Computer Organization
5-stage RISCV32I Processor
Phase 3: Regression Testing
Fall 2022

Objective: Proper planning can reduce the overall development and debug effort. Through writing the test to verify the functionality of each instruction type, you will gain an understanding of each of the processor's instructions and its assembly language format. By properly sequencing the instructions, each set of instructions will validate a limited portion of your 5-stage RISCV design. The goal here is to add as little new functionality to your design as possible and then verify it. By limiting the amount of new functionality to validate, if the test fails, you can begin to focus your debug efforts on the newly added segment. As you add new functionality, the earlier portions of your test will validate that the newly added functionality has not "broken" your earlier developed functions, regression testing.

In this phase, you will write an assembly program where each section of your program will test a new RISCV instruction type or functionality. These tests will be ordered to match the development of your 5-stage pipeline phases. You will use the Instruction Accurate (IA) model that you have been using to validate this assembly program. However, in the future phases you will be developing a CA model. This will be used to validate future phases instead of the IA model.

- Phase 5: Data-path (i and r-type instructions)
- Phase 6: Data Hazard Detection and Forwarding (improving the pipeline)
- Phase 7: Branch and Jump Instructions (b and j-type instructions)
- Phase 8: Load and Store Instructions (l and s-type instructions)
- Phase 9: C-Compiler Generation (u-type instructions)

Key Learning Outcomes of this phase:
Regression Test(ing): Fundamental processes in developing software or hardware are:

- Discover errors as soon as possible
- Retest previously working functionality

The first principle, "Discovering errors as soon as possible," focuses debug effort on the recent changes or additions since the last time the project had passed its testing. By limiting the scope of the debug effort, the developer can locate the error much faster by narrowing the search to a limited amount of code.

The second principle, "Retest previously working functionality," validates whether new functionality negatively impacts previously working functions at development and not at system validation. Similar to the first principle, learning early that new code affects previous functions narrows your debug scope and provides a higher level of confidence that a completed project
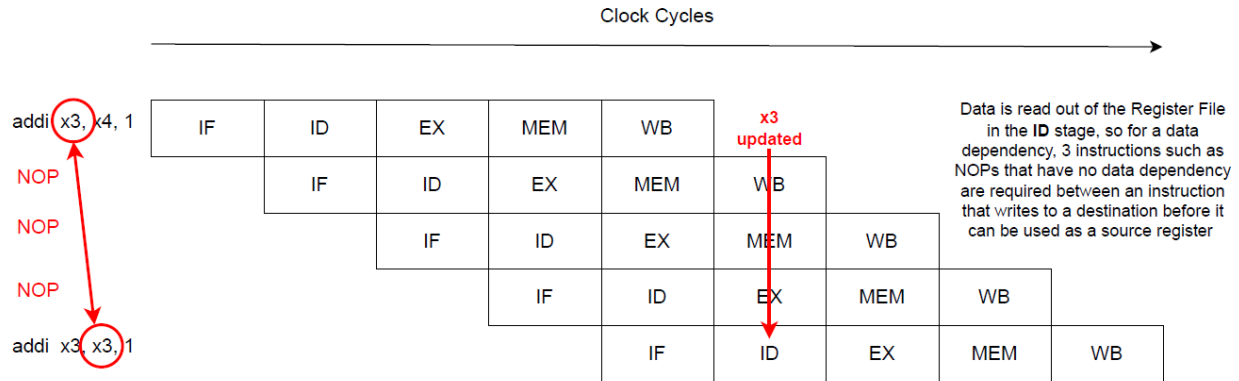
will pass system verification. This is because the regression test will have passed completely before moving to system validation.

- This is a 2-week phase
- Due date: Monday October 3rd 2022

Instructions:
- The two projects required for this phase are:
  - ia_riscv32i
  - regression
- You will be using an Instruction Accurate (IA) model which has errors built into the processor to help you identify test coverage of one error per Instruction Type. You will be able to disable the introduced error through the processor's ip.conf file. Import the following phase3_ia_riscv32i processor model
  - git clone https://github.com/tamisil/phase3_ia_riscv32i.git
- Import the regression project using the following git clone command
  - git clone https://github.com/tamisil/regression.git

- To complete an operation, an instruction utilizes the processor's resources in a sequence. Performance increase can be realized by pipelining these resources. Pipelining is achieved by allowing the following instruction to utilize a processor resource once the current instruction no longer requires it. To separate the stages, these resources are separated by a series of pipeline registers to hold the state of the instruction being operated. In the 5-stage pipeline design for this course, the processor resources are divided as follows:
  - Instruction Fetch (IF): Sending the address to fetch the next instruction
  - Instruction Decode (ID): Set the instruction's control-signals, data-paths, opcodes, and register file operands
  - Execute (EX): Perform an instruction's arithmetic operations
  - Memory (MEM): Complete a data load and store operations
  - Write Back (WB): Store the instruction's result into the CPU's register file
- With 5-stages, five instructions can be processed at one-time. If an instruction in the Execute stage requires data from the instruction in the Write Back stage, it cannot access the updated register file value since it will have read its operands in its previous stage, Instruction Decode (ID).

Clock Cycles

addi x3, x4, 1	IF	ID	EX	MEM	WB	x3 updated	
NOP		IF	ID	EX	MEM	WB	
NOP			IF	ID	EX	MEM	WB
NOP				IF	ID	EX	MEM	WB
addi x3, x3, 1					IF	ID	EX	MEM	WB

Data is read out of the Register File in the **ID** stage, so for a data dependency, 3 instructions such as NOPs that have no data dependency are required between an instruction that writes to a destination before it can be used as a source register

Standard 5-stage pipeline flow with no stalls

- ○ In the above pipeline diagram, three NOP or other instructions are required between an instruction that updates a register file location, upon completion of the Write Back stage, and the next instruction that requires it
- ○ This requirement of the intervening instructions can be eliminated through data hazard detection and the technique of data-forwarding. You will be implementing this technique in Phase 6. For Phase 5, you will need to develop the regression test with intervening NOP or other instructions for the i-type and r-type instructions.
- The first set of instructions to test will be the i-type immediate. These are the easiest instructions to implement in the processor and will be the first that you will develop in the Cycle Accurate (CA) model.
- Appendix A contains the RISCV32i assembly instructions in your ia_riscv32i project
- For additional information on the RISCV32I instructions and their immediate values, reference the RISC-V Instruction Set Manual (Unprivileged ISA)
  - ○ https://riscv.org/technical/specifications/
  - ○ Reference chapter 2: RV32I Base Integer Instruction Set
- Go to this phase's imported regression project and open regression.s in /src folder
  - ○ To begin this project, update the author and date for this assembly program at the top of regression.s

○ You will see that this phase has been started to show you an example. Each instruction that requires a result of a prior instruction requires three intervening NOP or other instructions

```
/*
   Immediate (i-type) ALU operations
*/
   addi x2, x0, 2              // load 2 into register x2
   nop
   nop
   nop
   addi x2, x2, (-1 & 0xfff)   // add -1 to x2
   nop                 // x2 = 2
   nop
   nop
   slti x3, x2, 2              // compare 1 to 2, less than so x3 = 1
   nop                 // x2 = 1
   nop
   nop
   slti x3, x2, 1             // compare 1 to 1, not less than, so x3 = 0
   nop                 // x3 = 1
   nop
   nop
```

