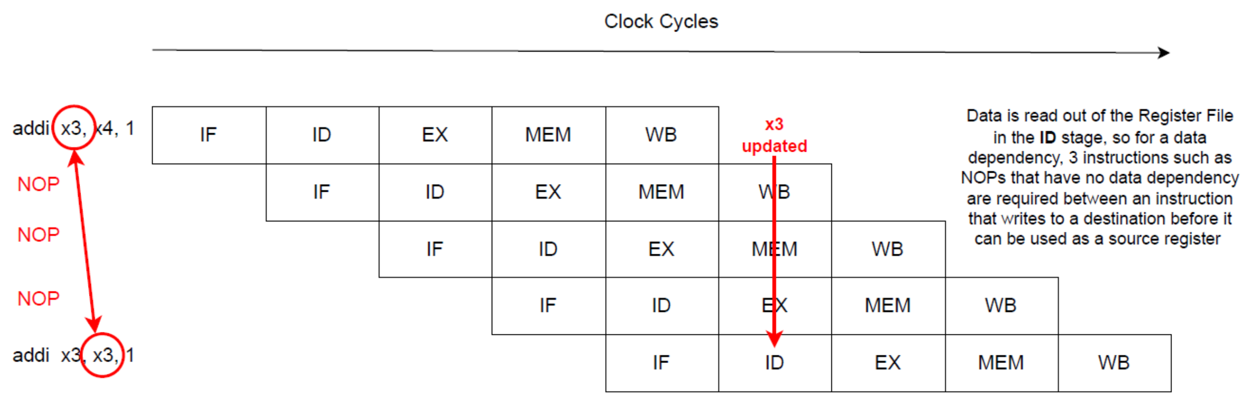
Computer Organization
5-stage RISCV32I Processor
Phase 6: Data Hazard Detection and Forwarding
Spring 2023

Objective: Parallelism is a key concept in Computer Architecture to improve performance. A 5-stage pipeline processor, enables a processor to simultaneously execute a portion of five different instructions simultaneously, providing parallelism. If an instruction in an earlier pipe stage, such as Instruction Decode (ID), requires the result from an instruction in a later pipe stage, as in the Execute stage, the implementation from the previous assignment 5 will calculate a different result than a single stage design. This is because the result of the instruction in the Execute stage has yet to be written to the Register File when the register file operands are read in the ID stage of this 5-stage implementation. One way to solve this “Data Hazard” is to have the compiler find an instruction that does not require the result from the Execute stage to be placed between the two instructions. This may work for a 5-stage design, but the same program binary code may not work in a 6 or 7-stage design. A better solution is to have the “hardware,” the processor, detect these Data Hazards and then forward the result directly from the later stage into the ALU input operand muxes, bypassing the register file. The register file still must be updated as expected in the Write Back (WB) stage.

For this assignment, you will add the logic to detect these Data Hazards. It will then generate the associated control-signals to forward the data-paths of the later stages as operands into the ALU instead of the Register File values. The hardware data hazard detection and forwarding will enable binary code to provide consistent results from either a 1, 5, or 7-stage implementation while maximizing parallelism (performance).

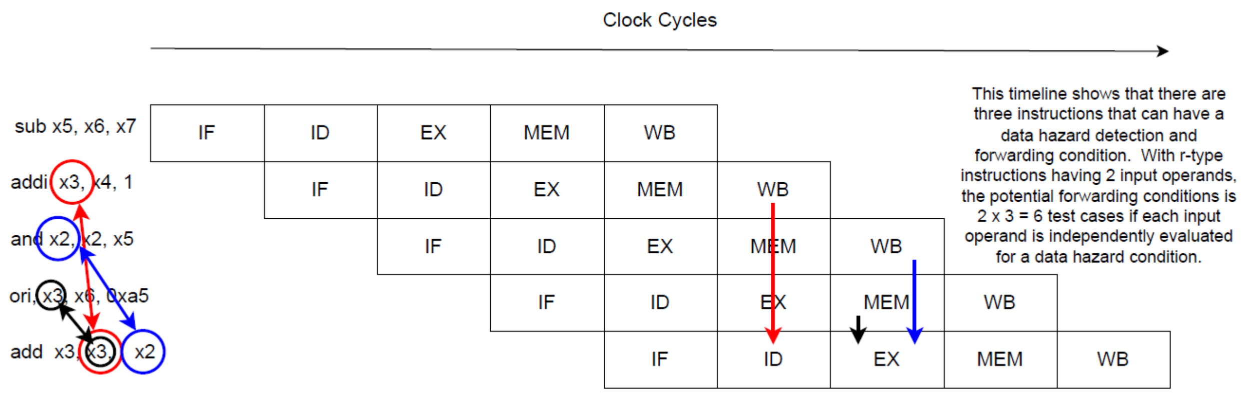
Key Learning Outcomes of this assignment:

- Data Hazard Detection: An important concept in data hazard detection is that an instruction in the Instruction Decode (ID) stage is aware of all the prior instructions that are in the later stages (EX, MEM, WB) of the processor. **Data Hazards** occur when the result of a later stage is required for an instruction operand in the EX stage. Either the pipeline must stall for the later stage instruction to complete, or the later stage result must be forwarded into the ALU. Instead of checking the instruction type such as an ADD or SUB in the later stages, we only need to check the regwrite signal and the rd. With the signal regwrite, we need to know whether the instruction is going to write a result to the Register File (the signal is set to a 1). As for the rd, we need to check if it matches either source 1 or 2 Register File operand defined by the instruction in the Instruction Decode (ID) stage.


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

- Data Forwarding: Data Hazards can be resolved by stalling the pipeline until the Data Hazard condition has been resolved, meaning the prior instruction has updated the Register File. However, a stall limits the pipeline's parallelism since only a portion of the 5-stages will proceed forward during a stall. Instead of stalling, the hazard result can be forwarded directly into the ALU operand input muxes enabling all pipeline stages to proceed, maximizing parallelism. With an instruction in the ID stage, there are three prior instructions in the processor's pipe stages which could result in a hazard (EX, MEM, and WB). The data forwarding must prioritize the forwarding from the most recent instruction hazard.


Clock Cycles
sub x5, x6, x7
addi x3, x4, 1
and x2, x2, x5
ori x3, x6, 0xa5
add x3, x3, x2
IF ID EX MEM WB
This timeline shows that there are three instructions that can have a data hazard detection and forwarding condition. With r-type instructions having 2 input operands, the potential forwarding conditions is 2 x 3 = 6 test cases if each input operand is independently evaluated for a data hazard condition.

5-stage pipeline flow showing possible Data Hazard Detection and Forwarding

Instructions:

Summary of the work you will be doing in this phase:

- Update the diagrams.net schematics from phase 4 to include all the muxes necessary for data forwarding.
- Update all the specified Codasip files to implement data forwarding. The files will be modifying in this phase are ca_defines.hcodal (create new enums to direct values from/to the correct pipeline latches), ca_resources.codal (create new variables for data

forwarding), ca_pipe_stage2_id.codal (forward values from writeback to decode to mimic hardware behavior) and ca_pipe_stage3_ex.codal (forward values either from EX/MEM or MEM/WB pipeline latches when data hazards arise).

Instructions to update the schematics:

- ♦ Copy the schematic standardname4.xml to standardname6.xml (i.e. for me it would be ekeller4.xml to ekeller6.xml). Open the schematic in diagrams.net. The changes required for data hazard forwarding (DHF) will be implemented in EX Stage.
- ♦ Figure 1, which is Figure 4.52 from the textbook, shows the components necessary to implement DHF in the EX stage.
  - Add two 3-1 Multiplexor components for the Mux elements in each of the ALU source inputs.
  - Assign appropriately named signals for the multiplexor select signals “Forward1” (ForwardA in Figure 1) and “Forward2” (ForwardB in Figure 1), such as s_ex_fwd1. Note that several of the input signals to these multiplexors come from later stages (ME and WB). Note that on the src2 side of the ALU, forwarding replaces the register file inputs so that the forwarding multiplexor must be before the immediate multiplexor from Phase 4, which is not shown in Figure 1.

