
58              s_id_instr = r_id_instr_preserved;
59              break;
60          case (uint2) 01:               // cache is stalled or instruction input is 0
61              s_id_instr = NOP_INSTRUCTION;
62              break;
63          case (uint2) 00:               // instruction from memory or i-cache
64              s_id_instr = id_instr_temp;
65              break;
66      }
```

Instruction read request and setting the s_id_instr instruction signal to be parsed

- No modification of the setting the s_id_instr signal required

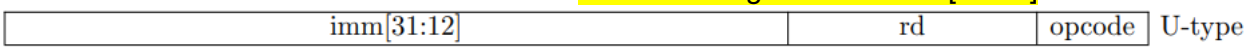
- Parse the effective instruction

Parsing using Bit Extraction and Concatenation: An instruction fetched by a processor must be parsed to perform the individual operations defined by the instruction format bit fields. For example, in RISCV, if an instruction requires the source 1 operator, it will always extract from the RISCV instruction or machine code bits 19 thru 15 to index the Register File to read the source 1 value. Other signals required by the instruction decode pipeline will require multiple bit fields to be concatenated together. This video will show you how to extract and concatenate bit fields from 32-bit instructions into the individual signals that are required by your 5-stage processor pipeline.



- With the effective instruction, s_id_instr, provided to be parsed, open ca_pipe2_id.codal to edit the Instruction Decode (ID) pipeline file
- To parse a 32-bit instruction, you will use the following:
  - To select specific bits of interest, type the bit number within the brackets, [], following s_id_instr, add the largest bit location of interest followed by ".." and then the least bit location desired. You will use the same format for a single bit such as s_id_inst[7..7]
    - ex: func7 = s_id_instr[31..25], func3 = s_id_instr[14..12], opcode = s_id_instr[6..0]

        - ex: single bit notation s_id_inst[11..11]
    - To concatenate bits into a single value, you will use the :: concatenate operator
        - ex: the complete RISCV opcode will be represented by s_id_opcode = s_id_instr[31..25] :: s_id_instr[14..12] :: s_id_instr[6..0];
- Now, complete the parsing for all the following signals provided in the file:
    - Use sections 2.2, 2.3, and 2.4 of the RISCV Instruction Manual to determine what bits are required for each of these signals
        - https://riscv.org/wp-content/uploads/2019/06/riscv-spec.pdf
    - s_id_src1: rs1 register
    - s_id_src2: rs2 register
    - s_id_rd: destination register
    - Generating the Immediate Operand: At this point, the instruction has not been fully decoded to determine if an immediate value will be required, but to optimize performance, all immediate values will always be calculated simultaneously and once the instruction is decoded, only the required immediate value will be passed to the EX stage via the ID pipeline register using a multiplexer
        - s_id_imm_itype: itype instruction immediate value
        - s_id_imm_stype: stype instruction immediate value
        - s_id_imm_btype: provided as an example.
            - Per the example, you will need to correctly sign extend the parsed value to make it a 32-bit operand
            - Some immediates are shifted over one-bit such as the btype immediate. You can specify an immediate shift over by one bit through concatenating, ::, a single 0 bit at the end of the parsed string (int1)0. Now, the immediate field in the signal starts at bit 1 instead of bit 0.
        - s_id_imm_utype: utype instruction immediate value
            - Notice, the utype immediate is shifted over by 12 since the range is shown as [31:12]



        - s_id_imm_jtype: jtype instruction immediate value
- Read the required register values for rs1 and rs2 from the register file
    - RISC-V design goal: “In RISC-V the specifiers of the registers to be read (rs1 & rs2) and written (rd) are always in the same location of all instruction types (i-type, r-type, b-type, s-type, j-type, and u-type), which means the registers can begin before the instruction is decoded.” (From the RISC-V Reader: An Open Architecture Atlas” by David Patterson and Andrew Waterman)
    - Similar to the immediate decode, both src1 and src2 values will be read in parallel from the register file as the instruction is decoded to optimize

performance. The register file has been defined as 1 write port, wrt0, and two read ports, src1 and src2. To specify the the port, you indicate it by affixing the port at the end of the register file name with a "." and the "port name"

- Here is the architectural definition of the register file found in "project"/model/share/resources/arch.codal which declares the Register File name as rf_xpr

```
arch register_file bit[WORD_W] rf_xpr
{
    dataport src1, src2 {flag = R;};
    dataport wrt0 {flag = W;};
    size = RF_XPR_SIZE;
    reset = true;
    default = 0;
};
```

- To specify the desired register, you use the register number as an index into an array as you would in c-programming.
  - Example is register_file.port[register_number]
  - ex: rf_xpr.src1[index into Register File], rf_xpr.src2[index into Register File], rf_xpr.wrt0[index into Register File]
  - Ex: rf_xpr.src1[15]; // register 15 value of the Register File src1 port
- Now, assign the value read from the Register File read ports to the src1 and src2 signals pointed by s_id_src1, rs1 register index, and s_id_src1, rs2 register index.
  - s_id_rf_src1: value of rs1 register; s_id_src1 register index
  - s_id_rf_src2: value of rs2 register; s_id_src2 register index

- Decode the Instruction: The combinatorial logic for the ID event is almost complete. The next step is to call the instruction decoder which has been included in the imported project. You will not be modifying the instruction decoder at this time.
  - inst_decode(s_id_opcode);
- Set the values in the ID pipeline register to pass them along to the EX stage
  - After the instruction has been decoded, it is time to store the appropriate values in the ID pipeline register by calling the Instruction Decode Output event within the fe event
    - id_output();
  - event id_output : pipeline(pipe.ID)
    - Signal and Register nomenclature. In this course, all global signals will begin with s_ and all registers will begin with r_. To easily identify what pipeline stage a signal/register belongs to, the next part of the name will either be the pipeline stage such as ID

or the pipeline register, ID signifying the pipeline register between the ID and EX stages. The next term after the signal/register location is the actual name such as pc for program counter.
    - ex: r_id_pc: ID pipeline register for the Program Counter (PC)
- The format to declare a signal is the following:
    - **signal bit**[number of bits wide] signal_name;
    - bit[] defines the width, number of bits that
    - From the example, you will notice that there are predefined widths for most of the signals such as INSTR_W, OPC_W, RF_XPR_W, etc. These definitions can be found in project/model/share/include/config.hcodal or project/model/ca/include/ca_defines.hcodal or ca_resources.codal
    - You will be using the signal name nomenclature of s_ for signal concatenated with pipeline_ concatenated with the name:
        - ex: s_id_instr

```
// Instruction and parsed instruction fields
signal bit[INSTR_W]      s_id_instr;     // Instruction
signal bit[OPC_W]        s_id_opcode;    // Complete 17-bit opcode
signal bit[RF_XPR_W]     s_id_src1;      // src1 register address
signal bit[RF_XPR_W]     s_id_src2;      // src2 register address
signal bit[RF_XPR_W]     s_id_rd;        // rd destination address
signal bit[WORD_W]       s_id_rf_src1;   // value of src1 from register file
signal bit[WORD_W]       s_id_rf_src2;   // value of src2 from register file
```

ca_resource.codal signal declaration example

- The format to declare a register is the following:
    - **register bit**[number of bits wide] register_name;
    - bit[] defines the width, number of bits that
    - From the example, you will notice that there are predefined widths for most of the registers such as ADDR, WORD_W, BOOLEAN_BIT, etc. These definitions can be found in project/model/share/include/config.hcodal or project/model/ca/include/ca_defines.hcodal or ca_resources.codal
    - You will be using the register name nomenclature of r_ for signal concatenated with pipeline_ concatenated with the name:
        - ex: r_id_icache_stall

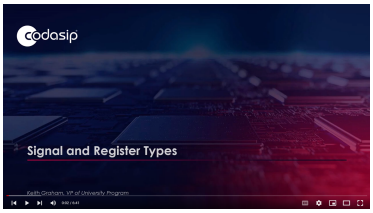
- NOTE: The curriculum videos may use an alternative naming convention for the pipeline stage where he name is concatenate by the stage name of the input to the pipeline register with the stage of registers output

- Example is r_idex_pc
  - Input to the pipeline register: ID stage
  - Output of the pipeline register: EX stage
- Alternative name is r_id_pc where the pipeline register name is associated with its input

```
// Singals/registers to preserve instruction durin a pipeline stall
register bit[BOOLEAN_BIT]   r_id_icache_stall           {default = false;};   // 1-cycle delay of s_id_icache_stall for timing
register bit[BOOLEAN_BIT]   r_id_flush_icache           {default = false;};   // used to flush current instruction being read
register bit[BOOLEAN_BIT]   r_id_buffer_full            {default = false;};   // indicate whether previous cycle stalled
register bit[INSTR_W]       r_id_instr_preserved;      // Preserve last instruction to reconstruct inst. in stall
register bit[WORD_W]        r_id_pc_preserved;         // Preserve the PC address of instruction safed in buffer
```

ca_resource.codal register declaration example

Signal and Register Types: Similar to other programming languages, processor elements must be declared. CodAL, a processor description language, has several new types of variables compared to c or c++. Signal types are the wires connecting one part of the circuit to the other while registers are sequential storage elements that retain state between clock edges. The pipeline register is a special type of register which includes two additional inputs, Clock EN(able) and Clear. This video shows you how to declare these different types of processor elements.



- Now, go to ca_resources.codal and **use your block diagram** to add all the data-path ID pipeline registers specified there.. At this time, do not add any data-path for data forwarding. Data forwarding will be a future phase. Look for the comments in the Instruction Decode (ID) portion of ca_resources.codal that do not have a declared variable.