○ Complete the i-type instruction test for the remaining i-type instructions
- addi
- slti
- sltiu
- xori
- ori
- andi

○ Additional information on these instructions can be found in **Appendix A: ia_riscv32i RISCV32i supported assembly instructions**
○ Each assembly instruction requires a comment statement
○ You will need to comment to indicate the register file has been updated when you simulate using the **CA model**, five instructions after the assembly instruction, for 5-stage pipeline design. From the above example, the comments indicating the register file has been updated are not as tabbed over as the assembly language comments.

```
/*
   Immediate (i-type) ALU operations
*/
   addi x2, x0, 2              // load 2 into register x2
   nop
   nop
   nop
   addi x2, x2, (-1 & 0xfff)   // add -1 to x2
   nop                 // x2 = 2
   nop
   nop
   slti x3, x2, 2              // compare 1 to 2, less than so x3 = 1
   nop                 // x2 = 1
   nop
   nop
   slti x3, x2, 1             // compare 1 to 1, not less than, so x3 = 0
   nop                 // x3 = 1
   nop
   nop
```
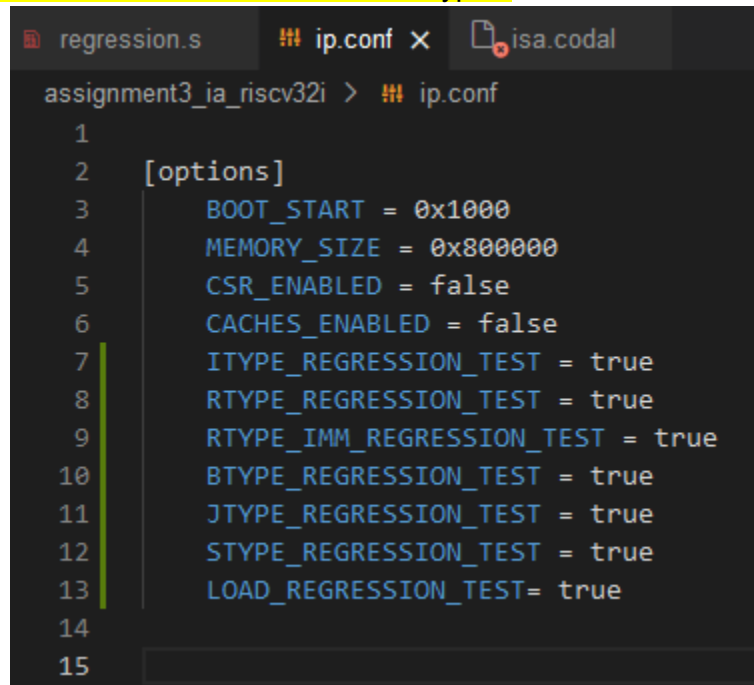
- ○ At the end of your i-type instruction test sequence, ensure to keep four NOPs before and after the halt instruction
  - ○ **Test Escapes** are situations that can occur, and if not tested, a failure may occur in the system. Failures are more easily debugged with properly developed test code because the tests set up a particular condition to test, and if it fails, you know what is being tested and thus, you can focus your debug effort effectively reducing your overall development and debug time.
    - ■ Example: addi x2, x0, -1
      - ● In RISCV, i-type immediate are 12-bits wide and are signed extended to 32-bits
      - ● -1 would equate to the sign immediate value of 0xfff (12-bits)
      - ● The above example instruction would add -1 to 0 (x0 is defined to be 0) which would store the result of -1 into register x2
      - ● -1 32-bits extended equates to 0xffffffff
      - ● If the result stored in x2 = 0xfff and not 0xffffffff, you would know that your sign extension of the i-immediate field did not work and you can focus your debug efforts on the code dedicated to sign extension
  - ○ Your test sequences should test all possible combinations to eliminate a possible **Test Escape.**
  - ○ The grading rubric for this phase includes test coverage and a **test escape** will negatively impact your grade
  - ○ Before you begin your first Checkpoint 1 phase below, go to the **File Explorer,** ⌘, perspective and open ip.conf for phase3_ia_riscv32i
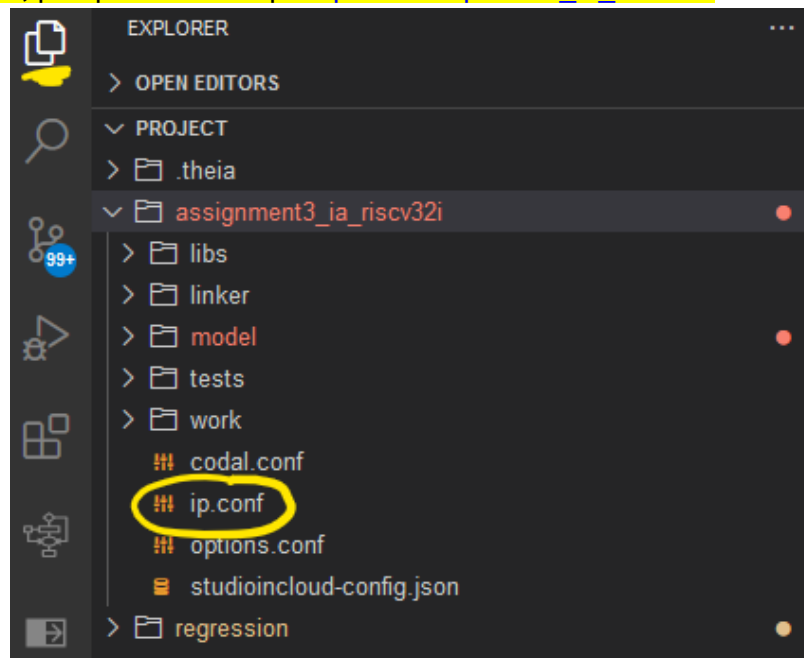
  - ○ You will see variables or constants that can be defined in this file that combined with compiler directives will modify the processor's project compile/build. The

○ If the variable/constant is "**true**," the error is introduced into the processor project.
You will start this phase with all the errors introduced into the processor model.
As you complete the test coverage for an instruction type and identify the error
through debugging your regression test, you can disable the test for future debug
runs

Checkpoint 1: Validating your i-type instruction test sequence

● As you did in phase 1 and 2, you will first need to **generate,** ⬡ Gen SDK , and **assign,**
▽ Assign SDK , the ia model for the regression project to your phase3_ia_riscv32i model

● In the Compiler Configuration, ⚙, for the assembly program, you must select "No
Startup or Default Lib," and then click on bottom left button, Save

Compiler Configuration

Choose Optimization Level

No Optmize(-O0)

