# Computer Organization
# 5-stage RISCV32I Processor
# Phase 8: Upper Immediate, Load and Store Instructions
# Fall 2022

Due Dates:

- This is a 1-week phase
- Due date: Monday November 28th 2022 11:59pm

Instructions for Schematics:

- Copy the schematic standardname7.xml to standardname8.xml. Open the schematic in diagrams.net

- **Utype Instructions:**
- The AUIPC uses the PC as the src1 into the ALU.
  - To implement this instruction, in the execute stage connect the pc into the src1 mux after the data hazard mux.
- The LUI simply uses the immediate plus 0 to load the upper immediate into the destination register.
  - To implement this instruction, in the execute stage add a third input into the src1 mux (after the data hazard mux) to use the value zero as the src1 input.
- The rest of the functionality of these instructions is already handled by the existing components of the schematics so no other changes are required.

- **Load and Store Instructions:**
- The Codasip memory model is split into two pieces, the address phase in the EX stage, and the data phase in the ME stage (just as the Instruction Memory is split between the IF and ID stages).
- Add the Memory control block (the address phase of the memory interface) in the execute stage. The memory control block in the EX stage will receive four inputs (memory operation type, the memory operation, the memory address, and the size of the memory operation).
  - Route the ALU output to the address to the Memory control block in the execute stage. NOTE: the ALU output needs to still be pipelined to the ME stage.
  - Add all input signals into the memory control block.
  - The write data for a store comes from the data hazard forwarding mux for src2 so pipeline this value into the ME stage as r_ex_wtdat and connect it as an input of an encode data block and then connect the output to a new pipeline register to pass the rs2 value into the ME stage.
    - Use the memop value to select the correct encoding of the data
- Add the Memory Data Block in the memory stage

    - For a load instruction, the data read from memory should be sent into a component called data decode to adjust the block offset and the value returned into the register file.
        - Add the data decode component in the memory stage
        - The read data output of the decode component should be pipelined to the WB stage as r_me_memdat.
        - Use the memop value to select the correct decoding of the data
- In the writeback stage add a mux to select the data read from memory (r_me_memdat) or the data computed in the execute stage (r_me_data_result).
- As described in section 4.7 of the textbook, a Data Hazard can occur on a load instruction which cannot be resolved by the forwarding logic we have implemented. This requires the creation of a stall in the pipeline.
    - Add a control component named LDHAZ in the ID stage which creates the signal s_id_loadhaz. Connect the required inputs to identify the load data hazard. The signal s_id_loadhaz will be used in the stall logic in Codasip which is not shown in the schematics.
- Submit the Updated Schematic
    - Although not required, it is recommended that you submit the updated schematic as soon as it is complete, as standardname8.xml. This will allow for feedback prior to creating the Codasip code. Submit the file via Canvas.

Instructions for Codasip:

- You will continue to build upon your phase 7 project. If required, import your phase 7 processor CodAL project.

**U type Instructions:**

- Import the lui_auipc_assembly_test into your Codasip workspace
    - git clone https://github.com/CompOrg-RISCV/lui_auipc_assembly_test.git
- Changes to the ca_defines.hcodal file:
    - Add two new enums for the src1 select values: ALU_SRC1_SEL_ZERO (for the LUI instruction) and the ALU_SRC1_SEL_PC (for the AUIPC instruction).
- Changes to the execute stage codal file:
    - Add the corresponding case statements for the two new enums for the src1 select mux.
- Changes to the decoder codal file:
    - Find and uncomment the utype element named i_hw_utype_ops.
    - Add all the control logic signal phases accordingly keeping in mind that the main difference between AUIPC and LUI is the src1 that is selected.
    - Add the utype element to the set of instructions to decode (inst_decode) on line 36.

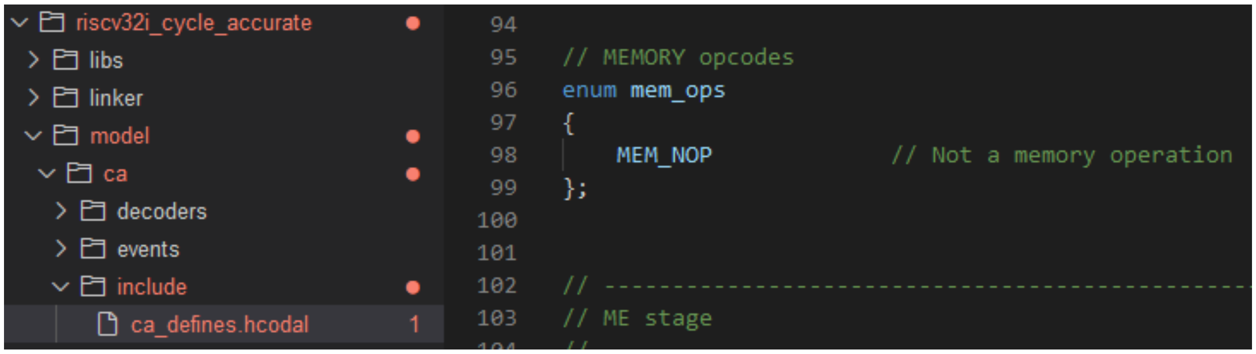
## Checkpoint 1: Validating *lui* and *auipc* instructions

- Import and configure the lui_aupic_assembly_test for your workspace/project

- Debug lui_aupic_assembly_test
  - Step through your code until it fails or it reaches the end of the program halt statement.
    - This assembly test uses branch tests to determine a failure, a branch to the label FAIL
    - Data-hazard and forwarding is required for this assembly test
- Once you have completely passed the lui_aupic_assembly_test, you have completed **Checkpoint 1**.

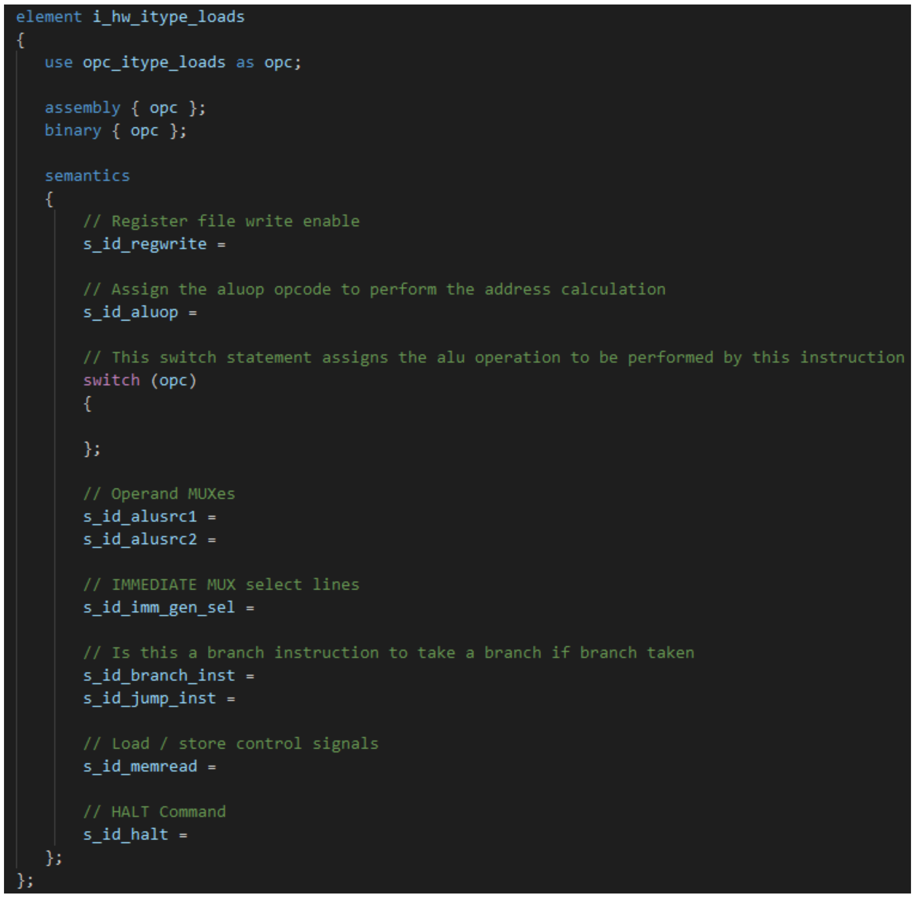
**Load and Store Instructions:**

- Import the store_load_assembly_test into your Codasip workspace
  - git clone https://github.com/tamisil/store_load_assembly_test.git
- The instruction decoder will encode the memory operation and memory size into a memory opcode, memop signal, which will be passed into the EX and MEM stages. The instructions you will be implementing are lb, lbu, lh, lhu, lw, sb, sh, sw. These instructions are defined in the isa.codal file inside of "project"/model/share/isa/ (search for "itype_loads" and "stype")
- Changes to the ca_defines.hcodal file:
  - Add the 8 corresponding mem_ops enum by adding an enum for each type of load and store instruction plus MEM_NOP.  MEM_NOP must equate to 0, so that when the pipeline register is cleared, it would clear the memory opcode to 0. Choose any name that makes the most sense, but you should have 9 different names in this enum (including the MEM_NOP).
    - By making MEM_NOP as the first enum element and equating it to 0, MEM_NOP



```
// MEMORY opcodes
enum mem_ops
{
    MEM_NOP                 // Not a memory operation
};


// -----------------------------------------------
// ME stage
```