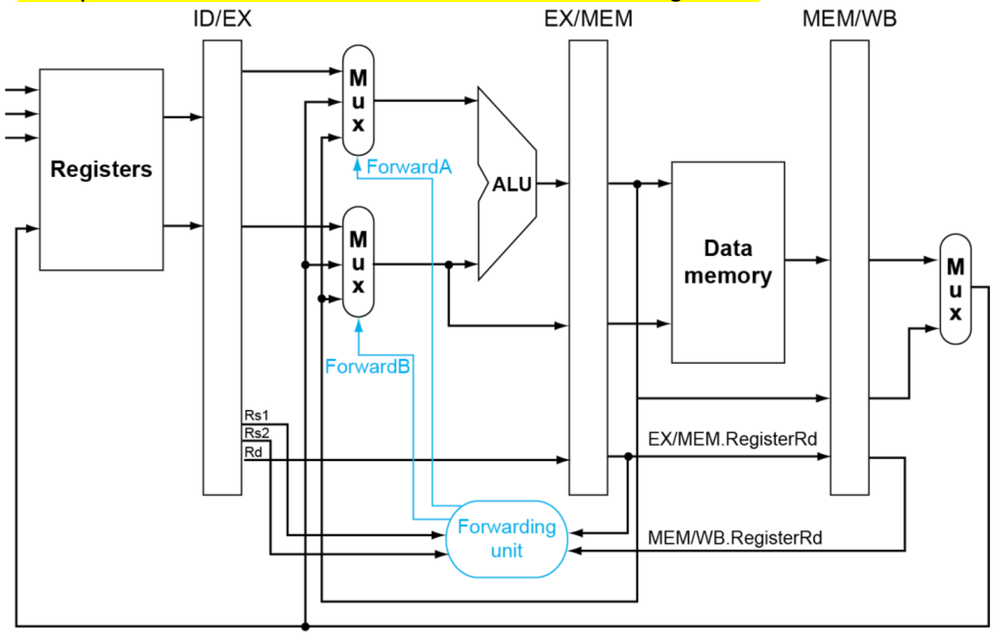


Figure 1

  - Add a control element named FWDCTL with the desired inputs and outputs to create the Forwarding Unit and name it appropriately. This control block will implement the functions in section 4.7 of the textbook. The multiplexor selects are described in Figure 4.53 in the textbook, replicated in Figure 2 below (I replaced ForwardA for Forward1 and ForwardB for Forward2) One set of equations in Figure 3 below shows the required input signals. The source select signals in the EX stage must be pipelined from the ID stage, so these pipeline registers must be added in the ID stage.