Choose Debug Level

Extra Information

Config

☑ No Startup or Default Lib (-nostdlib)
☐ Do not use standard start files (-nostartfiles)
☐ Do not use default libraries (-nodefaultlibs)
☐ Omit all symbol information (-s)
☐ No Shared Libraries (-static)
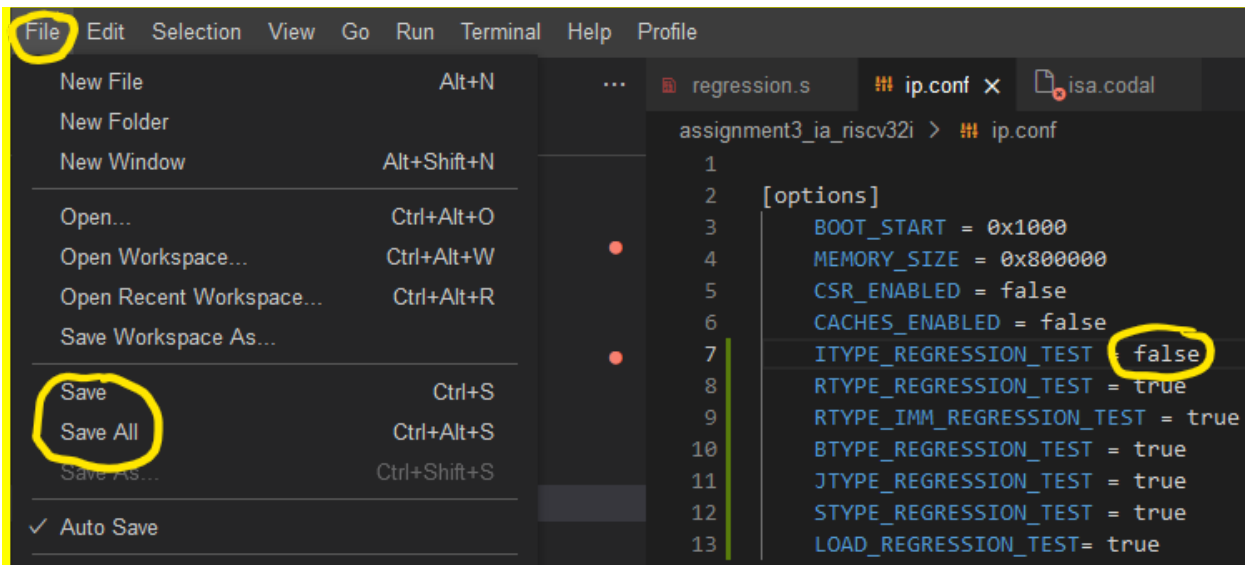Extra Flags [_____]
Extra Linker Flags [_____]

Save

- Compile your regression project
- Add a breakpoint on the first and second instruction of the regression assembly program
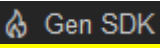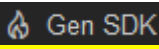
```
12      _start:
13          .global _start
14
15      /*
16          Immediate (i-type) ALU operations
17      */
● 18         addi x2, x0, 2               // load 2 into register x2
● 19         nop
20          nop
```

- Debug the regression test by stepping through the code.
- NOTE: For the IA model, the register value will change as the instruction moves to the next instruction, not after 5 instructions. In the IA model, it emulates a single-stage processor.
  - Use the "Registers" view to see the Register File values and confirm that you get the answers that are expected
- If you get a get a failure that you are confident is not a programming error, open ip.conf and disable, make **false**, the ITYPE_REGRESSION_TEST variable/constant to disable or take the error out of the processor model for the ITYPE instructions
    - Once you make this change, you will need to save the file through the top menu bar

```
File  Edit  Selection  View  Go  Run  Terminal  Help  Profile

New File                    Alt+N          ...   ≣ regression.s   ▦ ip.conf ×   ▱ isa.codal
New Folder
New Window              Alt+Shift+N              assignment3_ia_riscv32i  >  ▦ ip.conf
                                             1
Open...                   Ctrl+Alt+O        2     [options]
Open Workspace...         Ctrl+Alt+W        3         BOOT_START = 0x1000
Open Recent Workspace...  Ctrl+Alt+R        4         MEMORY_SIZE = 0x800000
Save Workspace As...                        5         CSR_ENABLED = false
                                            6         CACHES_ENABLED = false
Save                      Ctrl+S            7         ITYPE_REGRESSION_TEST = false
Save All                  Ctrl+Alt+S        8         RTYPE_REGRESSION_TEST = true
Save As...                Ctrl+Shift+S      9         RTYPE_IMM_REGRESSION_TEST = true
                                            10        BTYPE_REGRESSION_TEST = true
✓ Auto Save                                 11        JTYPE_REGRESSION_TEST = true
                                            12        STYPE_REGRESSION_TEST = true
                                            13        LOAD_REGRESSION_TEST= true
```

- ■ After you have saved the new configuration, you will need to **re-generate** SDK(ia), 🔥 Gen SDK , to re-compile/build the processor project. No need to re-assign this SDK(ia) model to your regression test. You are just rebuilding the model.
- ■ Rerun the debugger and validate that your code now passes the suspected failure.
   - ● If you are now passing, you have verified that your regression test has "test coverage" for this particular failure mode
   - ● If you are still failing, you will need to debug your assembly test and go back into ip.conf to re-enable this ITYPE_REGRESSION_TEST and **re-generate** the processor model, 🔥 Gen SDK
- ● Passing the introduced failure does not mean that you have complete test coverage of the ITYPE set of instructions. Please complete the test for all of the ITYPE instructions and their possible test conditions.
- ● Once you reach the halt instruction after your i-type test sequence successfully with getting the results in the register file as planned, you have completed **Checkpoint 1**

- ● With the i-type test completed, after the r-type comment field, complete a test sequence for all the r-type of instructions. Similar to the i-type, there must be a minimum of three NOP or intervening instructions

```
/*
    Immediate (r-type) ALU operations
*/
```

- ● The r-type of instructions to validate are:
   - ○ add
   - ○ sub
   - ○ sll
   - ○ slt

- ○ sltu
- ○ xor
- ○ srl
- ○ sra
- ○ or
- ○ and
- Additional information on these instructions can be found in **Appendix A:** phase3_ia_riscv32i RISCV32i supported assembly instructions
- Each instruction that requires a result of a prior instruction requires three intervening NOP or other instructions since data hazard detection and forwarding will not have been implemented
- You will need to comment to indicate the register file has been updated when you simulate using the **CA model**, five instructions after the assembly instruction, for 5-stage pipeline design. From the above example, the comments indicating the register file has been updated are not as tabbed over as the assembly language comments.
- At the end of your r-type instruction test sequence, ensure to keep four NOPs before and after the halt instruction
- The grading rubric for this phase includes test coverage and a **test escape** will negatively impact your grade
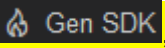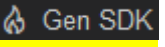
Checkpoint 2: Validating your r-type instruction test sequence

- Comment out the halt instruction at the end of your i-type test sequence so that as you step through your regression test, you will now step into your r-type test sequence after your i-type test
- Debug the regression test and step through the code. For the IA model, the register value will change as the instruction moves to the next instruction, not after 5 instructions. In the IA model, it emulates a single-stage processor.
  - Use the "Registers" view to see the Register File values and confirm that you get the answers that are expected
- If you get a get a failure that you are confident is not a programming error, open ip.conf and disable, make **false**, the RTYPE_REGRESSION_TEST variable/constant to disable or take the error out of the processor model for the RTYPE instructions
    - Once you make this change, you will need to save the file through the top menu bar

```
 regression.s        ttt ip.conf  X        isa.codal

assignment3_ia_riscv32i  >  ttt ip.conf
    1
    2     [options]
    3         BOOT_START = 0x1000
    4         MEMORY_SIZE = 0x800000
    5         CSR_ENABLED = false
    6         CACHES_ENABLED = false
    7         ITYPE_REGRESSION_TEST = false
    8         RTYPE_REGRESSION_TEST = false
    9         RTYPE_IMM_REGRESSION_TEST = true
   10         BTYPE_REGRESSION_TEST = true
   11         JTYPE_REGRESSION_TEST = true
   12         STYPE_REGRESSION_TEST = true
   13         LOAD_REGRESSION_TEST= true
   14
```