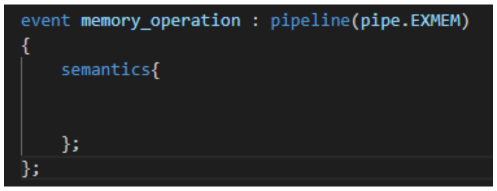
- Changes to the ca_decoder.codal:
  - Add the elements for these instructions into the Instruction Decoder by searching for "itype_loads" and "stype" until you find the element i_hw_itype_loads and i_hw_stype_store and uncommenting both declarations



```
element i_hw_itype_loads
{
   use opc_itype_loads as opc;

   assembly { opc };
   binary { opc };

   semantics
   {
       // Register file write enable
       s_id_regwrite =

       // Assign the aluop opcode to perform the address calculation
       s_id_aluop =

       // 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_memread =

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

- The first thing to add in these elements is the assigning of the s_id_aluop signal
  - You will notice that this phase is outside of the switch statements that have been previously used. s_id_aluop is not required to be in the switch statements because all loads and stores perform the same operation through the ALU, the addition, ALU_ADD, of the address.
- For each of these elements, the s_id_mem_ops will be assigned in the “switch” (opc) statement” instead of a single phase for the entire element. Assign appropriate mem_ops enum that you have created for each of the load and store instructions,
- With the s_id_aluop and s_id_mem_ops signals assigned, the remaining control-signals must be declared using the appropriate define from ca_defines.hcodal or the appropriate boolean value (true/false)
  - s_id_regwrite - Does the “itype_loads” operations update the rd, destination register? Does the “stype” operations update the rd, destination register?
  - s_id_alusrc1 - What is the source of the src1, rs1, operand for loads and stores?
  - s_id_alusrc2 - What is the source of the src2, rs2, operand for loads and stores?
  - s_id_imm_gen_sel - What immediate value should be passed to the EX stage for loads and for stores?

        - s_id_branch_inst - Is it a branch operation?
        - s_id_jump_inst - Is it a jump operation?
        - s_id_memread - Is it a load memory operation?
        - s_id_halt - Is it a halt operation?
        - s_id_branchops: is the instruction a branch operation?
        - s_id_rfwtsel: is the data written in the register file the PC+4 or the alu result?
    - With the i_hw_itype_loads and i_hw_stype elements fully defined, it is time to include them in the list of allowable instructions to decode. At the top of ca_decoders.codal, add the i_hw_itype_loads and i_hw_stype to the "inst_decode set" (line 36 in the ca_decoder.codal file)
- Changes to the ca_pipe1_if.codal file:
    - Add the logic that causes the program counter (r_pc) not to be updated when a load hazard is detected.
    - Note that the load hazard detection sets the signal called s_id_loadhaz (you will define this signal in ca_resources.hcodal and assign it in the decode stage)
- Changes to the ca_pipe2_id.codal file:
    - To implement the load hazard detection unit we need to check if the src1 or src2 is the same as a destination register of another instruction that is in the execute stage AND the instruction in the execute stage is a load instruction. In this case we want to add a NOP (aka a bubble) into the pipeline to enable the correct data to be forwarded.
        - Add the checks to set the load hazard signal (s_id_loadhaz) to detect when the hazard occurs.
- Changes to the ca_pipe3_ex.codal file:
    - Add the event memory_operation and declare it in event ex as you did for the branch_operation event in the last phase
    - Add a call to the memory_operation event in the ex event after the alu_operate event call