```
// ID pipeline register
                                { pipeline = pipe.ID; }; // Program Counter
                                { pipeline = pipe.ID; }; // rd (write) register number
                                                         // register file output for src1, rs1
                                                         // register file output for src2, rs2
                                                         // RF write? r_id_regwrite
                                                         // immediate value for src2
```

- Common definitions for the number of signal lines or register bits
  - ADDR_W = number of address bits wide (W)
  - WORD_W = number of data bits wide (W) for CPU
  - BOOLEAN_BIT = 1 bit commonly used for control signals
  - RF_XPR_W = number of bits wide (W) to address (index) the Register File
- Which of the above definitions will be used for each of the registers to be declared in the above screenshot?
- The first two variables defined are declared as pipeline variables with the addition of { pipeline = pipe.ID; };
  - For the last four variables, you must also make them ID pipeline registers by using the { pipeline = pipe.ID; };

- The reason we need to have the addition of { pipeline = pipe.STAGE; }; is for the ca_pipe_control. This addition is necessary to identify which registers belong to a given pipeline. In later phases we will implement pipeline stalls and pipeline clears. Without the addition, it would not be able to identify the pipeline this register belongs to.
- Make the appropriate register declaration for each of these ID-stage pipeline registers

```
// ID pipeline register
                                        { pipeline = pipe.ID; }; // Program Counter
                                        { pipeline = pipe.ID; }; // rd (write) register number
                                                                 // register file output for src1, rs1
                                                                 // register file output for src2, rs2
                                                                 // RF write? r_id_regwrite
                                                                 // immediate value for src2
```

- With these pipeline registers defined, go back to ca_pipe2_id.codal's id_output event to store the correct values into these pipeline registers. There are comments to help guide you through the implementation.
    - The Immediate Value ID pipeline register, that you will define, has multiple input options. Each of the RISCV instruction types has an immediate type. A register can only have a single input, so a multiplexer will be implemented in front of the pipeline register to select the desired immediate operand to send to the EX stage. A c-programming switch statement will be used. This multiplexer select line is from the instruction decoder, s_id_imm_gen_sel.
        - It is best to implement it as a switch statement whose case "values" are defined by the enum in ca_defines.hcodal

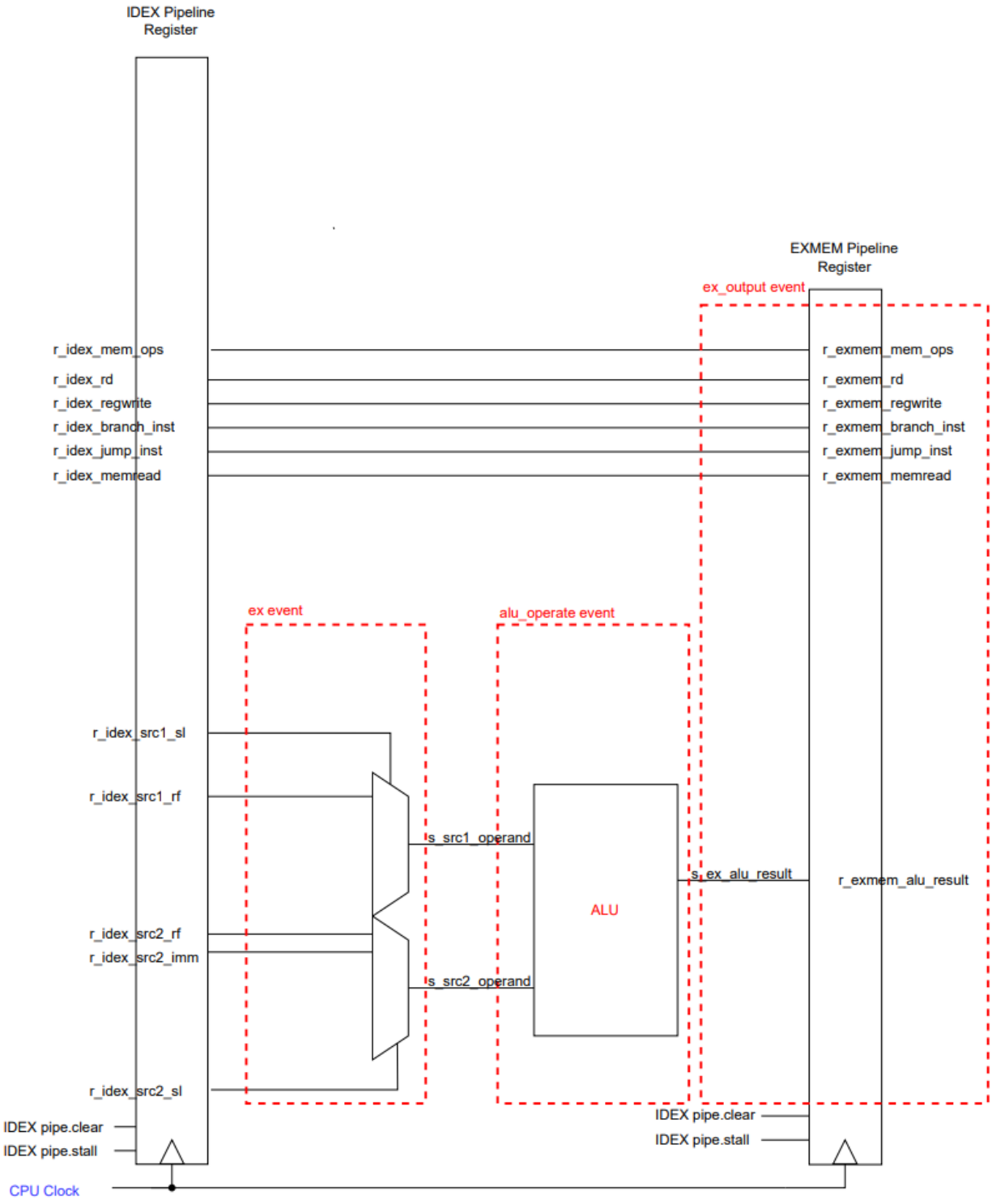
```
// ---------------------------------------------------------------------------------------------
// ID stage
// ---------------------------------------------------------------------------------------------
enum imm_gen_mux   // Select lines for generating the proper immediate to pass to the execute stage
{
    ITYPE_IMM_SEL,
    STYPE_IMM_SEL,
    BTYPE_IMM_SEL,
    UTYPE_IMM_SEL,
    JTYPE_IMM_SEL
};
```

    - The regwrite, the control-signal that specifies to write the result into the register file if true, is also not directly assigned.
        - In the RISCV architecture, register 0 is a special register that will always read 0 which implies that it

            - is a read-only register of value 0. If the destination register is 0, disable the writes to register 0 by setting the regwrite signal to *false*, else, regwrite equals the signal from the instruction decoder, s_id_regwrite
            - r_id_regwrite is assigned based on the condition whether the destination register, rd, decoded in this stage is x0 to be *false* or else assigned the value from the Instruction Decoder
        - After assigning the value for the regwrite to be passed into the ID pipeline register, assign the remaining registers whose signals are generated by ca_decoder.codal.
    - The ID stage data-path for this phase is now completed

- ca_pipe3_ex.codal

Execute (EX) Stage Block Diagram



- The third stage, Execute (EX), of a 5-stage pipeline design is responsible for:
  - Selecting the ALU input operands
  - Performing the ALU operation
  - Assign the EX pipeline register to pass along to the MEM stage

        - In later phases, ca_pipe3_ex.codal will be responsible for branch/jump address calculation and to initiate a load/store operation
    - Selecting the ALU input operands
        - Both ALU source operands, src1 and src2, have more than one source for the operand. As you build out your processor through the phases, src1 could be sourced by the register file rs1 input, saved program counter, a constant, or data forwarding from a later pipeline stage. Similarly, the src2 operand can be sourced by more than one source.
        - As a minimum three signals in the EX stage will be required:
            - Output of src1 operand mux
            - Output of src2 operand mux
            - ALU result
            - Go to ca_resources.codal, and add the above **signals** with their proper data widths in the EX pipeline section. Operands and results for a 32-bit CPU are 32-bits wide. What is the proper definition to use for the number of bits per signal? Use the proper definition from project/model/share/include/config.codal. Using definitions instead of specific values is best coding practice. By using a definition, the CPU can easily go from 32-bits to 64-bits by changing a few definitions. The grading rubric will include using best practices.
            - Also, add an **EX pipeline registe**r to store the result from the ALU. You can find a place holder in the first line after the comment // EX pipeline register