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Figure 2

MEM hazard

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
        and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))  Forward1 = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
        and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))  Forward2 = 01
```

Figure 3

- Register File Forwarding Schematic:
  - Register file forward (RFF) is not part of a normal RISC-V implementation, but is necessary in order to overcome a functional issue with the Codasip Register File module. In the textbook section 4.6, the Register File is assumed to write the result data into the register selected by the rd field in the first half of the clock cycle in the WB stage. This means that if a register written in the WB stage of an instruction is read in the ID stage by another instruction (i.e. read and write occur in the same clock cycle) the read data should be the data written in that cycle.
  - In the Codasip Register File model, however, the write data is written to the register selected by rd at the end of the clock cycle in the WB stage. This means that if that register is read in the ID stage by another instruction, the read data will be the old

    data in the register (before the write) and the behavior will not match the RISC-V architectural assumptions.
  - From a software standpoint, the code sequence is, for example:
    - add xA, xB, xC
    - instruction not writing or reading xA
    - instruction not writing or reading xA
    - add xD, xA, xA
  - In this case the fourth instruction should use the value of xA written in the first instruction. However, with the Codasip Register File, the fourth instruction will use the value in xA prior to the execution of the first instruction.
  - RFF is necessary to allow the Codasip implementation to match RISC-V, and will be implemented in the ID stage. This will be very similar to DHF – each Register File output will have a multiplexor selecting either the Register File output or the data to be written, and there will be a control block (RFFWD) which receives a set of inputs and produces the control signals for each of the muxes.
  - Update the diagrams.net schematic decode tab to include all the necessary new muxes, signals and registers to implement RFF.
- Examples: Figure 4 shows rough examples of what the ID Stage and EX Stage schematic pages should look like.

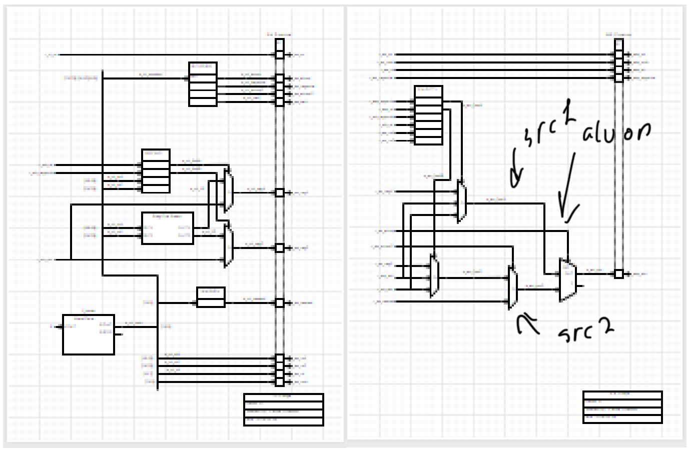


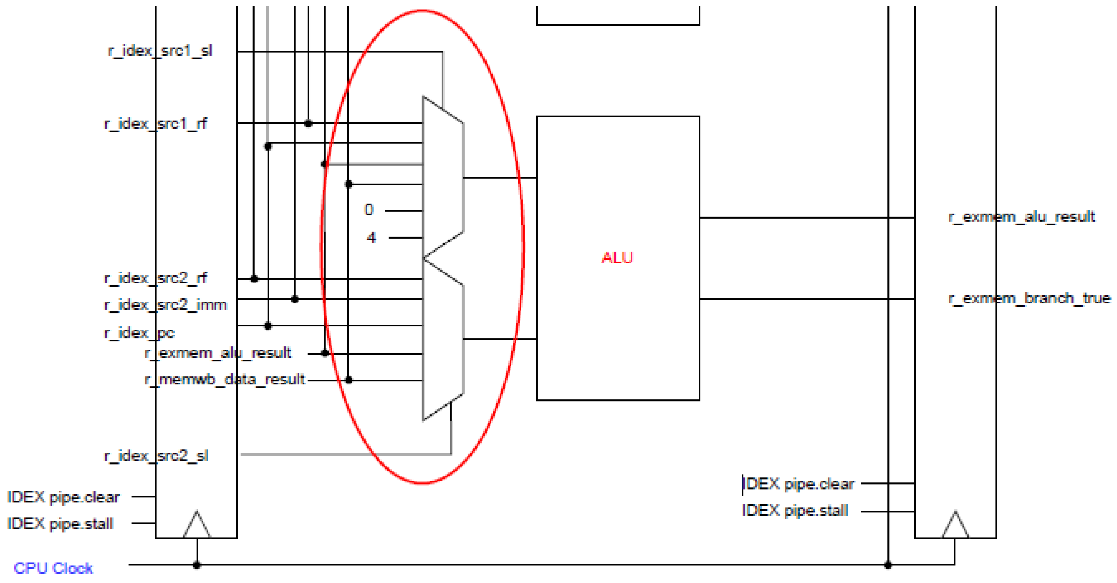
Figure 4

- Download the xml file from diagrams.net and submit the Updated Schematic
- Although not required, it is recommended that you submit the updated schematic as soon as it is complete, as standardname6.xml via Canvas. This will allow for feedback prior to creating the Codasip code. Your grade will include the correctness of your schematic similar to phase 4. The schematic will count towards 20% of phase 6 grading.

Instructions to update the Codasip files:

- For Assignment 6, you will be using the completed project from Assignment 5. If you have not completed Assignment 5, work with the TAs or professor to complete the assignment.

- Import the hazard_detection_test project using the following git clone command
    - git clone https://github.com/CompOrg-RISCV/hazard_detection_test.git
- Create the Data Forwarding data-paths
    - To enable Data Forwarding, data-paths from the ALU results in the Memory stage (MEM) and Write Back (WB) stages must be added
    - Add inputs to both the ALU operand 1 and operand 2 input muxes
    - Open ca_defines.codal and create two additional inputs (enums) to both muxes
        - Operand 1 mux: enum alu_src1_sel
            - ALU_SRC1_EXMEM_DF: EX stage Data Forwarding
            - ALU_SRC1_MEM_DF: ME stage Data Forwarding
        - Operand 2 mux: enum alu_src2_sel
            - ALU_SRC2_EXMEM_DF: EX stage Data Forwarding
            - ALU_SRC2_MEM_DF: ME stage Data Forwarding



    - When you declare the size of a signal you have used constants declared by #define statements, specifically ADDR_W which equates to 32 or BOOLEAN_BIT which equates to 1. The number of select signals to a mux may vary as you change the number of items to a mux. For example, by increasing the number of enum items, mux inputs, from 2 to 3 in ca_defines.codal, the number of select lines required to address these enums (the mux inputs) increases from 1 to 2 (2-bits wide). By using the bitsizeof operator for the number of bits for the select lines in our example below, the definition will automatically update as the number of enum items increases or decreases which then will correctly size the number of bits resourced for the select lines in ca_resources.codal. Enabling automatic bit sizing increases both design efficiency and decreases errors. Please check out the following operand in declaring the ALU_SRC1_SEL_W and ALU_SRC2_SEL_W.

FAQ: enums and switch statements: enums is a level of abstraction that makes programming easier and less error prone. The enum is a list of symbols where each symbol represents a distinct number which can be used by a switch statement's case statements. A switch statement will execute whose case's value matches the switch statement variable. An important benefit of the enum is not just the abstraction of a number, but the automation of any change of enumeration's value. For example, if the enum's symbol changes value from 0 to 1, all locations throughout the design will now be evaluated as a 1 instead of 0. This automation improves efficiency while reducing errors.

FAQ: bitsizeof() macro: Automation enables increased efficiency with less errors. The bitsizeof() macro can be used to automate the declaration of the signal width of the multiplexer select lines. This video describes what the bitsizeof() macro returns and how it is used to declare the number of multiplexer select lines. The video walks through an example where bitsizeof() is used in a processor's Cycle Accurate (CA) project files; ca_defines.hcodal and ca_resources.codal.