- ■ After you have saved the new configuration, you will need to **re-generate SDK(ia),** Gen SDK , to re-compile the project.
- ■ Rerun the debugger and validate that your code now passes the suspected failure.
  - ● If you are now passing, you have verified that your regression test has "test coverage" for this particular failure mode
  - ● If you are still failing, you will need to debug your assembly test and go back into ip.conf to re-enable this RTYPE_REGRESSION_TEST and **re-generate** the processor model, Gen SDK
- ● Passing the introduced failure does not mean that you have complete test coverage of the RTYPE set of instructions. Please complete the test for all of the RTYPE instructions and their possible test conditions.
- ● Once you reach the halt instruction after your r-type test sequence successfully, getting the results in the register file as planned, you have completed **Checkpoint 2**

- ● With the r-type test completed, after the immediate (r-type) field, complete a test sequence for all the immediate (r-type) of instructions. Similar to the i-type, there must be a minimum of three NOP or intervening instructions
- ● The immediate (r-type) of instructions to validate are:
  - ○ slli
  - ○ srli
  - ○ srai
- ● Additional information on these instructions can be found in **Appendix A: phase3_ia_riscv32i RISCV32i supported assembly instructions**

- Each instruction that requires a result of a prior instruction requires three intervening NOP or other instructions since data hazard detection and forwarding will not have been implemented
- You will need to comment to indicate the register file has been updated when you simulate using the **CA model**, five instructions after the assembly instruction, for 5-stage pipeline design. From the above example, the comments indicating the register file has been updated are not as tabbed over as the assembly language comments.
- At the end of your immediate (r-type) instruction test sequence, ensure to keep four NOPs before and after the halt instruction
- The grading rubric for this phase includes test coverage and a **test escape** will negatively impact your grade

---

Checkpoint 3: Validating your immediate (r-type) instruction test sequence

- Comment out the halt instruction at the end of your r-type test sequence so that as you step through your regression test, you will now step into your immediate (r-type) test sequence after your i-type and r-type test sequences
- Debug the regression test and step through the code. For the IA model, the register value will change as the instruction moves to the next instruction, not after 5 instructions. In the IA model, it emulates a single-stage processor.
  - Use the "Registers" view to see the Register File values and confirm that you get the answers that are expected
- If you get a get a failure that you are confident is not a programming error, open ip.conf and disable, make **false**, the RTYPE_IMM_REGRESSION_TEST variable/constant to disable or take the error out of the processor model for the RTYPE_IMM instructions
    - Once you make this change, you will need to save the file through the top menu bar

```
assignment3_ia_riscv32i  >  ᛁᛁᛁ  ip.conf
  1
  2     [options]
  3         BOOT_START = 0x1000
  4         MEMORY_SIZE = 0x800000
  5         CSR_ENABLED = false
  6         CACHES_ENABLED = false
  7         ITYPE_REGRESSION_TEST = false
  8         RTYPE_REGRESSION_TEST = false
  9         RTYPE_IMM_REGRESSION_TEST = false
 10         BTYPE_REGRESSION_TEST = true
 11         JTYPE_REGRESSION_TEST = true
 12         STYPE_REGRESSION_TEST = true
 13         LOAD_REGRESSION_TEST= true
```

    - After you have saved the new configuration, you will need to **re-generate** SDK(ia), [Gen SDK], to re-compile the project.

- ■ Rerun the debugger and validate that your code now passes the suspected failure.
  - ● If you are now passing, you have verified that your regression test has "test coverage" for this particular failure mode
  - ● If you are still failing, you will need to debug your assembly test and go back into ip.conf to re-enable this RTYPE_IMM_REGRESSION_TEST and **re-generate** the processor model, 🔥 Gen SDK
- ● Passing the introduced failure does not mean that you have complete test coverage of the RTYPE_IMM set of instructions. Please complete the test for all of the RTYPE_IMM instructions and their possible test conditions.
- ● Once you reach the halt instruction after your immediate (r-type) test sequence successfully, getting the results in the register file as planned, you have completed **Checkpoint 3. These three tests will be used for Phase 5.**

- ● Phase 6, Data Hazard Detection and Forwarding, will add new data-paths that will enable results of an instruction to be fed back into a later instruction before its result is written into the register file. The next test sequence will be used to validate your Data Hazard Detection and Forwarding logic once implemented in Phase 6
  - ○ Instead of testing new instructions, this test sequence will test all possible data hazard detections and forwarding
- ● The imported regression project has initialized four registers with different values that you can use for this portion of your regression test

```
/*
    Data hazard detection and forwarding test sequences
*/
    addi x2, x0, 1          // load x2 register with 1
    addi x3, x0, 2          // load x3 register with 2
    addi x4, x0, 3          // load x4 register with 3
    addi x5, x0, -1         // load x5 register with -1
    nop
```

- ● In your five stage pipeline, there are up to three instructions that can generate a result that may be required and will not update the Register File in time for an instruction to use this updated value.
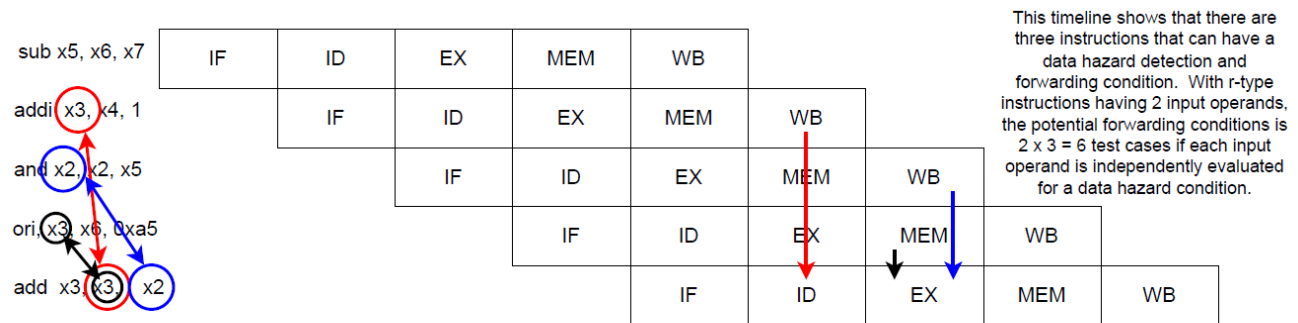
5-stage pipeline flow showing possible Data Hazard Detection and Forwarding

- The compiler could be told not to schedule an instruction that requires a result for a minimum of four instructions which would eliminate this data hazard, but this software solution has two main drawbacks:
    - If the compiler cannot find useful instructions to be placed between the instruction that results in a data hazard condition, the compiler will insert NOP operations, which perform no work which negatively impacts the performance of the processor, between the instructions until the data hazard condition has been resolved.
    - If the compiled code was used in a 6-stage processor that required instructions to be five instructions apart, the "binary" code compiled from your five-stage project could not be run on this alternative processor due to the compiled four instruction separation would not resolve the data hazard condition on a processor that required five. The code generated for your design would not be portable or reusable for this alternative processor implementation.
- A solution to solve both of these negative implications of using the compiler to solve the computer is data-forwarding that you will learn in Phase 6, Data Hazard Detection and Forwarding
- Using r-type of instructions, create a test sequence that tests all possible data hazards and forwarding conditions. The test sequence should be testing out each source operands, src1 and src2
    - How many test conditions are there?
- There are three main types of **Hazards** in Computer Architecture per the textbook "Computer Organization and Design: The Hardware / Software Interface, RISC-V Edition" by David Patterson and John Hennessy::
    - **Data Hazards:** "Also called a **pipeline data hazard**. When a planned instruction cannot execute in the proper clock cycle because data that are needed to execute the instruction are not yet available."
        - These hazards occur when any instruction that requires the result of an instruction further in the pipeline that has yet to update the **Register File**
        - **Data Hazards** in some circumstances can be resolved through **Data Forwarding**. You will be implementing **Data Forwarding** in phase 6