```
// ------------------------------------------------------------------------------------------
// Execute stage
// ------------------------------------------------------------------------------------------

// ALU signals
                                                                    // src1 operand into ALU
                                                                    // src2 operand into ALU
                                                                    // result of ALU operation

// EX pipeline register
                                                                    // Data result from ALU
```

        - In the ca_pipe3_ex.codal ex event, create two muxes to create both the src1 and src2 ALU operands using two of the signals that you have just declared as their output signal. There are comments to help you implement these mux in the ex event.

```
// Develop a multiplexer for the ALU src1 operand using a switch statement using the src1_sl
// control signal provided by the ID pipeline register generated by the Instruction decoder
// Hint:  Using the standard naming protocols for this project, what prefix would you add to
// to src1_sl to build the complete name of input value for this switch statement?
//
// In Phase 5, there is only one input which is the src1 register file, rf, value and
// this input is from the ID pipeline register
```

```
// Develop a multiplexer for the ALU src1 operand using a switch statement using the src1_sl
// control signal provided by the ID pipeline register generated by the Instruction decoder
// Hint:  Using the standard naming protocols for this project, what prefix would you add to
// to src1_sl to build the complete name of input value for this switch statement?
//
// In Phase 5, there are two inputs to the source 2 operand mux, the src2 register, rf, value
// and the immediate value.  Both of these inputs are from the ID pipeline register
```

- The use of a **switch** statement is best practices to generate a mux, multiplexer
  - ALU operand 1 mux switch variable is the r_id_src1_sl while the ALU operand 2 mux switch variable is the r_id_src2_sl
- The case statement **enum** definitions for these muxes can be found in project/model/ca/includes/ca_defines.hcodal. You will need to use the proper enum definitions for each of the ALU source muxes

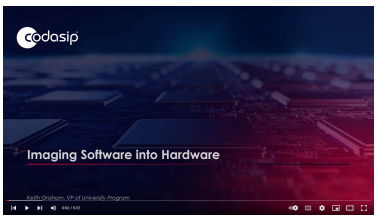
- Performing the ALU operation
  - To verify the data-path, at this point of the phase, you will stick with a single ALU operation, the ALU_ADD. You can view the ALU enums in the ca_defines.hcodal file.
  - The aluop enum defines the micro-architecture ALU opcodes

```
// ------------------------------------------------------------------------------------------
// EX stage
// ------------------------------------------------------------------------------------------

// ALU function codes
enum aluop
{
    ALU_NOP = 0,// In enums, you can specify a particular value to an element.  All other
    ALU_ADD,    // elements will have a value incremented by 1 unless assigned a specific value
};
```

- Similar to the muxes created for the ALU source operands, a switch statement will be used to implement the ALU operations. Within the ca_pipe3_ex.codal ex event, there is an alu_operate event. Inside this event's semantics statement, add a switch statement to evaluate all the possible alu operations enabled at this time based on the following:

Imaging Software into Hardware: The end goal of designing a processor is to implement it in hardware. The goal of the hardware design may be smaller area, lower power, or higher frequency of operation. To achieve these goals, it is important to understand how a High Level Synthesis (HLS) software construct relates to its hardware implementation. This video will describe the hardware construct of a software switch statement and how it can be used to

reduce the time to calculate a potential signal, effectively increasing the system frequency of operation.



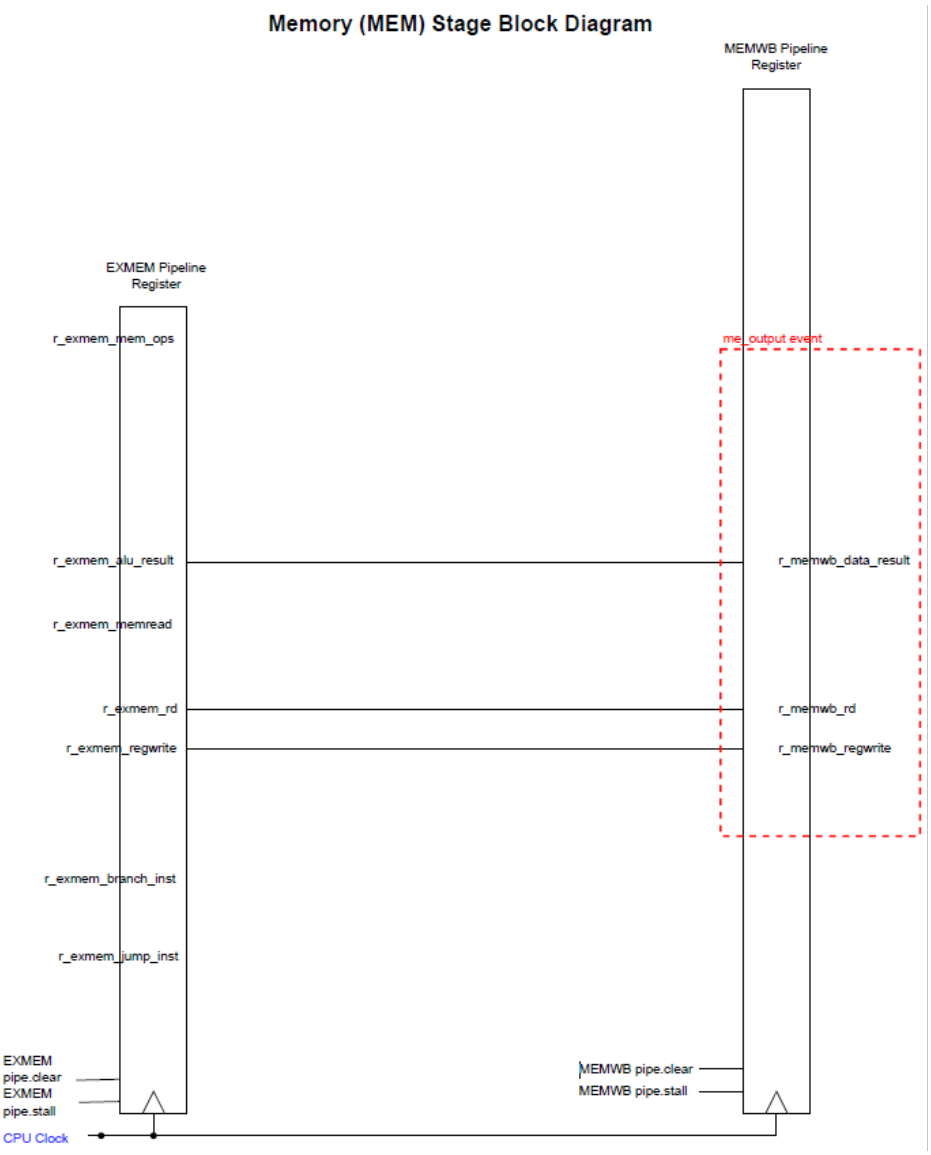
- aluop is the signal that controls the switch statement. It will come from the ID pipeline register.
- For each **element in the enum aluop**, there should be a case statement
  - The case statement should assign the alu operation result to the signal that you defined in ca_resources.codal for the ALU result. The operands are the signals that are derived from the operand muxes that you have just added above.

    Ex: case ALU_ADD:

    s_ex_alu_result = s_src1_operand + s_src2_operand;

    break;
  - It is best practice for each switch statement to have a default case. Add a default case. The grading rubric will include using best programming practices.
- Assign the EX pipeline registers to pass the required control signals and data-path to the WB stage
  - In the ca_pipe3_ex.codal ex_output event, assign the ALU result signal from the ALU mux to the proper EX pipeline register that you created to pass the ALU result to the MEM stage
  - In the ca_pipe3_ex.codal ex_output event, assign the proper ID pipeline registers to their EX equivalent to pass them into the MEM stage
    - The Instruction decoder sets the control signals for each of the pipeline stages and must be passed along from one pipeline register to the next until it is required or used
- The EX stage data-path for this phase is now completed

- ca_pipe4_me.codal



- The fourth stage, Memory, of a 5-stage pipeline design is responsible for:
  - The completion of the load and store memory operations. Load and Store operations will be added to the project in a later phase
  - Assign the MEM pipeline registers to pass along to the WB stage the required control-signals and data-path
    - The three signals that were passed into the MEM stage will need to be passed into the Write Back, WB, stage through the MEM pipeline register
- Assign the MEM pipeline registers the EXEMEM control-signals and data-path to the WB stage
  - Go into ca_resources.codal and in the MEM stage section, create the MEM pipeline registers to pass them into the WB stage after the comment // MEM pipeline register

- Hint: There are three MEM pipeline registers that need to be declared. There are comments for each of these pipeline registers.
- Hint: Ensure that you define them as pipeline registers by using { pipeline = pipe.MEM; }