```
event memory_operation : pipeline(pipe.EXMEM)
{
    semantics{


    };
};
```

        - memory_operation event must be called after the calculation of the store and load address in the ALU unit.
    - CodAL has a function called transport to define what will occur on the memory bus for each phase of the memory operation. Per AHB3-Lite specification, there are two phases:
        - Address phase: CP_PHS_ADDRESS
        - Data phase: CP_PHS_DATA

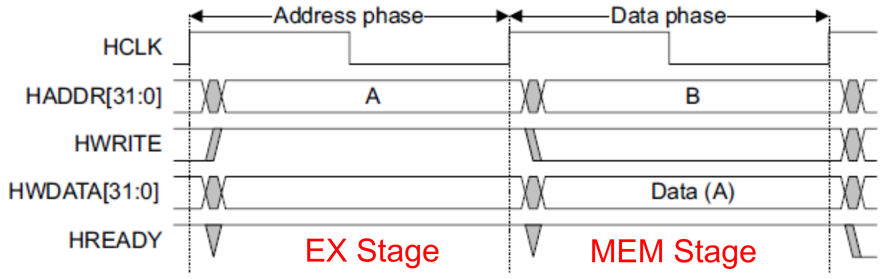
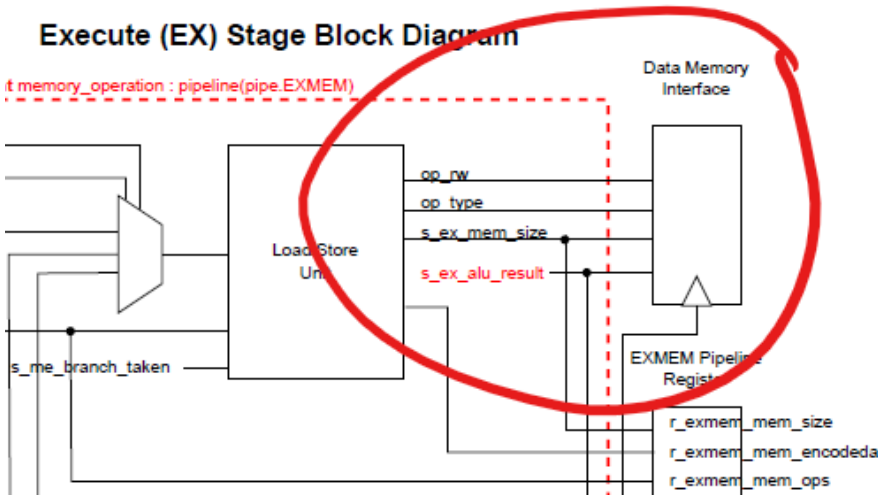


**Figure 3-2 Write transfer**

- The transport input arguments for the address phase are:
  - ldst.transport(*phase, op_type, op_rw, memory_address, memory op size);*
- ldst is the memory interface defined for the data memory in model/share/resources/interface.codal
- You can learn more about the *address phase* of the transport function supporting the AHB3-LITE bus by searching for “AHB3_LITE Transports” in the Codasip CodAL Language Reference Manual
- In the memory operation event create the necessary local signals for op_rw, op_type. These signals will be used when you call the memory interface function for the load instruction address phase.
  - op_rw (uint1): Is it a load, store, or idle (no memory request) operation?
  - op_type (uint2): Is the memory request idle, burst, or non-sequential.
- 
- For reference as to how to create a local signal refer to the fetch stage and how if_status is defined.
- Using the mem_ops pipeline register passed into the EX stage from the ID stage assign the required local variables op_rw, op_type, and the s_ex_mem_size signal using a switch statement to decode all the mem_ops enums defined in ca_define.codal which includes both store and load operations
  - The values assigned to these signals will be defined by the AHB3-Lite interface
    - op_rw: load = CP_AHB_READ, store = CP_AHB_WRITE, else = DONT_CARE defined as 0
    - op_type: load = CP_AHB_NONSEQ (random access), store = CP_AHB_NONSEQ (random access), else CP_AHB_IDLE (no bus operation)
    - s_ex_mem_size: byte = CP_AHB_SIZE_8 (1-byte), half-word = CP_AHB_SIZE_16 (2-bytes), word = CP_AHPB_SIZE_32 (4-bytes)



- After you have assigned all the signals for the transport function in the memory_operation event, add ldst.transport(*phase, op_type, op_rw, memory_address, memory op size);* to initiate the memory bus operation using the proper signal names for your project

- Store instructions use the rs2 Register File output data as the data to be written into memory. As with the rest of the instructions, this operand requires data hazard detection and data forwarding, which was already implemented in phase 6. For the data value to be written into memory you will use the output of the data hazard forwarding mux for src 2. The Codasip memory interface requires the data to be properly "encoded" by placing it in the correct byte location on the memory bus. For example, in a Little Endian machine, the least significant byte is located in the least significant bit positions. If a byte store whose value is 0xaa is to be written to the 1 location, 0xaa must be placed on the bit location of 15..8

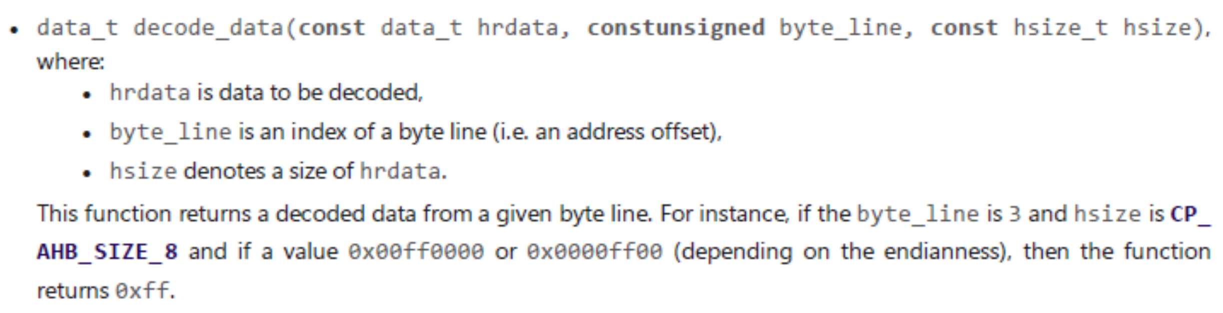
| Bytes | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Bits | 31 .. 24 | 23..16 | 15..8 | 7..0 |

- CodAL has a encode_data function whose return value and input arguments are:

- `data_t encode_data(const data_t hwdata, constunsigned byte_line, const hsize_t hsize)`, where:
  - `hwdata` is data to be encoded,
  - `byte_line` is an index of a byte line (i.e. an address offset),
  - `hsize` denotes a size of `hwdata`.

  This function returns an encoded data to a proper byte line. If the value `0xff` should be encoded to an address `0x3`, and `hsize` is **`CP_AHB_SIZE_8`**, then `encode_data` creates either `0x00ff0000` or `0x0000ff00` (depending on the endiannes).

        - From Codasip's CodAL Language Reference Manual
            - *encoded_data* = ldst.encode_data(*store value*, *memory_address (s_ex_alu_result)*, *memory op size*);
            - Add a call to the encode function after your transport request in memory_operation
            - You can learn more about the encode_data function within the "AHB3_LITE Transports" section of the Codasip CodAL Language Reference Manual
    - You have now initiated the memory bus request. It is now time to pass all the required signals to complete the memory bus operation in the MEM stage.
- Changes to ca_pipe4_me.codal:
    - You will use the transport function to retrieve the data from the memory bus using the CP_PHS_DATA phase definition
        - For data phase, the transport function has the following arguments
            - ldst.transport(*data phase constant* (input), *data ready signal*(return), *data response* signal (return), *load data signal* (output), *encoded data to store pipeline register* (input);
            - In the ME event create local signals for the return values of data_ready and data_response. These signals will be used when you implement the load instruction data phase.
            - For reference as to how to create a local signal refer to the fetch stage and how if_status is defined.
        - In the me event, call the ldst.transport data phase function using the local variables, the encoded data that you pipelined from the EX stage and the additional signals you created in the ca_resources.codal file.
            - You have now placed data onto the memory bus which will be latched into memory on the next rising edge of the CPU clock cycle
        - To complete the load transport an inverse operation to the encode_data function will be used. The encode_data function was performed in the EX stage to place the store byte and halfwords in the correct position on the memory bus for store instructions. A decode_data function will be used to take the data off the memory bus and align it with the least significant bit for all load instructions.


- `data_t decode_data(const data_t hrdata, constunsigned byte_line, const hsize_t hsize)`, where:
    - `hrdata` is data to be decoded,
    - `byte_line` is an index of a byte line (i.e. an address offset),
    - `hsize` denotes a size of `hrdata`.

  This function returns a decoded data from a given byte line. For instance, if the `byte_line` is `3` and `hsize` is **CP_AHB_SIZE_8** and if a value `0x00ff0000` or `0x0000ff00` (depending on the endianness), then the function returns `0xff`.

From Codasip's CodAL Language Reference Manual

    - In the me event, create a local *decoded_data* signal for the return of the decode_data function. The return value will be a 32-bit data word matching the width of the memory bus.
        - The decoded_data function takes value from the memory bus and aligns it with the least significant bit, bit 0.
    - You can learn more about the decode_data function within the "AHB3_LITE Transports" section of the Codasip CodAL Language Reference Manual
        - After the load transport function call, add the decode_data function call

*decoded_data signal* = ldst.decode_data(*load data signal from transport function*, *memory_address (r_ex_alu_result)*, *memory op size (r_ex_mem_size)*);

- The decoded data is aligned to least significant byte. It now must be sign-extended if the load instruction requires it (i.e. for lb and lh only).

| Bytes | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Bits | 31 .. 24 | 23..16 | 15..8 | 7..0 |

- Using the r_ex_mem_ops signal create a switch statement where each load operation properly sign extends the *decoded_data* signal per the mem_op onto the s_me_mem_result signal.
    - As done with the immediate values in Phase 5, sign-extension is performed by first type casting the value to its original data width as an integer and then type casting to the desired data word width as an integer

```
    case MEM_LB:
        s_me_mem_result = (int32)(int8) decoded_data;
        break;
```

    - No sign-extension is performed when first type casting the value to its original data width as an unsigned integer and then unsigned type casting to the desired data word width