- **Control Hazards:** "Also known as jump or branch hazards. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected."
  - These hazards occur when a branch or jump operation changes the program flow and there are instructions already in the pipeline that would not be executed if the processor completed every instruction in a single clock cycle.
  - The impact of **Control Hazards** can be reduced through **Branch Prediction** which is a technique of branching earlier in the pipeline assuming that the branch or jump will occur. If the prediction is correct, the number of pipeline cycles that must be cleared (canceled) can be reduced. If the prediction is incorrect, the processor must recover as if the prediction had not occured.
- **Structural Hazards:** "When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute."
  - These hazards occur when two or more instructions simultaneously require a resource or a resource is not available to respond. The first example is the case that both instruction and data memory access the same memory. In this case, only an instruction fetch or a data access can be made in a single clock cycle, not both. An example of the second case is that a memory bus may need a clock cycle to turn "around." If the memory bus is writing data to the memory, a store operation, it may take a clock cycle to turn around to perform a load operation. If a load operation attempts to read memory during this required clock cycle to turn the bus around, the load operation will experience a **structural hazard**.
  - **Structural hazards** can not be solved by implementing a new data path. **Structural Hazards** can be resolved with significant architectural changes such as separate instruction and data memories or data caches that can support back to back store and load operations.
- You will need to comment to indicate the register file has been updated when you simulate using the **CA model**, five instructions after the assembly instruction, for 5-stage pipeline design. From the above example, the comments indicating the register file has been updated are not as tabbed over as the assembly language comments.
- At the end of your data hazard and data forwarding test sequence, ensure to keep four NOPs before and after the halt instruction
- The grading rubric for this phase includes test coverage and a **test escape** will negatively impact your grade

Checkpoint 4: Validating your data hazard / forward instruction sequence

- Comment out the halt instruction at the end of your immediate r-type test sequence so that as you step through your regression test, you will now step into your data hazard detection and forwarding test sequence
- Debug the regression test and step through the code. For the IA model, the register value will change as the instruction moves to the next instruction, not after 5 instructions. In the IA model, it emulates a single-stage processor.
  - Use the "Registers" view to see the Register File values and confirm that you get the answers that are expected
- There is no introduced error for this set of regression tests
- Once you reach the halt instruction after data hazard detection and forwarding test sequence successfully, getting the results in the register file as planned, you have completed **Checkpoint 4.** This test sequence will be used in Phase 6.

- Phase 7, Branch and Jump Instructions, will add new control-signals that will enable a program to change its program flow. The next test sequence will be used to validate your Branch and Jump Instructions once implemented in Phase 7.
- The next test sequence will be used to test the Branch Instructions
- The branch (b-type) instructions to be included in this test sequence are:
  - beq
  - bne
  - blt
  - bge
  - bltu
  - bgeu
- Additional information on these instructions can be found in **Appendix A:** phase3_ia_riscv32i RISCV32i supported assembly instructions
- How many test cases are there for the beq instruction?
  - It is not 2, but a minimum of 3
  - You want to test >, <, and = for complete coverage
- With branch instructions now implemented, to help you evaluate whether a branch fails, you can have any test condition that should not pass to the label **BRANCH_FAIL**. If your code ever reaches this halt statement, a failure has occurred and it is time to debug your branch circuit or condition.
  - If your code reaches the beq x0, x0, PASS instruction, it will branch to PASS the "failed" halt statement since comparing a register to itself will always be equal

```
89        nop
90        beq x0, x0, PASS
91  ∨ BRANCH_FAIL:
92        nop
93        nop
94        nop
95        nop
96        halt          // Branch test has failed, time to debug
97        nop
98        nop
99        nop
100       nop
101 ∨ PASS:
102       nop
103       nop
104       nop
105       halt
106       nop
```

- All b-type immediate values (jump offsets) are sign extended. A negative branch offset from the current Program Counter (PC) enables a portion of the code to be repeated to implement a c-program for or while loop

```
86      NEG_OFFSET:
87          nop
88          nop
89          bge x0, x2, NEG_OFFSET
90          nop
91          nop
92          bge x2, x4, POS_OFFSET
93          nop
94          nop
95      POS_OFFSET:
96          nop
97          nop
```
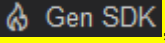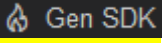
16

- You need to comment to indicate in the comment field whether the particular branch instruction is expected to be taken or not
- If you get a get a failure that you are confident is not a programming error, open ip.conf and disable, make **false**, the BTYPE_REGRESSION_TEST variable/constant to disable or take the error out of the processor model for the BTYPE instructions
    - Once you make this change, you will need to save the file through the top menu bar

```
assignment3_ia_riscv32i  >  ☰ ip.conf
  1
  2    [options]
  3        BOOT_START = 0x1000
  4        MEMORY_SIZE = 0x800000
  5        CSR_ENABLED = false
  6        CACHES_ENABLED = false
  7        ITYPE_REGRESSION_TEST = false
  8        RTYPE_REGRESSION_TEST = false
  9        RTYPE_IMM_REGRESSION_TEST = false
 10        BTYPE_REGRESSION_TEST = false
 11        JTYPE_REGRESSION_TEST = true
 12        STYPE_REGRESSION_TEST = true
 13        LOAD_REGRESSION_TEST= true
```

    - After you have saved the new configuration, you will need to **re-generate** SDK(ia), ♨ Gen SDK, to re-compile the project.
    - Rerun the debugger and validate that your code now passes the suspected failure.
        - If you are now passing, you have verified that your regression test has "test coverage" for this particular failure mode
        - If you are still failing, you will need to debug your assembly test and go back into ip.conf to re-enable this BTYPE_REGRESSION_TEST and **re-generate** the processor model, ♨ Gen SDK
- Passing the introduced failure does not mean that you have complete test coverage of the BTYPE set of instructions. Please complete the test for all of the BTYPE instructions and their possible test conditions.
- At the end of your branch (b-type) instruction test sequence, ensure to keep four NOPs before and after the halt instruction
- The grading rubric for this phase includes test coverage and a **test escape** will negatively impact your grade

Checkpoint 5: Validating your branch (b-type) instructions

- Comment out the halt instruction at the end of your data hazard detection and forwarding test sequence so that as you step through your regression test, you will now step into your branch (b-type) test sequence
- Debug the regression test and step through the code. For the IA model, the branches will occur as you transition to the next instruction if the branch condition is true
    - As you step through the code, does the program branch as you suspected
- Once you reach the halt instruction after the PASS label in the branch (b-type) test sequence successfully, you have completed **Checkpoint 5.** This test sequence will be used in Phase 7.