```
// ------------------------------------------------------------------------------------------
// Memory stage
// ------------------------------------------------------------------------------------------
signal bit[BOOLEAN_BIT] s_me_take_branch;        //Informs the pipeline in a branch or jump is occuring
signal bit[BOOLEAN_BIT]   s_me_mem_busy;         // Memory bus is busy


// MEM pipeline register
                                                 // ALU result from Memory stage
                                                 // Destination, rd, register index
                                                 // Register Write (regwrite)
```

        - As you did in the ex_ouput event, go to ca_pipe4_me.codal's me_output event to assign the appropriate EX pipeline register as the input to its equivalent MEM pipeline register
    - The MEM stage data-path for this phase is now completed

- ca_pipe5_wb.codal

    - The fifth stage, Write Back, of a 5-stage pipeline design is responsible for:
        - Updating the Register File with the result of the ALU or load operation if the instruction in the WB stage is to write to the Register File
    - The Write Back stage is the last stage of this 5-stage pipeline design which infers that there is no pipeline register to update, no stage to pass additional data-paths or control-signals to.
    - Update the Register File with the result of the ALU or load operation if the instruction in the WB stage is to write to the Register File
        - There are three MEM pipeline registers that pass information into the WB stage. Use all three registers values to write to the Register File
            - rd: The register to store the ALU result
            - regwrite: Control-signal which specifies whether the instruction in the WB stage is to write to the Register File if true
            - ALU result

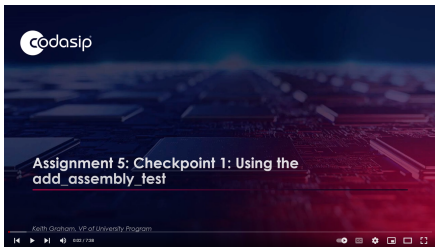
```
// if the register write (regwrite) control bit is true, update the register file with the mem
// alu result and if false, do nothing
//
// There are three ports to the register file, two read (source) inputs and one write port in this
// 5-stage CPU design
//
 // In phase 5, you will use all the three pipeline registers that were passed into the WB stage,
// regwrite, rd, and alu_result
```

- The write to the Register File should be performed with an if statement to write the appropriate register location, rd, in the Register File with the ALU result only if the instruction decoder has asserted the control signal regwrite is true
    - For your reference: The register file has been defined as 1 write port, wrt0, and two read ports, src1 and src2. To specify the the port, you indicate it by affixing the port at the end of the register file name with a "." and the "port name"
        - Example: reading from register indexed location 5 from the src1 port would be represented as **rf_xpr.src1[5]**
    - Here is the architectural definition of the register file found in "project"/model/share/resources/arch.codal

```
arch register_file bit[WORD_W] rf_xpr
 {
      dataport src1, src2 {flag = R;};
      dataport wrt0 {flag = W;};
      size = RF_XPR_SIZE;
      reset = true;
      default = 0;
};
```

- The WB stage data-path for this phase is now completed

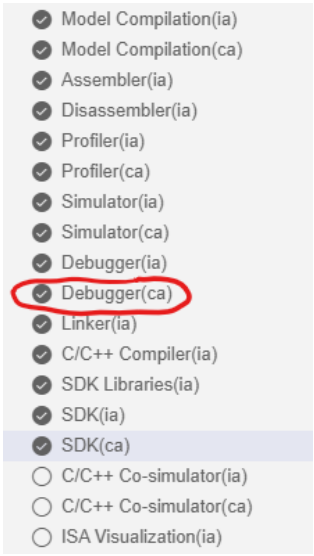
Checkpoint 1: Validate a single i-type instruction through the CA design

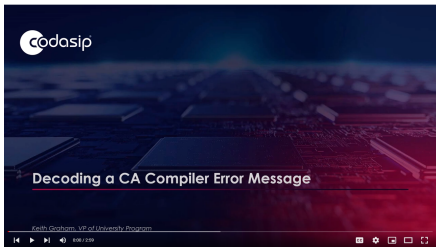


Phase 5: Checkpoint 1: Using the addi_assembly_test: In Phase 5, Checkpoint 1, you will be using the addi_assembly_test provided to validate your ITYPE data path. This video will show you how to launch the addi_assembly_test using your Cycle Accurate (CA) model and how the Architectural & Microarchitectural registers can be used to validate & debug your ITYPE data path.

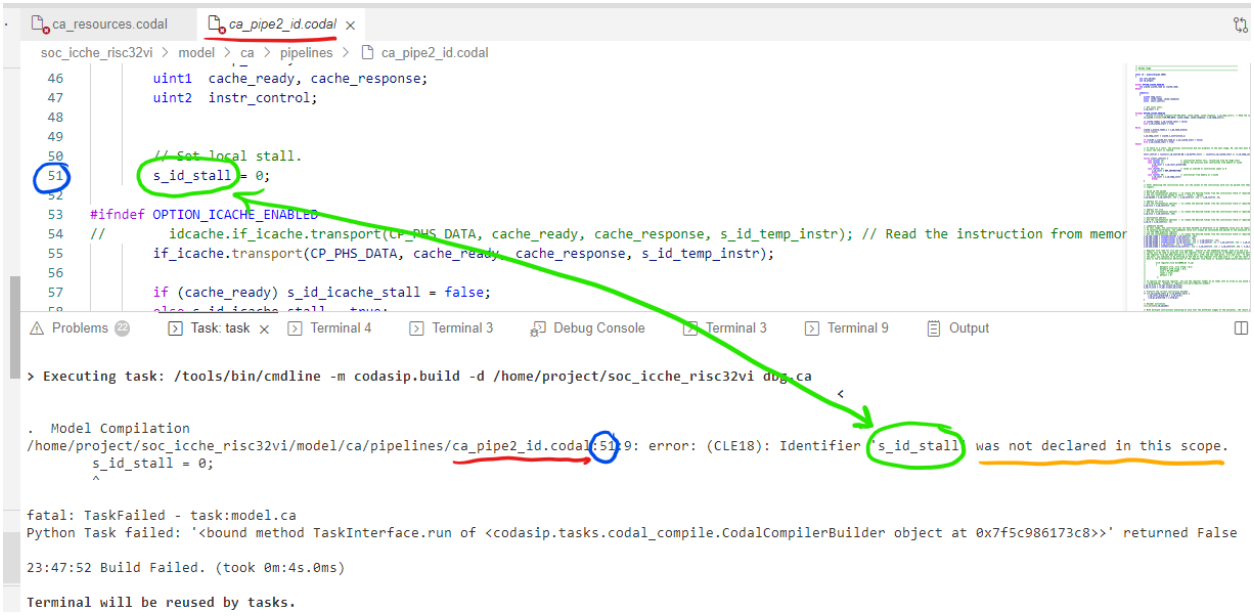
- The imported addi_assembly_test project provided with this phase into your workspace. It is an assembly program that is designed to validate the add immediate, addi, instruction. There are comments which will provide details of the instruction and comments noting when the destination register in the Register File has been updated and its new value.
- You will notice that there is no Assembler(ca) for the Cycle Accurate (CA) tasks. When compiling (building) an assembly routine, the IA model is used.

- To build all the required tasks to simulate an assembly program, double click the **Debugger(ca)** task or **SDK(ca)** which will build Model Compilation(ca), Simulator(ca), and Debugger(ca)





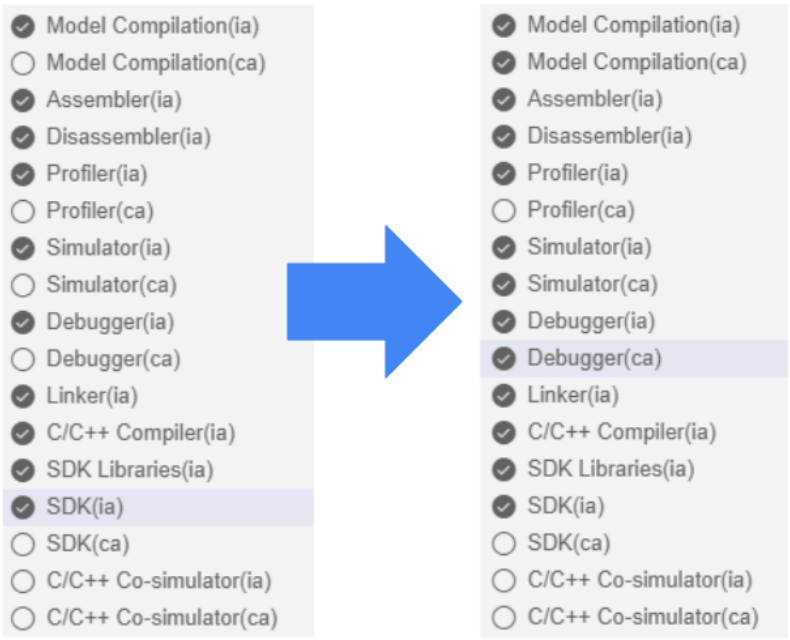
Decoding a CA Compiler Error Message: Getting as quickly as possible to an error in a project will speed up your debug and project development time, especially if you have a project which comprises many files. This video will teach you how to decode or dissect your Error Message while compiling your Cycle Accurate (CA) model enabling you to locate the error with helpful information on the error type.



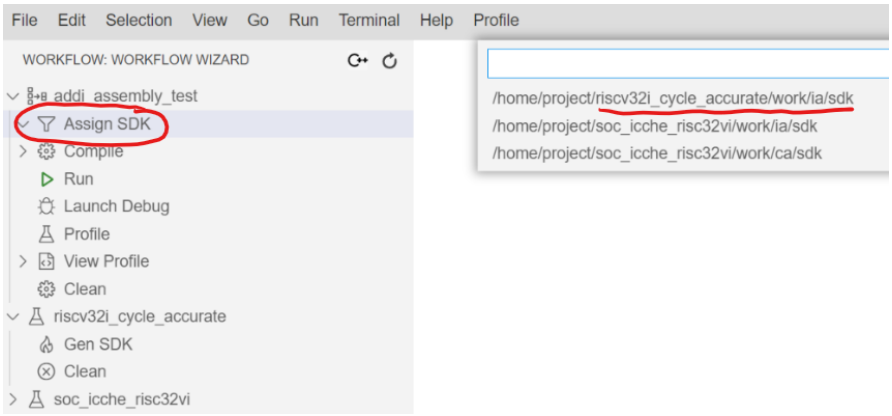
- If your project fails the build, generally you will receive detailed information on the failure. You can click on the file name to open the file. The number immediately following the filename is the line number in the file where the error has occurred. Additional information such as the identifier will help locate where in the line of code the error occurred or the error itself.
  - Please refer to the course debug videos to help with specific error messages of program compilation errors
  - It is OK if you get these Unused signals, registers, and resources messages while building your CA model. They may be used in a future phase.