```
    case MEM_LBU:
        s_me_mem_result = (uint32)(uint8) decoded_data;
        break;
```

- Add logic in the me event to assert (true) the memory bus stall (s_me_stall) upon the following condition. Please refer to the transport arguments above.
        - If data_ready is false, not ready to accept a new operation
        - And, if data_response is false, the memory has not finished the memory transfer or there is a memory bus failure

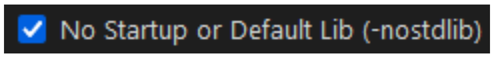
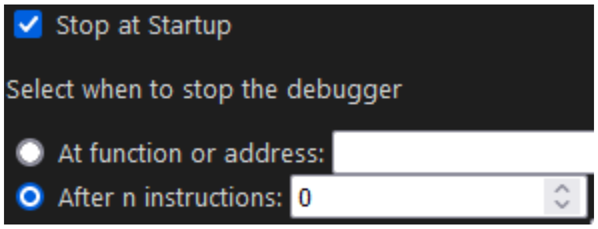
- Changes to ca_pipe5_wb.codal:
    - Add a mux to select which piece of data to write into the register file. The mux selects between the ALU result or the Memory data read.
    - You will need to update the name of the data write input of the register file to match that of the mux output. This change will also require you to change the forwarded value into the ID and EX stages.
    - The select bit (the s_id_memread) is assigned in the decoder.

- Changes to ca_resources.codal:
    - Inside the memory operation you will use the memory interface provided by Codasip.
    - Create the necessary signals for the codasip memory interface:
        - Memory address of the operation. s_ex_address (value from the alu_operate event in the execute stage)
        - Size of the operation (byte, half-word, or word). s_ex_mem_size (uint2).
        - Data from the processor for store operations and data from the memory for loads.
    - Create a signal and corresponding pipeline register for the returned value of the *encoded_data* function so that it can be used by the ex_output() event to pass the encoded data to the MEM stage.
    - Create the s_id_loadhaz signal that will enable the identification of a data hazard with the output of a load instruction that would require a one cycle stall.
    - Create any additional resources that are required for the memory_operation event as well as to pass the required signals along to the MEM stage
    - Add a resource that will be used to stall the earlier pipeline stages until the memory bus responds with HREADY, a memory bus stall (s_me_stall). This resource bit a single bit wide signal that informs whether the pipeline is to be stalled due to a data memory bus operation if true and no stall is required if false
- Changes to ca_pipe_control.codal:
    - Add an else if statement (after the if statement created for when branches are taken) to add the logic that will create a stall when the memory interface is not ready. This will entail checking the value of the signal that indicates if the memory should stall or not.
    - The command to add a stall in a pipeline stage is: pipe.[stage name].stall() (example: pipe.IF.stall()). Keep in mind that if the value in memory is not ready and all stages to the left have to stall the writeback stage has to stall as well to preserve the semantics of the data hazard forwarding logic.
    - You will be using the register resource provided, r_id_buffer_full, in ca_resources.codal to inform the ID stage that the previous cycle was stalled and that the instruction to be used will be the previous saved instruction.  It is a boolean value, true = stalled and false = not stalled.
        - For each if' and else if in pipeline control, specify whether the boolean register should be set to be true or false.  It will be true if the pipeline is stalled.  The else statement which represents no branches and no stalls should set it to false.
    - The following resources have been provided in ca_resources.codal to enable to preserve a fetched instruction

        - r_id_instr_preserved: preserve the current instruction fetched
        - r_id_pc_preserved: preserve the current pc of instruction fetched
        - r_id_buffer_full: instruction buffer is full, stall in progress
    - Add an else if clause to the stall/clear IF statement that will implement the desired stalls and NOP bubble insertions with assigning the ID pipe stall register to true when the s_id_loadhaz signal is set to true.

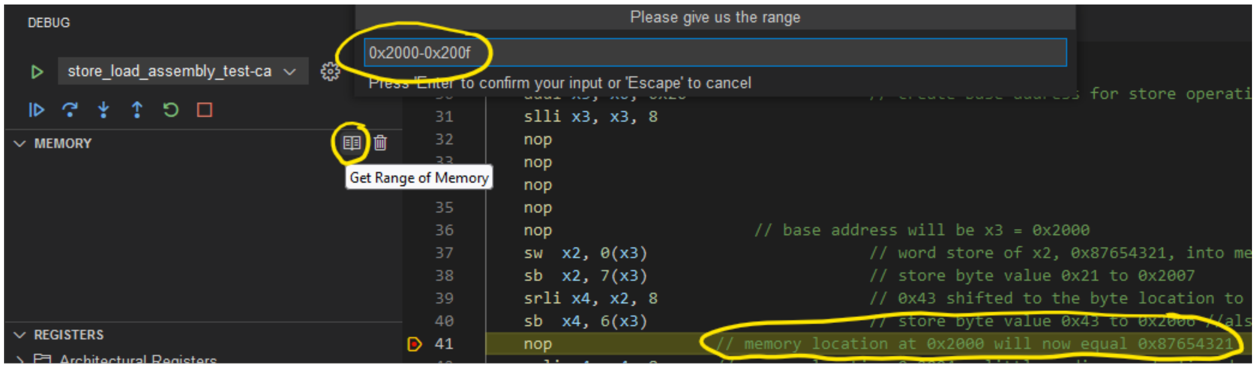
---

## Checkpoint 1: Validating store operations

- **Gen**/Build your **SDK(ca)** model
- In the Workflow, , perspective with the store_load_assembly_test project selected, using **Assign SDK** assign your **SDK(ia)** model to the software project
- Set the **Compiler Configuration** settings, , to the standard assembly project settings
    - 
      ☑ No Startup or Default Lib (-nostdlib)
- **Compile** store_load_assembly_test
- Set the **Debug Configurations** settings, , to the standard assembly project settings
    - 
      ☑ Stop at Startup

      Select when to stop the debugger

      ○ At function or address:
      ◉ After n instructions: 0
- In the Workflow, , perspective with the store_load_assembly_test project selected, using **Assign SDK** assign your **SDK(ca)** model to the software project
- Note: If you need to recompile the assembly project, you will need to reassign the software project to your SDK(ia) model
- Time to **Launch Debug**, , of your CA model to step through your project and to evaluate that the store operations occur as expected per the comments in the test program
- You can view a memory location by selecting a range of memory to view while in the debug perspective,
        - In the Enter expression, type in the address 0x2000 which is the starting memory address that this test program will store values



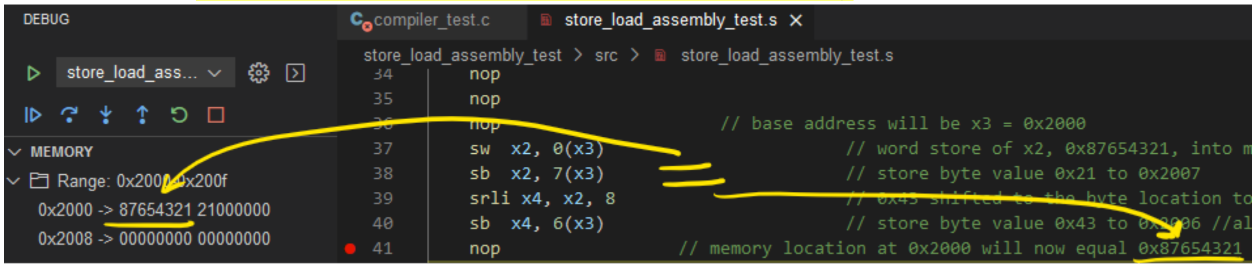
Please give us the range

0x2000-0x200f

Press 'Enter' to confirm your input or 'Escape' to cancel

```
slli x3, x3, 8
nop
nop
nop
nop
nop                     // base address will be x3 = 0x2000
sw  x2, 0(x3)                    // word store of x2, 0x87654321, into me
sb  x2, 7(x3)                    // store byte value 0x21 to 0x2007
srli x4, x2, 8                   // 0x43 shifted to the byte location to
sb  x4, 6(x3)                    // store byte value 0x43 to 0x2006 //als
nop                   // memory location at 0x2000 will now equal 0x87654321
```

- Run your code to each //comment to verify that a register has the value as expected as well as the value stored in memory


DEBUG
store_load_ass...
MEMORY
compiler_test.c
store_load_assembly_test.s
store_load_assembly_test > src > store_load_assembly_test.s