- Phase 7, Branch and Jump Instructions, will also add new control-signals that will enable a program to change its program flow using jump instructions. Jump instructions change flow like branch instructions, but they are unconditional jumps. The next test sequence will be used to validate your Branch and Jump Instructions once implemented in Phase 7.
    - The next test sequence will be used to test the Jump Instructions
- The jump (j & i-type) instructions to be included in this test sequence are:
    - jalr   (i-type instruction)
    - jal   (j-type instruction)
- Additional information on these instructions can be found in **Appendix A:** phase3_ia_riscv32i RISCV32i supported assembly instructions
- There must be a minimum of one test sequence where the jump offset is negative. If there is not a negative jump case, this portion of the regression test will have a **test escape**. This can be implemented by having the jump label that occurs before the jump instruction.
- Each assembly instruction requires a comment statement
- At the end of your jump instruction test sequence, ensure to keep four NOPs before and after the halt instruction
- The grading rubric for this phase includes test coverage and a **test escape** will negatively impact your grade
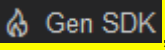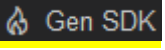

Checkpoint 6: Validating your jump instructions

- Comment out the halt instruction at the end of your branch (b-type) test sequence so that as you step through your regression test, you will now step into your jump instruction test sequence
- Debug the regression test and step through the code. For the IA model, the jumps will occur as you transition to the next instruction after the jump
    - As you step through the code, does the program run as you suspected
- If you get a get a failure that you are confident is not a programming error, open ip.conf and disable, make **false**, the JTYPE_REGRESSION_TEST variable/constant to disable or take the error out of the processor model for the JTYPE instructions
        - Once you make this change, you will need to save the file through the top menu bar

```
assignment3_ia_riscv32i > ||| ip.conf
 1
 2    [options]
 3        BOOT_START = 0x1000
 4        MEMORY_SIZE = 0x800000
 5        CSR_ENABLED = false
 6        CACHES_ENABLED = false
 7        ITYPE_REGRESSION_TEST = false
 8        RTYPE_REGRESSION_TEST = false
 9        RTYPE_IMM_REGRESSION_TEST = false
10        BTYPE_REGRESSION_TEST = false
11        JTYPE_REGRESSION_TEST = false
12        STYPE_REGRESSION_TEST = true
13        LOAD_REGRESSION_TEST= true
```

- ■ After you have saved the new configuration, you will need to **re-generate** SDK(ia), ⟨Gen SDK⟩, to re-compile the project.
- ■ Rerun the debugger and validate that your code now passes the suspected failure.
    - If you are now passing, you have verified that your regression test has "test coverage" for this particular failure mode
    - If you are still failing, you will need to debug your assembly test and go back into ip.conf to re-enable this JTYPE_REGRESSION_TEST and **re-generate** the processor model, ⟨Gen SDK⟩
- Passing the introduced failure does not mean that you have complete test coverage of the JTYPE set of instructions. Please complete the test for all of the JTYPE instructions and their possible test conditions.
- Once you reach the halt instruction at the end of the jump instruction test sequence, you have successfully passed **Checkpoint 6.** This test sequence will be used in Phase 7.

- Phase 8, Load and Store Instructions, will add both new data-paths and control-signals to enable these data operations to memory. In Phase 8, you will first add the Store Instruction functionality so that when you implement the Load Instructions, you can validate the loads by reading back the value written into memory.
- The s-type of instructions to be included in this test sequence are:
    - sw
    - sh
    - sb
- Additional information on these instructions can be found in **Appendix A**
- The regression_test for the Store Instruction test has prepared two registers that will be used for both the Store and Load Instruction tests.
    - x20: test value 1 to be used for the test sequences
    - x21: test value 2 to be used for the test sequences

- ○ x10: base address of data space for the test sequences
- ● 8 32-bit data spaces have been reserved at the memory location starting at the label DATA
  - ○ Your test will overwrite these NOP instructions
  - ○ This data space should not be executed as instructions due to the beq instruction before the data space will branch over the memory locations used for data

```
123        Store (s-type) operations
124    */
125        // Loading test data into registers for Store / Load tests
126        addi x20, x0, 0x765
127        slli x20, x20, 12
128        ori x20, x20, 0x432
129        slli x20, x20, 8
130        ori x20, x20, 0x10            // x20 = 0x76543210
131        xori x21, x20, 0xfff          // x21 = 0x89abcdef
132        // Load register x10 with base DATA memory location
133        addi x10, x0, (DATA >> 12)       // Assume DATA memory address less than 24-bits
134        slli x10, x10, 12               // Move the upper 12-bits to locations 12..23
135        ori x10, x10, (DATA & 0xfff)    // OR in the lower 12-bits to create all 24-bits
136        // Load register x11 with base DATA_MINU location
137        addi x11, x0, (DATA_MINUS >> 12)    // Assume DATA memory address less than 24-bits
138        slli x11, x11, 12                  // Move the upper 12-bits to locations 12..23
139        ori x11, x11, (DATA_MINUS & 0xfff)  // OR in the lower 12-bits to create all 24-bits
140        // start of Store Instruction test
141        nop
142        nop
143        nop
144        nop
145        halt
146        nop
147        beq x0, x0, LOAD_TEST
148        nop
149        nop
150    /*
151        Data Memory Space for regression test
152        - There are 8 NOP locations which are available
153             to be overwritten for test
154        - Accessing the first data location by 0 offset of x10 => 0(x10)
155        - Accessing the 1st byte in data space is 1 offset of x10 => 1(x10)
156        - Accessing the 2nd half-word in data space is 2 offset of x10 => 2(x10)
157        - Accessing the 2nd word in data space is 4 offset of x10 => 4(x10)
158    */
159    DATA:
160        nop
161        nop
162        nop
163        nop
164        nop
165        nop
166        nop
167        nop
168    DATA_MINUS:
169
170    /*
171        Load (l-type) operations
172    */
```

- To access data memory, stores and loads address the memory space as an offset of a base address. In the screenshot above, register x10 is stored in the base data memory address
  - The offset is the number of bytes to be added to the base address
  - For words to be data aligned, the offset must be word aligned (0, 4, 8, 12, …)
  - For half-words to be data aligned, the offset must be half-word aligned (0, 2, 4, 6, 8, 10, 12, …)
  - For bytes to be data aligned, the offset must be byte aligned (0, 1, 2, 3, 4, …)
- The value to be stored is aligned to the lower memory bits of the data register
  - Example:
    - x5 = 0x12345678
    - x6 = 0x1000   // based data memory address
    - sb x5, 3(x6)   => 0x1000 = 0x78000000
    - srli x7, x5, 8   // shift test data 8-bits to the right
    - sb x7, 2(x6)   => 0x1000 = 0x78560000
- Using no more than the 8 data memory locations, create your store instruction test validating the following:
  - sw: words are stored correctly in memory
  - sh: half-words are stored correctly in each of the two locations in a data word
  - sb: bytes are stored correctly in each of the four locations in a data word
- Each assembly instruction requires a comment statement
- After writing all your test conditions, create one more test of storing a word with a negative offset. As in a branch, the RISCV ISA s-type immediate is sign-extended
  - ex: sw x20, -4(DATA_MINUS) = sw x20, 28(DATA)

Checkpoint 7: Validating your store (s-type) instructions

- Comment out the halt instruction at the end of your jump test sequence so that as you step through your regression test, you will now step into your store instruction test sequence
- Debug the regression test and step through the code. For the IA model, the stores will occur as you transition to the next instruction after the store
  - You can validate your store operations by viewing the memory location directly using the Memory Window
  - Once you are in the debug view, you will be using the Memory window to validate that the stores to memory occurred as programmed