```
> Executing task: /tools/bin/cmdline -m codasip.build -d /home/project/riscv32i_phase_5 sdk.ca

.  Model Compilation
/home/project/riscv32i_phase_5/model/ca/resources/ca_resources.codal:81:1: warning: (CLW11): Unused register
'r_id_flush_icache'.
register bit[BOOLEAN_BIT]   r_id_flush_icache           {default = false;};   // used to flush current instruction being read
^
/home/project/riscv32i_phase_5/model/ca/resources/ca_resources.codal:169:1: warning: (CLW11): Unused signal
's_syscall_pending'.
signal bit [BOOLEAN_BIT]   s_syscall_pending;
^
/home/project/riscv32i_phase_5/model/ca/resources/ca_resources.codal:170:1: warning: (CLW11): Unused signal 's_syscall_stall'.
signal bit [BOOLEAN_BIT]   s_syscall_stall;
^
/home/project/riscv32i_phase_5/model/ca/resources/ca_resources.codal:171:1: warning: (CLW11): Unused register
'r_syscall_delay'.
register bit [TWO_BIT]  r_syscall_delay        {default = 0;};
^
/home/project/riscv32i_phase_5/model/share/resources/interface.codal:71:1: warning: (CLW11): Unused interface 'ldst'.
interface ldst          // if refers to Interface element and ldst refers to load/store (data) space
^
```

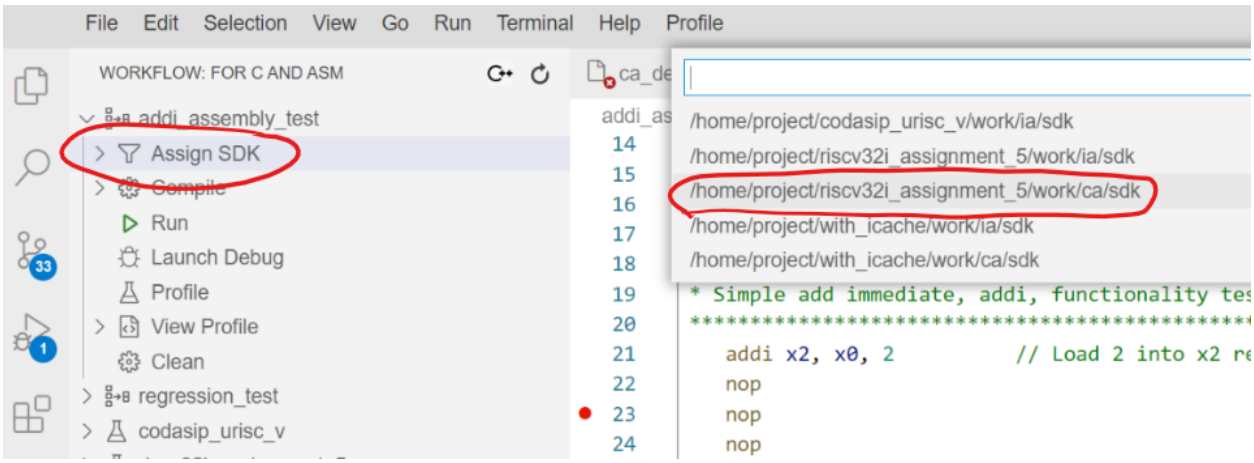
- Once your project has successfully been built, all the CA tasks required to run assembly programs should be built and identified with a checkmark: Model Compilation(ca), Simulator(ca), and Debugger(ca).



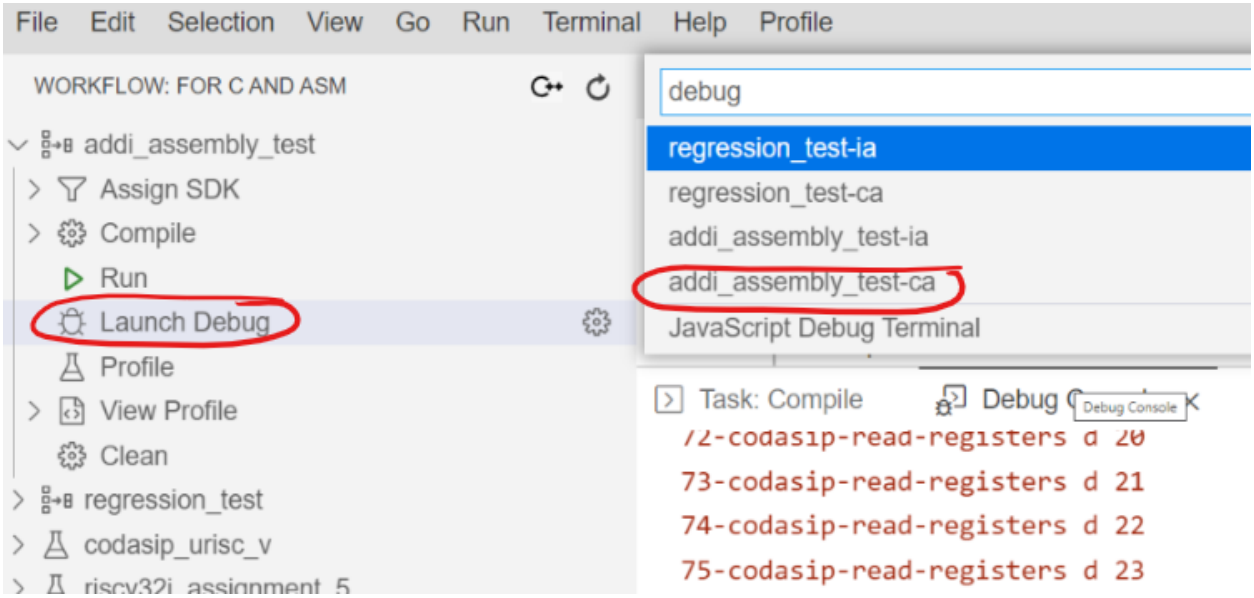
- The (ia) model contains the assembler and c-compiler, so for all assembly and c-program tasks, you will associate the (ia) model with the project.
- If your (ia) model is not built, click on the SDK(ia) task to build the project
- Assign the SDK of the addi_assembly_test project to point to your .ia project



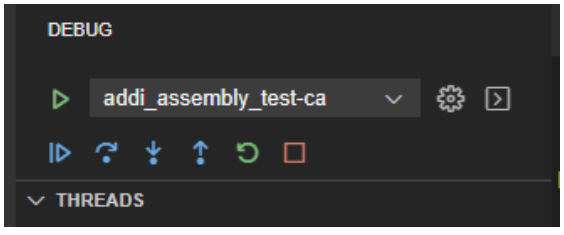
- Build the addi_assembly_test project
- With the IA, CA models and test program built (compiled), you will now run the assembly program to verify your data path and instruction decode. Configure a “Debug Configuration” to run the addi_assembly_test project.
- Next, click on the Debugger Tab and change the select to “Stop at startup” to **“after n instruction(s)” of 0**
- After a successful compile of your assembly project, Assign the software project to your CA model to enable the CA debug configuration



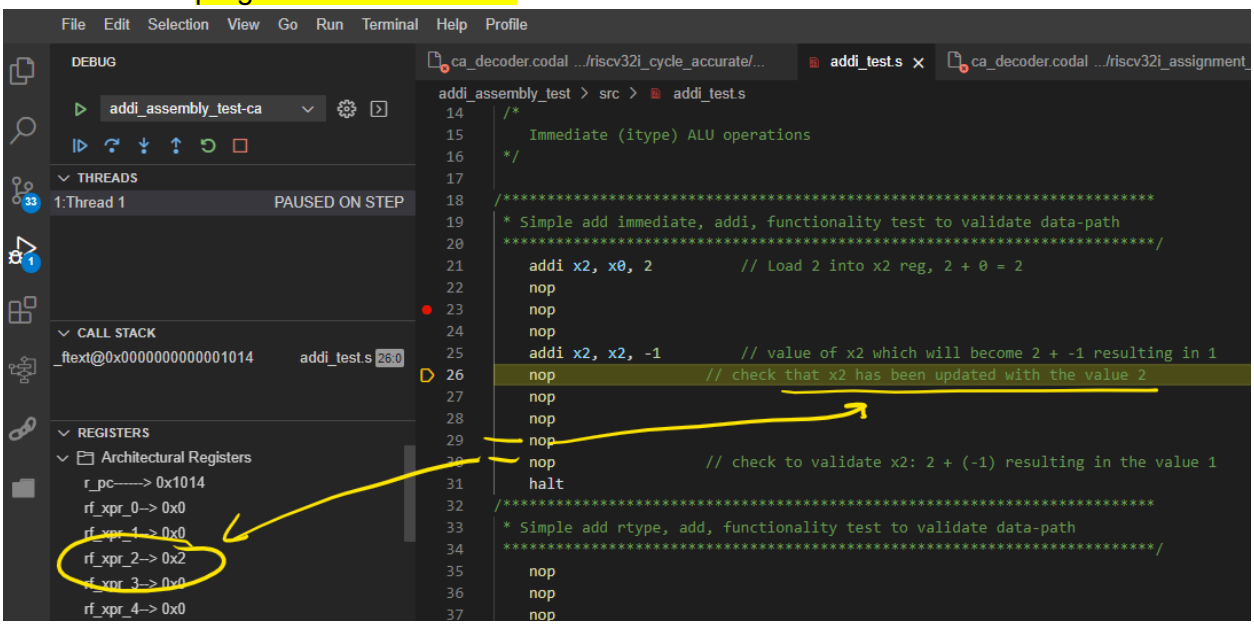
- **Click on Launch Debug and select your "addi_assembly_test-ca"**



- Debug addi_assembly_test.
- Note: You may get a warning that states there is a "Signed relocation truncation overflow." This is because we are adding -1 which is 0xFFFFFFFF as the immediate value. Since the immediate value is only 12 bits wide, the compiler thinks there is a truncation and sends a warning. Regardless, there is no issue if you get this warning since the value is truncated to 0xFFF which is still -1.



- Single step the assembly program by clicking on either the “Step Into,”  , or “Step Over,”  icons in the menu bar until it points to the nop instruction on line 26, right after addi x2, x2, -1. Verify that the value in the rf_xpr 2 instruction equals the value specified in the assembly program’s comment field



  - If they equal, you have successfully completed your first add immediate instruction. Congratulations.
  - If they are not equal, view the video on “Debugging your CodAL project”
- Now, single step to line 30, immediately before the halt instruction. Does the value in the register file match the expected value?
  - If yes equal, you have completed **Checkpoint 1**
  - If not, view the video on “Debugging your CodAL project”
  - You can click on resume,  , which will execute each instruction including the halt instruction or you can terminate,  , the debug session.

- With validating the data-path for an add immediate, addi, instruction, it is time to add the remaining i-type instructions
  - slti: set less than immediate
    - It is Best Practice to specify for all comparison operations whether the operand is signed or unsigned by using the proper type casting. For the slti, signed operation, both operands should be casted integer (int32).
      - ex: if ((int32) r_id_src1_operand > (int32) r_id_src2_operand) for comparison of signed data
  - sltiu: set less than immediate unsigned
    - It is Best Practice to specify for all comparison operations whether the operand is signed or unsigned by using the proper type casting. For the

sltiu, unsigned operation, both operands should be casted unsigned integer (uint32).

- xori: **bitwise** exclusive OR immediate
- ori: **bitwise** OR immediate
- andi: **bitwise** AND immediate

- All instructions of a single type, such as i-type or r-type use the same data-path and require the same control configuration, so adding additional instructions to an instruction type once validated is quite simple
  - First, you must add enums in aluop in ca_defines.hcodal for each of these new ALU operations. These enums will be used to tell the ALU what operations to perform
    - Note: An immediate add, addi, and a register-to-register add, add, perform the same ADD operation in the alu, so they will share the same enum aluop, ALU_ADD

```
// ALU function codes
enum aluop
{
    ALU_NOP = 0,        // In enums, you can specify a particular value to an element.  All other
    ALU_ADD,            // elements will have a value incremented by 1 unless assigned a specific value
};
```

  - Second, go to ca_decoder.codal to add a case statement for each of these new i-type instructions in the element i_hw_itype_alu along with the *ITYPE_ADDI* case statement setting s_id_aluop to the appropriate aluop enum

```
element i_hw_itype_alu
{
    use opc_itype_alu as opc;

    assembly { opc };
    binary { opc };

    semantics
    {
        // Register file write enable
        s_id_regwrite = true;

        // This switch statement assigns the alu operation to be performed by this instruction
        switch (opc)
        {
            case ITYPE_ADDI:
                s_id_aluop = ALU_ADD;
                break;
            default:
//                codasip_fatal(ALU_ERROR, "ALU Immediate Decoder case: %d\n", opc);
                break;
        };
```

  - The case statements test for the particular itype instruction opcodes. You can find the ITYPE opcode Definitions in model/share/include/opcodes.hcodal

```
// I-Type Immediate Opcodes (format)
// func3(bits 12:14) | opcode (bits 0:6)
```