```
Range: 0x2000-0x200f
0x2000 -> 87654321 21000000
0x2008 -> 00000000 00000000
```

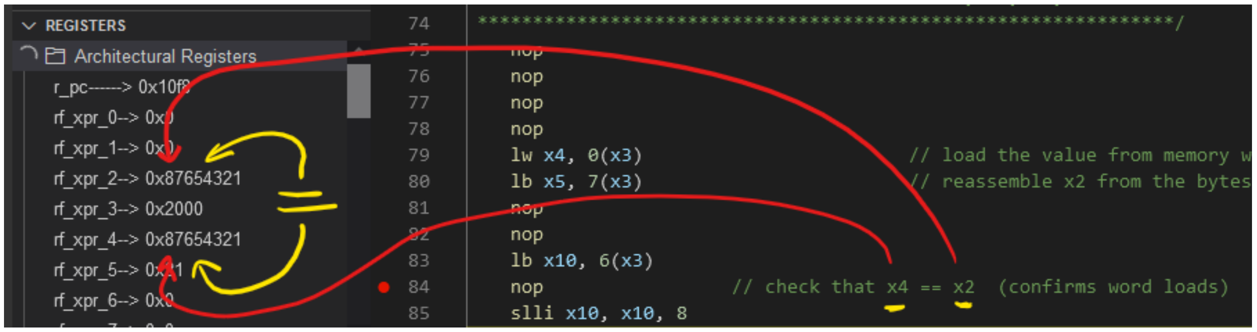
```
nop
nop
nop                     // base address will be x3 = 0x2000
sw  x2, 0(x3)                 // word store of x2, 0x87654321, into m
sb  x2, 7(x3)                 // store byte value 0x21 to 0x2007
srli x4, x2, 8                // 0x43 shifted to the byte location to
sb  x4, 6(x3)                 // store byte value 0x43 to 0x2006 //al
nop                  // memory location at 0x2000 will now equal 0x87654321
```

- If there are incorrect values written to the memory locations, debug your CodAL project. Here are a few suggestions:
  - If the values written are to the incorrect memory location, validate that you are calculating the correct memory address of rs1 + stype_immediate
  - If the bytes or half-words are being written to the wrong location in the memory word, check that you are sending the correct memory size and address
  - If some of the store bytes and half-words store properly but others do not, such as repeating a value in the incorrect byte location, it could be a data-hazard / data-forwarding error. This test program test both data-hazard conditions as well as store operations
  - This test also validates whether a store immediately following a branch is canceled if the branch is taken. If there is a value in 0x200c memory location before you reach the halt instruction, your CA model is not handling this condition correctly. If a branch is taken, the memory operation in the EX stage must be forced to be an idle bus operation since the instruction in all earlier pipeline stages are to be flushed on a branch taken or jump.
- Once you have completely passed the store_load_assembly_test, you have completed **Checkpoint 1**.

## Checkpoint 2: Validating load instructions

- **Gen**/Build your **SDK(ca)** model
- For **Checkpoint 2**, you will continue to use the store_load_assembly_test
- Comment out the **halt** statements after the store instruction tests, the first halt statement on line 70

- In the Workflow, , perspective with the store_load_assembly_test project selected, using **Assign SDK** assign your **SDK(ia)** model to the software project
- **Compile** the store_load_assembly_test
- **Launch Debug** specifying your **SDK(ca)** model
- You can step through your project in the debugger or place a breakpoint and resume to line # 79 which is the start of the load test
- The load test will validate whether each load instruction performs as required
- Debug the store_load_assembly_test by stepping through the load portion of the assembly test or until it fails.  As in the store test, review the comments and validate that the changed value in the registers match.



```
REGISTERS
  Architectural Registers
    r_pc------> 0x10f8
    rf_xpr_0--> 0x0
    rf_xpr_1--> 0x0
    rf_xpr_2--> 0x87654321
    rf_xpr_3--> 0x2000
    rf_xpr_4--> 0x87654321
    rf_xpr_5--> 0x[illegible]1
    rf_xpr_6--> 0x0
```

```
74  *****************************************************************/
75      nop
76      nop
77      nop
78      nop
79      lw x4, 0(x3)                        // load the value from memory w
80      lb x5, 7(x3)                        // reassemble x2 from the bytes
81      nop
82      nop
83      lb x10, 6(x3)
84      nop                 // check that x4 == x2  (confirms word loads)
85      slli x10, x10, 8
```

- This program does test whether lb (sign extended load byte) and lbu (unsigned load byte) load the correct value into the registers, and it tests lh (signed half-word) and lhu (unsigned half-word)
- Once you have passed these load tests, you have completed **Checkpoint 2.**

## Checkpoint 3:  Validating MEM stage stall if memory bus indicates data is not ready (available)

- **Gen**/Build your **SDK(ca)** model
- For **Checkpoint 3**, you will continue to use the store_load_assembly_test
- Comment out the **halt** statements on line 70 and 120
- Place a new breakpoint on line 132
- In the Workflow, , perspective with the store_load_assembly_test project selected, using **Assign SDK** assign your **SDK(ia)** model to the software project
- **Compile** the store_load_assembly_test
- **Launch Debug** specifying your **SDK(ca)** model
- The code should now execute through all the store instructions, the basic load tests and now a sequence of load tests that should create a memory stall.  This test will validate whether the project properly indicates a stall situation, stalls the correct pipeline registers, inserts a NOP bubble, preserves the program counter in the IF stage, and uses the preserved instruction in the ID stage.

- Debug the store_load_assembly_test and run your code to the breakpoint on line 132. As in the store and load test, review the comments and validate that the changed value in the registers match.
- If your register values match the values expected in the comment field, you have passed Checkpoint 3
    - If not, debug using CodAL debug to pass Checkpoint 3

---

## Checkpoint 4: Validating proper detection and implementation of the load Structural Hazard

- For **Checkpoint 5**, you will continue to use the store_load_assembly_test
- Comment out the **halt** statements up to

```
/*************************************************************************
 * Load test to validate structural hazard detected and stall
 *************************************************************************/
```

- Your code should now execute through all load and store tests including the **Structural Hazards** tests
- Debug the store_load_assembly_test and step through your code until it fails or it reaches the end of the program. As in the store and load test, review the comments and validate that the changed value in the registers match.
- After you have passed the assembly test, you can move on to the regression test.

---

## Checkpoint 5: Running your regression test

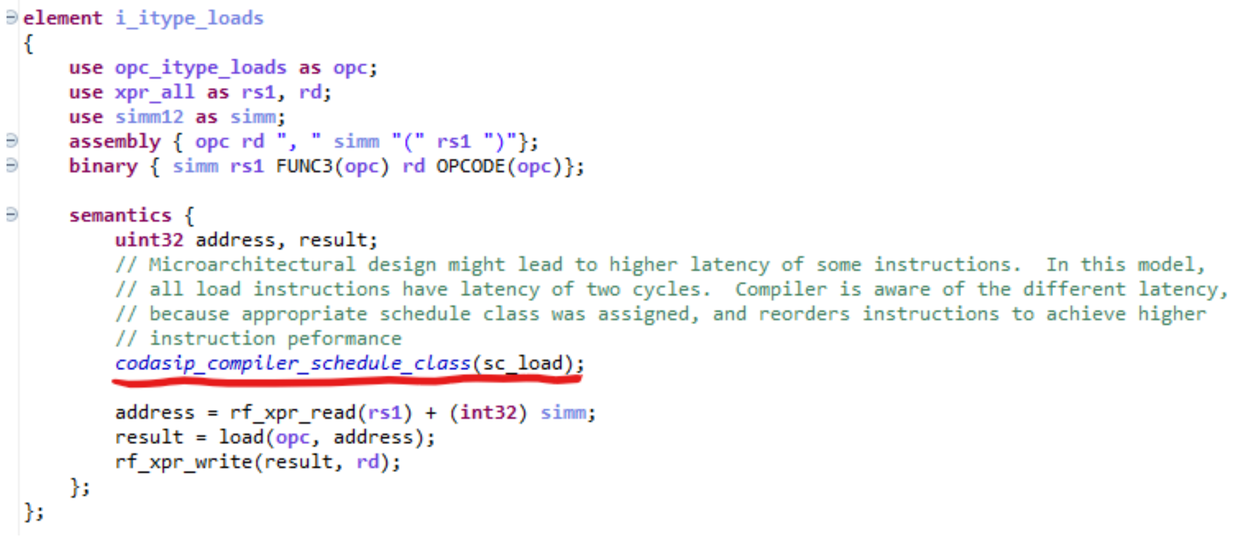
- For **Checkpoint 5**, you will use your regression test
- Comment out all the **halt** statements until the last halt instruction.
- Your code should now run all the way through the load store and upper immediate test code until the end of your regression test
- Debug your regression test and step through your code until it fails or it reaches the halt at the end of your regression test
- After your regression test passes all the way to the halt after the u-type instruction sequence, you are ready to submit your project for grading

---

- Complete the phase
    - Download the model folder
    - Submit the xml file and the tar file into Canvas for grading.

---

- OPTIONAL: You can further test your memory operation implementation by modifying the memory operation latency.
    - CodAL contains functions that are used to specify to the compiler requirements defined by the micro-architecture.
        - *codasip_compiler_schedule_class*(sc_load);
    - In the ISA model, the load operations call *codasip_compiler_schedule_class*(sc_load); to enable the architect to define its instruction latency