```
#define ALU_SRC1_SEL_W          bitsizeof(enum alu_src1_sel)
#define ALU_SRC2_SEL_W          bitsizeof(enum alu_src2_sel)
```

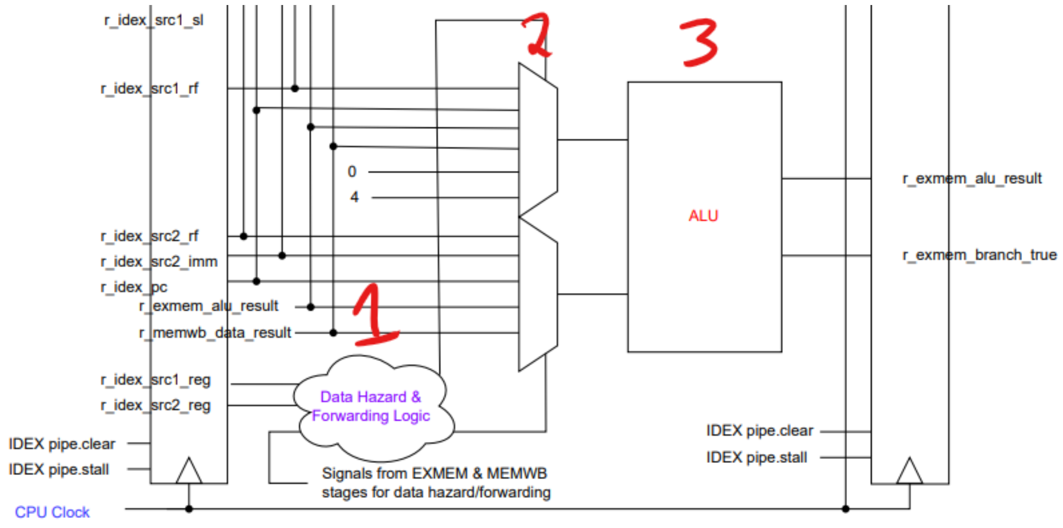
bitsizeof operator example from ca_defines.codal