```
enum ITYPE_OPCODES : uint10 {
    ITYPE_ADDI  = 0b0000010011,
    ITYPE_SLTI  = 0b0100010011,
    ITYPE_SLTIU = 0b0110010011,
    ITYPE_XORI  = 0b1000010011,
    ITYPE_ORI   = 0b1100010011,
    ITYPE_ANDI  = 0b1110010011,
```

- Next, you must add the aluop enum operations in the ALU switch statement located in the alu_operate event within ca_pipe3_ex.codal setting s_ex_alu_result to the proper value based on the case statement operation for src1 and src2 operands
  - For **unsigned** operations, such as sltiu, you must specify that the operand is unsigned by casting the operand unsigned
    - ex: result = ((uint32) soperand1 < (uint32) soperand2) ? true : false;

```
event alu_operate : pipeline(pipe.EX)
{
    semantics
    {
        // A switch statement will be used to determine and evaluate the ALU operation using
        // the aluop code provided by the ID pipeline register whose value is determined
        // by the instruction decoder.
        //
        // Hint:  Using the standard naming protocols for this project, what prefix would you add to
        // to aluop to build the complete name of input from ID pipeline register for this switch
        // statement?
        //
        // The value to assign in the case statements was added to ca_resources.codal earlier in
        // phase 5
        switch (r_id_aluop) {
            case ALU_ADD:
                s_ex_alu_result = s_ex_src1_operand + s_ex_src2_operand;
                break;
            default:
                s_ex_alu_result = 0;
                break;
        }
    };
};
```

Checkpoint 2: Validating the remaining i-type instructions

- You will use your regression test from phase 4 for Checkpoint 2. The first set of instructions in the regression test should be validating all the i-type instructions where no data hazard detection or forwarding is required.
- Debug your regression test and validate that each of the i-type instructions correctly evaluates by seeing the correct value change in the Architectural Registers/Register File view
  - If any of the instructions failed to correctly perform its operation, debug the operation. Below is a good debug process

        - Check that the ALU switch/case statement for the i-type instruction is specified as required for the operation. A common error in cut & paste is not modifying the pasted statement as required
        - Check that the instruction decoder is setting the appropriate aluop for the desired instruction
        - If both of above are as expected, place a breakpoint in the ALU switch/case statement and rerun the debugger to determine if this breakpoint is reached, and if yes, evaluate with src1 and src2 operands are as expected
        - Debug your project until your regression test successfully passes all i-type instructions. Note: The error may be in your regression test. If yes, correct your regression test.
- Upon successful completion through all i-type instructions, you have completed **Checkpoint 2**

- The next set of arithmetic operations are the register-to-register or r-type instructions. You will use a similar development strategy as the i-type instructions. First, implement a single instruction, ADD, and validate its data-path and control-signals before implementing the remaining r-type of instructions
- Go to ca_decoder.codal, and uncomment the complete element i_hw_rtype_alu

```
//element i_hw_rtype_alu
//{
//    use opc_rtype_alu as opc;
//
//    assembly { opc };
//    binary { opc };
//
//    semantics
//    {
//        // Register file write enable
//        s_id_regwrite =
//
//        // This switch statement assigns the alu operation to be performed by this instruction
//        switch (opc)
//        {
//
//        };
//
//        // Operand MUXes
//        s_id_alusrc1 =
//        s_id_alusrc2 =
//
//        // IMMEDIATE MUX select lines
//        s_id_imm_gen_sel =
//
//        // Is this a branch instruction to take a branch if branch taken
//        s_id_branch_inst =
//        s_id_jump_inst =
//
//        // Load / store control signals
```

```
//         s_id_mem_ops =
//         s_id_memread =
//
//         // HALT Command
//         s_id_halt =
//     };
//};
```

- Near the top, add i_hw_rtype_alu into the set of inst_decode along with the other set elements of i_halt and i_hw_itype_alu

```
// -----------------------------------------------------------------------------------------------
// Decoder
// -----------------------------------------------------------------------------------------------