```
element i_itype_loads
{
    use opc_itype_loads as opc;
    use xpr_all as rs1, rd;
    use simm12 as simm;
    assembly { opc rd ", " simm "(" rs1 ")"};
    binary { simm rs1 FUNC3(opc) rd OPCODE(opc)};

    semantics {
        uint32 address, result;
        // Microarchitectural design might lead to higher latency of some instructions.  In this model,
        // all load instructions have latency of two cycles.  Compiler is aware of the different latency,
        // because appropriate schedule class was assigned, and reorders instructions to achieve higher
        // instruction peformance
        codasip_compiler_schedule_class(sc_load);

        address = rf_xpr_read(rs1) + (int32) simm;
        result = load(opc, address);
        rf_xpr_write(result, rd);
    };
};
```

    - In /model/share/resources/arch.codal, search for schedule_class sc_load and change the latency from 1 to 2
        - 1 equates the load result is immediate available after the load in a single cycle computer (default latency)
        - 2 equates that the the result is available 2 cycles after the load in a single cycle, resulting in the instruction immediate after the load cannot use the load result

Objective: In a Reduced Instruction Set Computer (RISC), the computer architecture is based on the assumption that simple instructions (register to register operations) will minimize circuitry that will minimize the CPU clock period. With CPU frequency inversely proportional to the CPU clock period, the shorter the clock period equates to higher clock frequency or performance. To get data into these registers, load operations must be performed. To get data out of these registers, a store to memory operation will be performed.

In this phase, you will add the load/store instructions into the decoder and the circuitry to perform both load and store operations. With a 5-stage pipeline architecture where the fourth stage is the MEM (memory) stage and the third stage is the EX (execute) stage, a **Structural Hazard** is created. The MEM stage load operation cannot data-forward back into the EX stage for an EX stage instruction that requires the load result. This **Structural Hazard** will need to be handled by your project. Load results are dependent on the response from memory. If the memory does not respond in the one cycle MEM stage, the processor must be stalled until the memory bus responds with the load result. A "stall" holds state of the appropriate pipeline stages to prevent them from progressing to the next stage.

Key Learning Outcomes of this phase:

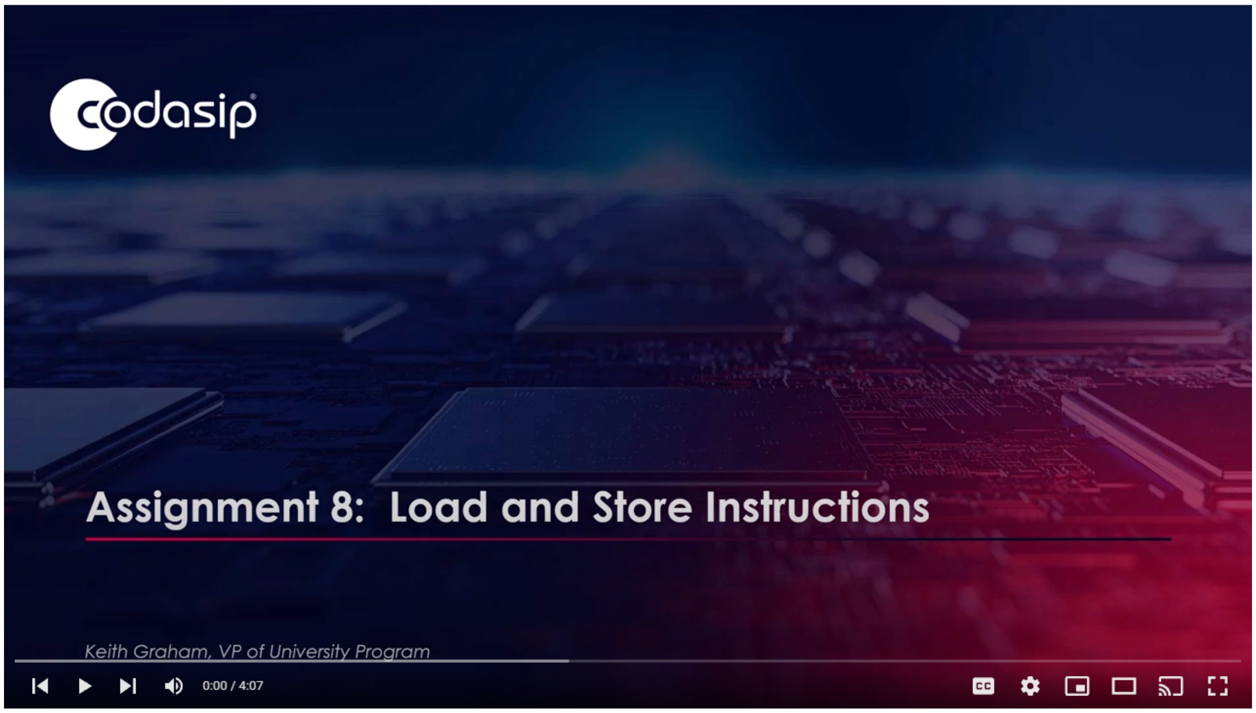
Load/Store operations: Load and Store operations transfer data from memory to the register file and back to memory. These transfers across the CPU boundary to the external memory resource occur via a memory bus. The bus requires four pieces of information to perform the memory operation:

- The operation type: Is it a load, store, or idle (no memory request) operation?
- Memory address of the operation
- Size of the operation (byte, half-word, or word)
- Data from the processor for store operations or data from the memory for loads

Most modern buses are synchronous. A synchronous bus uses a clock to hold the state of the bus to be processed on. In the Execute stage, the CPU will launch the memory operation onto the bus during the next CPU clock which will then transition the load/store from the EX stage to the MEM stage. In the Memory stage, the memory interface will complete the operation.

Load data hazard: The load data hazard is a special type of hazard, a **Structural Hazard**. The difference between a computational data hazard and a load data hazard is that the load result is not available until after the MEM stage where the computational instruction is available after the EX stage. With the load result not available after the EX stage, an EX data-forwarding is not possible, a **Structural Hazard**. The structure of the pipeline is the root cause of this hazard. The only solution to resolve this **Structural Hazard** is to stall the instruction in the ID stage that requires the memory value from the load instruction in the EX stage. As the load moves forward, a NOP bubble must be inserted between the two instructions. After the stall, the load instruction is now in the MEM stage and the data hazard condition with the ID instruction has changed to a MEM data hazard which can be solved through data forwarding. Which means the **Structural Hazard** has been removed.

**Phase 8:** Load and Store Operations



Assignment 8: Load and Store Instructions

- Store Operation, how does it work across the memory bus?
  - The RISCV curriculum project utilizes the AHB3-Lite bus to access memory. You can learn more about the AHB3-Lite bus at the following weblink
    - https://developer.arm.com/documentation/ihi0033/a/Transfers/Basic-transfers?lang=en

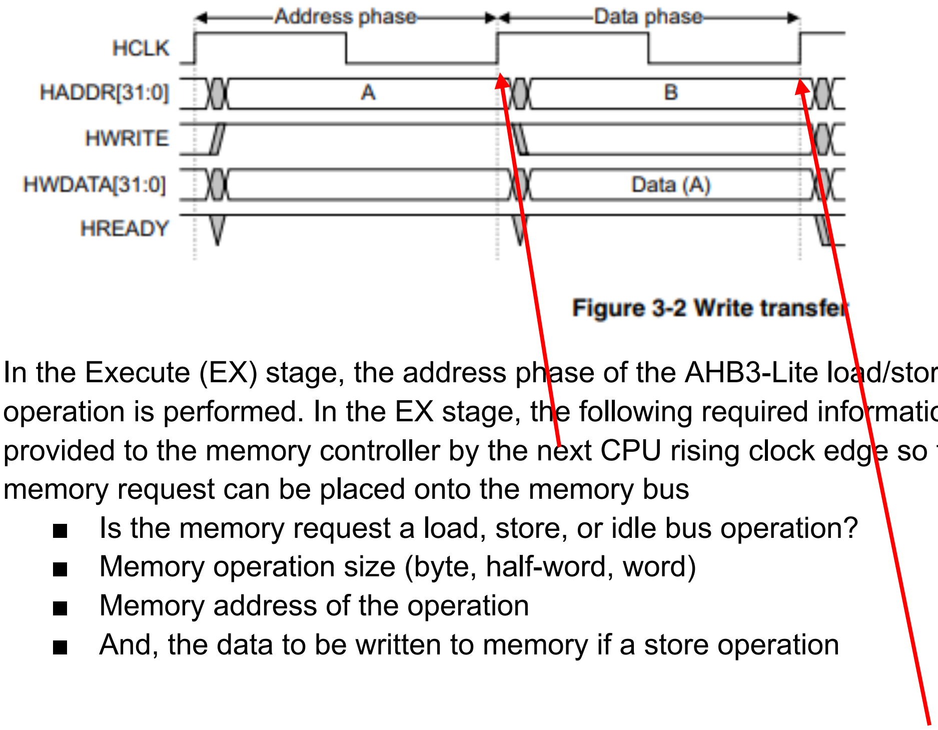
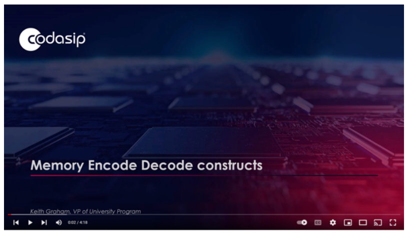


Figure 3-2 Write transfer

  - In the Execute (EX) stage, the address phase of the AHB3-Lite load/store operation is performed. In the EX stage, the following required information is provided to the memory controller by the next CPU rising clock edge so that the memory request can be placed onto the memory bus
    - Is the memory request a load, store, or idle bus operation?
    - Memory operation size (byte, half-word, word)
    - Memory address of the operation