- ■ Select the Memory tab within the Debug view
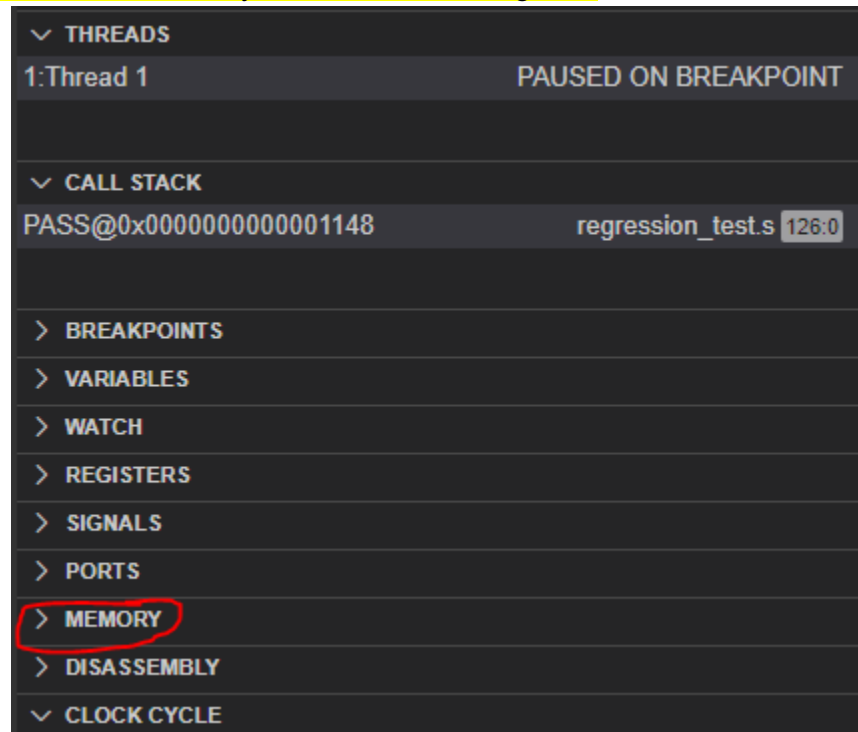
- ■ To add a memory range hover over the opened memory tab and click on the icon.

- ■ This will open up the memory range at the top of your window. Copy the value in register x10, the base memory data address, for the bottom bound of memory. For the upper bound, add 0x0050 to the number stored in x10 and enter that for the upper range.
  - ● For example, let's say the value in x10 is 0x2000, then the lower bound would be 0x2000 and the upper bound would be 0x2050. You would enter the range as 0x2000-0x2050. It is also shown in the screenshot below.
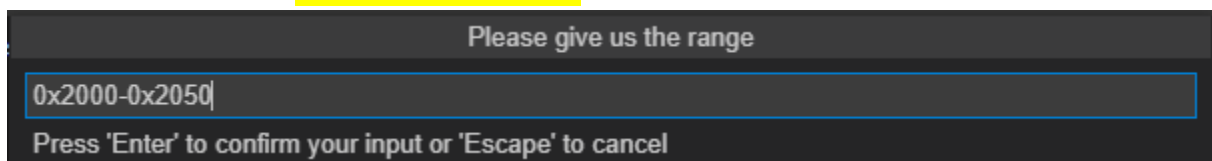
Navigate back to the memory tab. You should now see the memory range you entered. If the simulation is running, you should be able to open up the folder. The data presented is in little endian with the least significant byte on the right

■ Single step through your code. Compare the changed value in the comments to validate whether your writes to the memory locations match the expected values

● If there are incorrect values written to the memory locations, debug your regression test.

- If you get a get a failure that you are confident is not a programming error, open ip.conf and disable, make **false**, the STYPE_REGRESSION_TEST variable/constant to disable or take the error out of the processor model for the STYPE instructions
   - Once you make this change, you will need to save the file through the top menu bar

```
assignment3_ia_riscv32i  >  ﷽ ip.conf
 1
 2    [options]
 3        BOOT_START = 0x1000
 4        MEMORY_SIZE = 0x800000
 5        CSR_ENABLED = false
 6        CACHES_ENABLED = false
 7        ITYPE_REGRESSION_TEST = false
 8        RTYPE_REGRESSION_TEST = false
 9        RTYPE_IMM_REGRESSION_TEST = false
10        BTYPE_REGRESSION_TEST = false
11        JTYPE_REGRESSION_TEST = false
12        STYPE_REGRESSION_TEST = false
13        LOAD_REGRESSION_TEST= true
```

    - After you have saved the new configuration, you will need to **re-generate** SDK(ia), Gen SDK, to re-compile the project.
    - Rerun the debugger and validate that your code now passes the suspected failure.
        - If you are now passing, you have verified that your regression test has "test coverage" for this particular failure mode
        - If you are still failing, you will need to debug your assembly test and go back into ip.conf to re-enable this STYPE_REGRESSION_TEST and **re-generate** the processor model, Gen SDK
- Passing the introduced failure does not mean that you have complete test coverage of the STYPE set of instructions. Please complete the test for all of the STYPE instructions and their possible test conditions.
- Once you reach the halt instruction at the end of the store instruction test sequence, you have successfully passed **Checkpoint 7.** This test sequence will be used in Phase 8.

- Phase 8, Load and Store Instructions, will add both new data-paths and control-signals to enable these data operations from memory. In Phase 8, you will first add the Load Instruction functionality and use the previous Store instruction tests to validate the loads by reading back the value written into memory.
- To access data memory, load address the memory space as an offset of a base address. In the screenshot above, register x10 is stored the base data memory address
    - The offset is the number of bytes to be added to the base address
    - For words to be data aligned, the offset must be word aligned (0, 4, 8, 12, …)

- - For half-words to be data aligned, the offset must be half-word aligned (0, 2, 4, 6, 8, 10, 12, …)
    - For bytes to be data aligned, the offset must be byte aligned (0, 1, 2, 3, 4, …)
  - The value to be loaded will be placed in the lower memory bits of the data register
    - Example:
      - x6 = 0x1000   // based data memory address
      - 0x1000 = 0x87654321
      - lb x5, 1(x6)   => x5 = 0x00000043
      - slli x5, x5, 8   // shift test data 8-bits to the left
      - lb x7, 0(x6)   => 0x1000 = 0x00004321
  - Load bytes (lb) and half-words (lh) are sign extended while lbu and lhu are not sign-extend
    - If sign extended, the value store in memory will take the upper-bit of the load and copy it to all higher bits
      - x6 = 0x1000   // based data memory address
      - 0x1000 = 0x87654321
      - lb x5, 3(x6)   => x5 = 0xffffff87 (lb is sign-extended)
        - In the above, the byte, 0x87, in binary is 0b1000 0111. The most significant bit of the byte is a "1," and when it is signed extended, the higher 24 bits of the word are assigned the value of 1.
      - slli x5, x5, 8   // shift test data 8-bits to, x7 = 0xffff8700
      - lbu x7, 3(x6)   => x7 = 0x00000087 (lbu not signed extended)
        - In the above example of unsigned byte load, the upper byte bit of "1" (0x87 or 0b1000 0000) is not sign extended to the upper 24-bits. Instead, the upper 24-bits are assigned a "0."
      - or x5, x5, x7   => x5 = 0xffff8787
- The I-type of instructions to be included in this test sequence are:
  - lw
  - lh
  - lhu
  - lb
  - lbu
- Additional information on these instructions can be found in **Appendix A**
- Develop a test to validate all the above load instructions using the data that was stored into memory in the Store Instruction test sequence
  - Use a branch to compare the load with the expected return value and if the return does not match the expected value, branch to LOAD_FAIL

```
170    /*
171        Load (l-type)  operations
172    */
173    LOAD_TEST:
174        nop
175        nop
176        nop
177        nop
178        halt
179        nop
180        nop
181        nop
182        nop
183    LOAD_FAIL:                          // Using branch statements, if load does not
184        nop                             // return result expected, branch to LOAD_FAIL label
185        nop
186        nop
187        halt
188        nop
189        nop
190        nop
191        nop
```