set inst_decode = i_hw_halt,
        i_hw_itype_alu;
set inst_decode += i_hw_rtype_alu;
```

- For the element i_hw_rtype_alu, review each of the control-signals and set them to the values required to complete a register-to-register operation.
    - Helpful hints:
        - Options for s_id_alusrc1 & s_id_alusrc2
            - Register File: ALU_SRC1_SEL_RF or ALU_SRC2_SEL_RF
            - Immediate: ALU_SRC2_SEL_IMM
        - Options for s_id_mem_ops
        - MEM_NOP: Not a memory operation, r-type are arithmetic operations
    - Reviewing your block diagram from phase 4 as well as the block diagrams throughout this phase may be helpful in assigning these control-signals as well as reviewing sections 2.2, 2.3, and 2.4 of the RISCV Instruction Manual
        - https://riscv.org/wp-content/uploads/2019/06/riscv-spec.pdf
- For the i_hw_rtype_alu switch(opc) statement, assign a single case statement, *RTYPE_ADD*, for *ALU_ADD* properly assigning the enum to s_id_aluop
    - Note: An add immediate, addi, and r-type, add, perform the same operation, add. The difference between the two instructions is the operands. For addi, src2 operand is ALU_SRC2_SEL_IMM where for the r-type add, src2 operand is ALU_SRC2_SEL_RF, the register file. To minimize the number of gates/circuits for the design, the aluop for both the r-type and i-type will reuse the same ALU enum (operation) to implement the add. The control-signals for s_id_alusrc2 and s_id_imm_gen_sel defined by addi and add in the instruction decoder change the behavior of the ALU add function.
- It is best practice to add the smallest amount to your design and validate before adding additional instructions or features. Minimizing the amount of new code focuses where to debug if the new code or feature fails validation

---

Checkpoint 3: Validating the a single r-type instruction add

- Go to the addi_assembly_test program, and comment out the first halt instruction so that the program can proceed to validate the r-type add instruction
- Step through the addi_assembly_test program and validate that the registers are updated correctly using the Registers:Architectural Registers view with the help of the //comment statements
- If the registers update as expected, you are ready to proceed onto adding the remaining r-type of instructions
    - If not, review the control signals and if required, place breakpoints within your CodAL project to view the r-type instruction proceed through your pipeline
    - If yes, you have successfully completed **Checkpoint 3**

- With validating the data-path for an r-type add instruction, it is time to add the remaining r-type instructions
    - sub: subtract
    - sll: shift logical left
    - slt: set less than
    - sltu: set less than unsigned
    - xor: bitwise exclusive OR
    - srl: shift right logical
        - logical shift inserts 0s into the upper bits as the value is shifted to the right
        - The insertions of 0s is accomplished by casting the operand to be shifted as unsigned integer (uint32)
    - sra: shift right arithmetic
        - arithmetic shift copies the sign-bit, the most significant bit, into the upper bits as the value is shifted to the right
        - The copying of the signed bit is specified by casting the operand to be shifted as an integer (int32)
    - or: bitwise OR immediate
    - and: bitwise AND immediate
- There are additional ALU operations defined by the r-type of instructions. You will need to add additional aluop enums in ca_defines.hcodal and create additional switch/case statements for these enums in the ALU switch statement in alu_operate event in ca_pipe3_ex.codal highlighted in purple.
    - **Best Coding Practice:** With the result field only 32-bits, shift operations should be limited to the number of possible shifts, 32. This can be accomplished by type casting the operand used to determine the number of bits to shift by the number of bits required to obtain the maximum number of shifts. In binary, 5-bits are required to define 32, 0 thru 31. The operand defining the number of bits to shift should be type casted to 5-bits, (uint5).
- Next, go to ca_decoder.codal to add a case statement for each of these new r-type instructions in the element i_hw_rtype_alu along with the *RTYPE_ADD* case statement setting s_id_aluop to the appropriate aluop enum
    - Remember, each of these case statements are testing for a rtype opcode. You can find the RTYPE opcodes in model/share/include/opcodes.hcodal

- It is time to test out the r-type instructions

Checkpoint 4: Validating the remaining r-type instructions

- You will use your regression test from phase 4 for **Checkpoint 4**. The first set of instructions are the i-type instructions and the second set are the r-type instructions where no data hazard detection or forwarding is required.
    - If you have a halt statement between the i-type and r-type of instructions, you should now comment it out
- Debug your regression test and validate that each of the r-type instruction correctly evaluates by seeing the correct value change in the Architectural Registers/Register File view
    - If any of the instructions failed to correctly perform its operation, debug the operation. Below is a good debug process
        - Check that the ALU switch/case statement for the r-type instruction is specified as required for the operation. A common error in cut & paste is not modifying the pasted statement as required
        - Check that the instruction decoder is setting the appropriate aluop for the desired instruction
        - If both of above are as expected, place a breakpoint in the ALU switch/case statement and rerun the debugger to determine if this breakpoint is reached, and if yes, evaluate with src1 and src2 operands are as expected
        - Debug your project until your regression test successfully passes all r-type instructions. Note: The error may be in your regression test. If yes, correct your regression test.
    - Note: The i-type of instructions are still included in the regression test to validate that the addition of the r-type instructions did not “break” the functionality of the i-type instructions
- Upon successful completion through the regression test through the r-type of instructions, you have validate successful completion of both the i-type and r-type instructions, you have completed **Checkpoint 4**

- The last set of computational instructions are the r-type immediate
    - slli: shift left logical immediate
    - srli: shift right logical immediate
    - srai: shift right arithmetic immediate
- You will add all of three of these instructions at this phase of the project.
- The ALU operation of all three of these instructions were added in the previous section and validated in Checkpoint 4. The difference between the standard r-type version of these instructions is that both operands are from the Register File. The control-signals for both of them are the same, except the control-signal for the src2 operand will be an immediate and the control-signal selecting the r-type immediate.

- Uncomment the complete element i_hw_rtype_shift and near the top, add i_hw_rtype_shift into the set of inst_decode along with the other set elements of i_halt, i_hw_itype_alu, and i_hw_rtype_alu as you did to add i_hw_rtype_alu to the set of inst_decode
- For the element i_hw_rtype_shift, review each of the control-signals and set them to the values required to complete a register-to-register(immediate) operation.
    - Reviewing your block diagram from phase 3 may be helpful in assigning these control-signals as well as reviewing sections 2.2, 2.3, and 2.4 of the RISCV Instruction Manual
        - https://riscv.org/wp-content/uploads/2019/06/riscv-spec.pdf