    - And, the data to be written to memory if a store operation

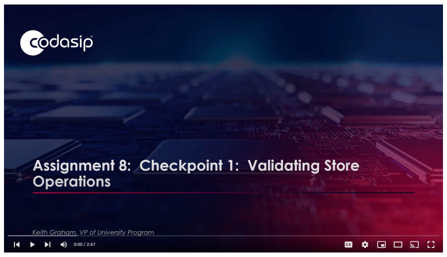
    - For writes/store operations, the processor must present the data onto the AHB3-Lite data bus in the next bus cycle, while the store operation is in the MEM stage. The data will be latched into the memory upon the rising CPU clock edge as the store operation progresses from the MEM stage to the Write Back (WB) stage.
- How does the store operation determine what to present to the AHB3-Lite at the end of the EX stage?
    - Is it load, store, or idle?
        - Determined by the instruction decoder
    - Memory operation size?
        - Determined by the instruction decoder (sw - word, sh - half-word, and sb - byte)
    - Memory address to access
        - rs1 (src1) operand plus load or store offset will be calculated through the ALU as an immediate ADD operation using the s-type immediate
- Slower memories can improve their performance by supporting a BURST mode. Dynamic Memories, DRAM, are a good example. The first line of data may take several memory bus cycles, but subsequence accesses are accessed much faster such as the next bus cycle. The DRAM can provide back to back reads after the first line of data by internally accessing multiple lines of data simultaneously and putting the data lines onto the memory bus on back to back bus cycles.
    - Instruction and data caches commonly use BURST access to fill their memories
- Processor load and store addresses are typically "random". Back to back loads or stores are commonly not addressed sequentially. The interface between the processor and cache are non-BURST mode.
- **FAQ: Memory Encode Decode constructs:** The Memory Encode and Decode constructs are functions within the CodAL processor description language which provides a higher level of abstraction that takes detail work away from the developer to improve the processor engineer's productivity while minimizing the introduction of errors (bugs).


codasip
Memory Encode Decode constructs
Keith Graham, VP of University Program

- 
- The encode construct is used to place data onto the proper location of the memory bus during a store operation. For example, if a byte is to be stored into the 2nd byte location of a memory word, the byte from the register file must be shifted from the 0-byte location in the register file onto the 2nd-byte location of the memory bus.

- 
- The decode construct performs the opposite operation for the load instruction. If a byte is read from the 1st-byte location in a memory word, it must be shifted to the 0-byte location to be written into the register file.
- You can learn more about the *data phase* of the transport function supporting the AHB3-LITE bus by searching for "AHB3_LITE Transports" in the Codasip CodAL Language Reference Manual

- **Phase 8: Checkpoint 1: Validating store operations:** There are three type of store operations that are validated through the store_load_assembly_test.s assembly program in checkpoint 1:


codasip
Assignment 8: Checkpoint 1: Validating Store Operations
Keith Graham, VP of University Program

  - store word (sw)
  - store byte (sb)
  - store half-word (sh)
  - This video reviews how the test program can be used to validate the store instructions that you have implemented for your RISCV processor.
- To add as little code as possible before **Checkpoint 2**, you will add the load operations without the load hazard detection circuitry (Best coding practice). The **Checkpoint 2** will execute load instructions that will not result in load hazard conditions.

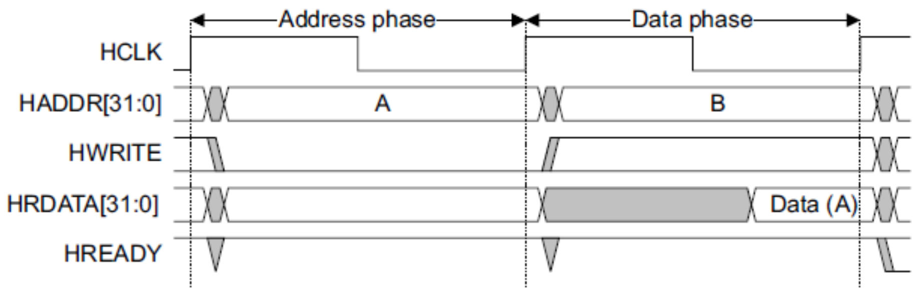


**Figure 3-1 Read transfer**

AHB3-Lite Load Memory Operation

- Earlier in this phase, you added the load instructions into the instruction decoder and in the EX stage, and you should have assigned op_rw, op_type, and s_ex_mem_size control-signals for all the mem_ops enums defined in ca_defined.codal. If you have not added the load case statements in assigning these control-signals, you should add them now.
- Both the loads and stores calculate their address through the ALU which simplifies the design. No changes are required.
- The only Cycle Accurate (CA) updates required to enable the load instructions will be in the MEM stage. Since the load data will be received from memory in this stage, it must be placed onto the result bus to be forwarded to the WB stage. In the WB stage, the load data will be written to the Register File and available for data forwarding.

- **Phase 8: Checkpoint 2: Validating Load Instructions:** There are three type of load operations that are validated through the store_load_assembly_test.s assembly program in checkpoint 2:

codasip
Assignment 8: Checkpoint 2: Validating Load Operations
Keith Graham, VP of University Program

- 
- load word (lw)
- load byte (lb, lbu)
- load half-word (lh, lhu)
- 
- This video reviews how the test program can be used to validate these load instructions that you have implemented for your RISCV processor.

- 
- In addition to the actual load instructions, this assembly test will also validate whether the signed operations of load byte, lb, and load half-word, lh, are implemented correctly.
- The AHB3-Lite bus can perform back to back store operations or back to back load operations. If a load operation immediately follows a store operation, the AHB3-Lite memory bus will inform the processor that data is not available by bringing HREADY signal low, to 0.