- Each assembly instruction requires a comment statement
- After writing all your test conditions, create one additional test of loading a word with a negative offset. As in a branch, the RISCV ISA l-type immediate is sign-extended
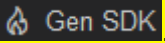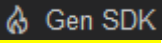 - ex: lw x20, -4(DATA_MINUS) = lw x20, 28(DATA)

Checkpoint 8: Validating your load (l-type) instructions

- Comment out the halt instruction at the end of your store test sequence so that as you step through your regression test, you will now step into your load (l-type) test sequence
- Debug the regression test and step through the code. For the IA model, the loads will occur as you transition to the next instruction after the load statement
  - As you step through the code, does the program continue to flow through the test sequence or branch to LOAD_FAIL?
  - If you branch to LOAD_FAIL, debug your regression_test
- If you get a get a failure that you are confident is not a programming error, open ip.conf and disable, make **false**, the LOAD_REGRESSION_TEST variable/constant to disable or take the error out of the processor model for the LOAD instructions
    - Once you make this change, you will need to save the file through the top menu bar

```
assignment3_ia_riscv32i > ### ip.conf
  1
  2    [options]
  3        BOOT_START = 0x1000
  4        MEMORY_SIZE = 0x800000
  5        CSR_ENABLED = false
  6        CACHES_ENABLED = false
  7        ITYPE_REGRESSION_TEST = false
  8        RTYPE_REGRESSION_TEST = false
  9        RTYPE_IMM_REGRESSION_TEST = false
 10        BTYPE_REGRESSION_TEST = false
 11        JTYPE_REGRESSION_TEST = false
 12        STYPE_REGRESSION_TEST = false
 13        LOAD_REGRESSION_TEST= false
 14
```

- ■ After you have saved the new configuration, you will need to **re-generate** SDK(ia), 🔥 Gen SDK, to re-compile the project.
- ■ Rerun the debugger and validate that your code now passes the suspected failure.
    - ● If you are now passing, you have verified that your regression test has "test coverage" for this particular failure mode
    - ● If you are still failing, you will need to debug your assembly test and go back into ip.conf to re-enable this LOAD_REGRESSION_TEST and **re-generate** the processor model, 🔥 Gen SDK
- ● Passing the introduced failure does not mean that you have complete test coverage of the LOAD set of instructions. Please complete the test for all of the LOAD instructions and their possible test conditions.
- ● Once you reach the halt instruction after the load instruction test sequence successfully, you have completed **Checkpoint 8.** This test sequence will be used in Phase 8.
- ● NOTE: The U type instruction test will not be implemented in this phase. In future phases there will be a provided test sequence for it.

- ● Once you have successfully completed all **Checkpoints**, uncomment all the halt statements to prepare your regression_test for the future phases

- ● Complete the phase
    - ○ Download your regression.s file within your regression/src folder
        - ■ Submit it into your course phase 3 canvas assignment
    - ○ You have completed Phase 3

Appendix A: phase3_ia_riscv32i RISCV32i supported assembly instructions

i-type: Immediates instructions
- addi rd, rs1, immediate   (rd = rs1 + immediate)
- slti rd, rs1, immediate   (set rd to 1, if rs1 < immediate)
- sltiu rd, rs1, immediate   (set rd to 1, if unsigned compare of rs1 < immediate)
- xori rd, rs1, immediate   (rd = rs1 ^ immediate)
- ori rd, rs1, immediate   (rd = rs1 | immediate)
- andi rd, rs1, immediate   (rd = rs1 & immediate)
- lb rd, immediate(rs1)   (rd = sign extended byte from memory at rs1 + immediate)
- lh rd, immediate(rs1)   (rd = sign extended halfword from mem at rs1 + imm)
- lw rd, immediate(rs1)   (rd = word from memory at rs1 + immediate)
- lbu rd, immediate(rs1)   (rd = not signed byte from memory at rs1 + immediate)
- lhu rd, immediate(rs1)   (rd = not signed halfword from memory at rs1 + immediate)
- jalr rd, immediate(rs1)   (rd = pc + 4, pc = rs1 + immediate)

r-type: Register-to-Register instructions
- add rd, rs1, rs2   (rd = rs1 + rs2)
- sub rd, rs1, rs2   (rd = rs1 - rs2)
- sll rd, rs1, rs2   (rd = rs1 << rs2)
- slt rd, rs1, rs2   (set rd to 1, if rs1 < rs2)
- sltu rd, rs1, rs2   (set rd to 1, if unsigned compare of rs1 < rs2)
- xor rd, rs1, rs2   (rd = rs1 ^ rs2)
- srl rd, rs1, rs2   (rd = rs1 >> rs2, logical shift of inserting 0s in upper bits)
- sra rd, rs1, rs2   (rd = rs1 >> rs2, arithmetic shift insert sign bit in upper bits)
- or rd, rs1, rs2   (rd = rs1 | rs2)
- and rd, rs1, rs2   (rd = rs1 & rs2)
- slli rd, rs1, immediate   (rd = rs1 << rtype immediate)
- srli rd, rs1, immediate   (rd = rs1 >> rtype immediate, logical shift)
- srai rd, rs1, immediate   (rd = rs1 >> rtype immediate, arithmetic shift)

s-type: Store instructions
- sb rs2, immediate(rs1)   (byte 0 of rs2 stored at memory location rs1 + immediate)
- sh rs2, immediate(rs1)   (byte 0 & 1 of rs2 stored at memory location rs1 + imm)
- sw rs2, immediate(rs1)   (byte 0,1,2, and 3 stored at memory location rs1 + imm)

b-type: Branch instructions
- beq rs1, rs2, immediate   (branch to pc + imm if rs1 = rs2)
- bne rs1, rs2, immediate   (branch to pc + imm if rs1 != rs2)
- blt rs1, rs2, immediate   (branch to pc + imm if rs1 < rs2)
- bge rs1, rs2, immediate   (branch to pc + imm if rs1 >= rs2)
- bltu rs1, rs2, immediate   (branch to pc + imm if rs1 < rs2 (unsigned compare))
- bgeu rs1, rs2, immediate   (branch to pc + imm if rs1 >= rs2 (unsigned compare))

j-type: Jump instructions
- jal rd, immediate   (rd = pc + 4, pc = pc + immediate)

u-type: upper immediate instructions
- lui rd, immediate   (rd upper 20 bits = immediate, lower 12 bits = 0)
- auipc rd, immediate   (rd = pc + 20-bit upper immediate)

For simulation
- halt   (halts simulation and exits debugger, run, or profiler)