- For the i_hw_rtype_shift switch(opc) statement, assign a case statements for *ALU_SLLI, ALU_SRLI, and ALU_SRAI* properly assigning the enum to s_id_aluop
    - Remember, each of these case statements are testing for a rtype opcode. You can find the RTYPE opcodes in model/share/include/opcodes.hcodal
- It is time to validate the last set of computational RISC-V instructions

---

Checkpoint 5: Validating the r-type immediate instructions

- You will use your regression test from **Checkpoint 4**.
- Now, comment out the halt instruction after the r-type instructions to enable the regression test to execute the r-type immediate instructions: slli, srli, and srai.
- Note: The i-type and r-type set of instructions are still included in the regression test to validate that the addition of the r-type immediate instructions did not "break" the functionality of the standard i-type and r-type instructions
    - If you have a halt statement between the r-type and r-type immediate instructions, you should now comment it out
- Debug your regression test and validate that each of the r-type immediate instructions correctly evaluates by seeing the correct value change in the Architectural Registers/Register File view
    - If any of the instructions failed to correctly perform its operation, debug the operation. Below is a good debug process
        - Check that the ALU switch/case statement for the r-type instruction is specified as required for the operation. A common error in cut & paste is not modifying the pasted statement as required
        - Check that the instruction decoder is setting the appropriate aluop for the desired instruction
        - If both of above are as expected, place a breakpoint in the ALU switch/case statement and rerun the debugger to determine if this breakpoint is reached, and if yes, evaluate with src1 and src2 operands are as expected
        - Debug your project until your regression test successfully passes all r-type instructions. Note: The error may be in your regression test. If yes, correct your regression test.

- Upon successful completion through r-type immediate instructions in the regression test, you have validate successful completion of all the RISC-V computational instructions
- You have completed **Checkpoint 5**

- Completing the phase
    - Export your regression test
    - Submit your exported regression test for grading
    - Export your phase
    - Submit your exported phase for grading

Appendix A: YouTube videos for Phase 5

Phase Videos:

- **Phase 5: Data Path & Instruction Decoder**
    - A processor comprises a data-path which transports instructions, operands, and results while control-signals determine the flow of the data-path and how the information in the data-path is manipulated by processing elements such as the Arithmetic Logic Unit (ALU). In this phase, you will be creating the ITYPE and RTYPE data-paths for a 5-stage RISCV32I processor as well as adding all the ITYPE and RTYPE instructions in the Instruction Decoder.
    - https://www.youtube.com/watch?v=qkMs15eVAgc&list=PLTUn6Ox9e6q2ienoqI3KClFtRPMqO28uj&index=3
- **Phase 5: Parsing using Bit Extraction and Concatenation**
    - An instruction fetched by a processor must be parsed to perform the individual operations defined by the instruction format bit fields. For example, in RISCV, if an instruction requires the source 1 operator, it will always extract from the RISCV instruction or machine code bits 19 thru 15 to index the Register File to read the source 1 value. Other signals required by the instruction decode pipeline will require multiple bit fields to be concatenated together. This video will show you how to extract and concatenate bit fields from 32-bit instructions into the individual signals that are required by your 5-stage processor pipeline.
    - https://www.youtube.com/watch?v=emiv0cR__Q8&list=PLTUn6Ox9e6q2ienoqI3KClFtRPMqO28uj&index=4
- **Phase 5: Signal and Register Types**
    - Similar to other programming languages, processor elements must be declared. CodAL, a processor description language, has several new types of variables compared to c or c++. Signal types are the wires connecting one part of the circuit to the other while registers are sequential storage elements that retain state between clock edges. The pipeline register is a special type of register which includes two additional inputs, Clock EN(able) and Clear. This video shows you how to declare these different types of processor elements.
    - https://www.youtube.com/watch?v=Snb-IiOF2L0&list=PLTUn6Ox9e6q2ienoqI3KClFtRPMqO28uj&index=5
- **Phase 5: Checkpoint 1: Using the addi_assembly_test**
    - In Phase 5, Checkpoint 1, you will be using the addi_assembly_test provided to validate your ITYPE data path. This video will show you how to launch the addi_assembly_test using your Cycle Accurate (CA) model and how the Architectural & Microarchitectural registers can be used to validate & debug your ITYPE data path.
    - https://www.youtube.com/watch?v=G5SKv5qZHeg&list=PLTUn6Ox9e6q2ienoqI3KClFtRPMqO28uj&index=6
- **Phase 5: Imaging Software into Hardware**
    - The end goal of designing a processor is to implement it in hardware. The goal of the hardware design may be smaller area, lower power, or higher frequency of operation. To achieve these goals, it is important to understand how a High Level Synthesis (HLS) software construct relates to its hardware implementation.

This video will describe the hardware construct of a software switch statement and how it can be used to reduce the time to calculate a potential signal, effectively increasing the frequency of operation.
    - https://www.youtube.com/watch?v=7wxHO-ves68&list=PLTUn6Ox9e6q2ienoqI3KClFtRPMqO28uj&index=7

Frequently Asked Questions (FAQs) Videos

- It is intended for students to provide them **real-time support** who have been assigned a project-based learning phase, based on the Codasip Curriculum.
- **Decoding a CA Compiler Error Message**
    - Getting as quickly as possible to an error in a project will speed up your debug and project development time, especially if you have a project which comprises many files. This video will teach you how to decode or dissect your Error Message while compiling your Cycle Accurate (CA) model enabling you to locate the error with helpful information on the error type.
    - https://www.youtube.com/watch?v=z2EHv2a1Qj8&list=PLTUn6Ox9e6q1ii0fp-N_GDPZjAtkDZmqe&index=7
- **Functional Unit has to be Combinational Logic**
    - In a standard processor design, each pipeline stage must be deterministic which infers if I have identical input conditions at different times, at each of these times you will get the exact same result or output. The error condition related to Functional Unit has to be Combinational Logic is stating that the design is non-deterministic. This video will teach you how to find the signal or register that is non-deterministic and where to locate the instants.
    - https://www.youtube.com/watch?v=AbnXW6bGnKw&list=PLTUn6Ox9e6q1ii0fp-N_GDPZjAtkDZmqe&index=8
- **Not Declared in this Scope**
    - Just like programming in c or c++, a signal or register cannot be used until it is declared. This video will show you the two most common causes of the “Not Declared in this Scope” errors and where to resolve them.
    - https://www.youtube.com/watch?v=c0BCuJlJiw4&list=PLTUn6Ox9e6q1ii0fp-N_GDPZjAtkDZmqe&index=9
- **Why won't my breakpoint go where I want it to go?**
    - You are trying to debug your Cycle Accurate (CA) model, and you find that you are breaking below the breakpoint or while in the debugger, you find you cannot place a breakpoint in a portion of your CA model. This video explores why these situations occur and how to resolve them.
    - https://www.youtube.com/watch?v=_WvUlr0L0jc&list=PLTUn6Ox9e6q1ii0fp-N_GDPZjAtkDZmqe&index=10