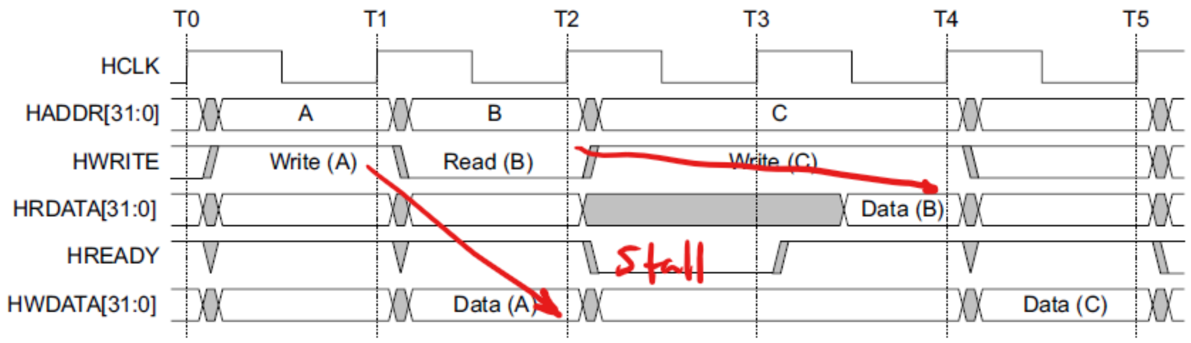
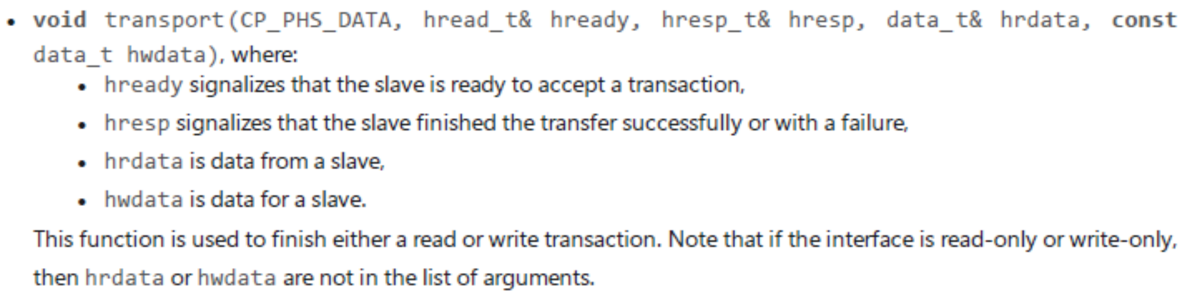


**Figure 3-5 Multiple transfers**

- The HREADY signal informs the stall is made available through the data phase transport function call

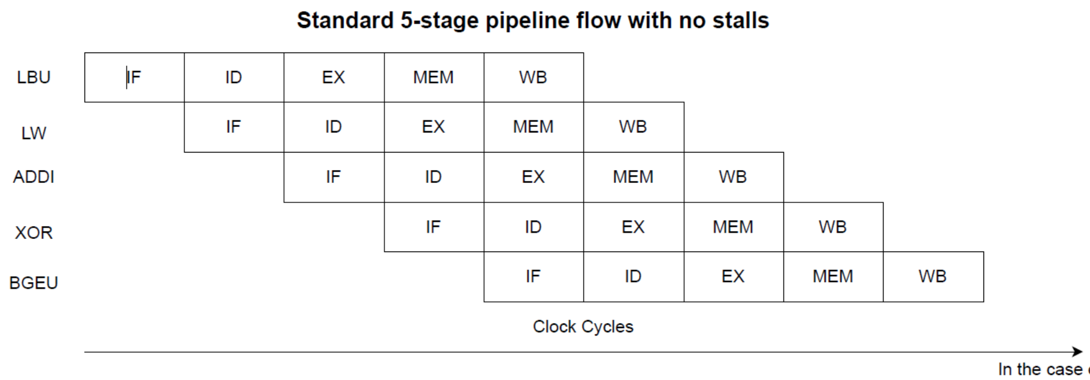


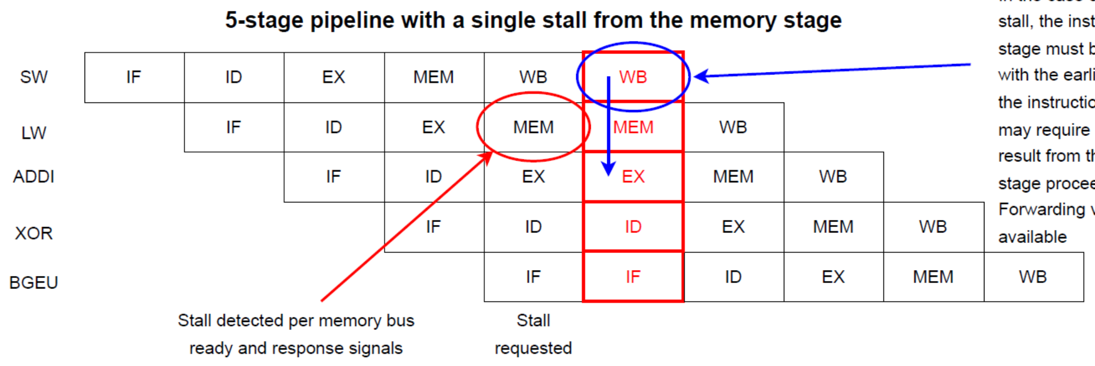
- `void transport(CP_PHS_DATA, hread_t& hready, hresp_t& hresp, data_t& hrdata, const data_t hwdata)`, where:
  - `hready` signalizes that the slave is ready to accept a transaction,
  - `hresp` signalizes that the slave finished the transfer successfully or with a failure,
  - `hrdata` is data from a slave,
  - `hwdata` is data for a slave.

  This function is used to finish either a read or write transaction. Note that if the interface is read-only or write-only, then `hrdata` or `hwdata` are not in the list of arguments.

- Upon a memory stall, instructions in later stages such as the WB stage must be stalled as well due to an instruction in the EX stage that may have been relying on a result value in the WB stage as a Data Forwarding operand. All earlier stages including the MEM stage must stall.
  - All stages must stall which can be done in the ca_pipeline_control.codal file

        - The PC register can be considered the left-side or entry pipeline register of the Instruction Fetch (IF) stage. This stall will need to occur in the ca_pipe1_if.codal file
    - The instruction read from memory in the ID stage must be preserved for the stalled pipeline cycle since the IF stage has already given the memory bus a new address to access
        - The saving of the instruction read from memory in the ID stage will need to be implemented in the ca_pipe2_id.codal file




5-stage pipeline with a single stall from the memory stage
SW
LW
ADDI
XOR
BGEU
IF
ID
EX
MEM
WB
Stall detected per memory bus ready and response signals
Stall requested

In the case of a MEM stage stall, the instruction in the WB stage must be stalled along with the earlier stages due to the instruction in the EX stage may require a Data Forwarding result from the WB stage. If WB stage proceeded, the Data Forwarding value would not be available

-

## Appendix A: YouTube videos for Phase 8

Phase Videos:

- **Phase 8: Load and Store Instructions**

  The RISCV instruction set architecture is based on the concept of a load-store processor architecture. In a load-store system, there are specific instructions that move data from memory to the register file and from the register file to memory while another set of instructions operate on the data only from the register file. With the register file tightly coupled with the processor, it minimizes access (delay) to the data which results in optimal processor frequency or performance.

  In this phase, you will learn how to implement load and store operations in a 5-stage RISCV pipeline as well as overcome a new type of hazard, a structural hazard. A

structural hazard in one that results in a pipeline stage not able to complete its operation due to a required resource is not available.
    - https://www.youtube.com/watch?v=lTvrJPL8vZY&list=PLTUn6Ox9e6q2ienoql3KClFtRPMqO28uj&index=16
- **Phase 8: Checkpoint 1: Validating Store Operations**
    - There are three type of store operations that are validated through the store_load_assembly_test.s assembly program in checkpoint 1:

      store word (sw)
      store byte (sb)
      store half-word (sh)

      This video reviews how the test program can be used to validate the store instructions that you have implemented for your RISCV processor.
    - https://www.youtube.com/watch?v=zeFB2cdMBZs&list=PLTUn6Ox9e6q2ienoql3KClFtRPMqO28uj&index=17&t=1s
- **Phase 8: Checkpoint 2: Validating Load Operations**
    - There are three type of load operations that are validated through the store_load_assembly_test.s assembly program in checkpoint 2:

      load word (lw)
      load byte (lb, lbu)
      load half-word (lh, lhu)

      This video reviews how the test program can be used to validate these load instructions that you have implemented for your RISCV processor.

      In addition to the actual load instructions, this assembly test will also validate whether the signed operations of load byte, lb, and load half-word, lh, are implemented correctly.
    - https://www.youtube.com/watch?v=G-l2sfiSwPc&list=PLTUn6Ox9e6q2ienoql3KClFtRPMqO28uj&index=18

## 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.
- **FAQ: Memory Encode Decode constructs**
    - The Memory Encode and Decode constructs are functions within the CodAL processor description language which provides a higher level of abstraction that takes detail work away from the developer to improve the processor engineer's productivity while minimizing the introduction of errors (bugs).

      The encode construct is used to place data onto the proper location of the memory bus during a store operation. For example, if a byte is to be stored into the 2nd byte location of a memory word, the byte from the register file must be shifted from the 0-byte location in the register file onto the 2nd-byte location of the memory bus.

The decode construct performs the opposite operation for the load instruction.  If a byte is read from the 1st-byte location in a memory word, it must be shifted to the 0-byte location to be written into the register file.

- https://www.youtube.com/watch?v=_2s2XP6cStQ&list=PLTUn6Ox9e6q1ii0fp-N_GDPZjAtkDZmqe&index=16