```
// Signals generated by the Instruction Decoder
signal bit[CNTL_BIT]       s_id_regwrite;          // Write to rd (write register) if true
signal bit[ALU_SRC1_SEL_W] s_id_alusrc1;           // src1 operand mux select line
signal bit[ALU_SRC2_SEL_W] s_id_alusrc2;           // src2 operand mux select line
```

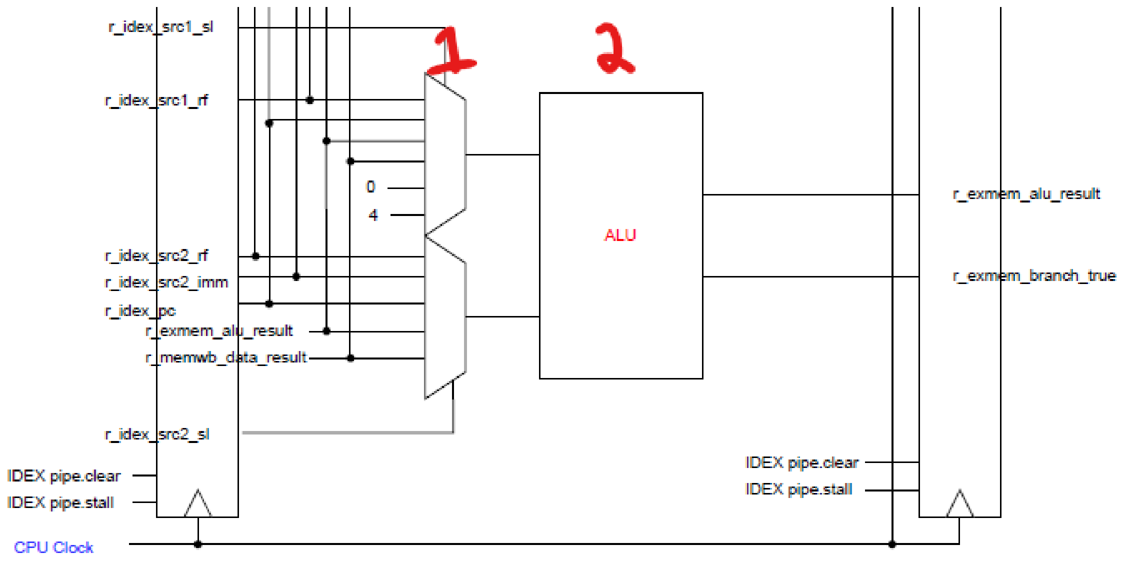
Using auto-sizing #defines to set mux select line width in ca_resources.codal

- With the addition of two additional inputs for both the ALU operand muxes, go to the ex event in ca_pipe3_ex.codal. For both of the src1 and src2 mux switch statements, add two case statements corresponding to the two additional enums created for each set of select lines.
  - Following the nomenclature for this processor, the ALU_SRCx_EXMEM_DF case statement will (data) forward, DF, the ALU result from the EXMEM pipeline register
  - Complete all four of these case statements by setting their respective src1 or src2 input operand signals with the respective pipeline alu_results
- The data-path has now been routed for the ALU operands in the EX stage

- To utilize the data forwarding, data hazard detection for both the EXMEM and MEM stages must be determined to select the data forwarding data-paths of the ALU input operand muxes.
    - Referring back to the Data Hazard Learning Outcome, an earlier stage knows what instructions have proceeded it in the later pipeline stages
    - The data hazard and forwarding logic can be performed in the Execute (EX) stage per the drawing below by adding the mux select logic for the data hazard and forwarding into the timing path. This logic must be calculated before a valid operand is passed into the ALU. For the ALU to produce a valid result, three logic blocks must be completed in series.

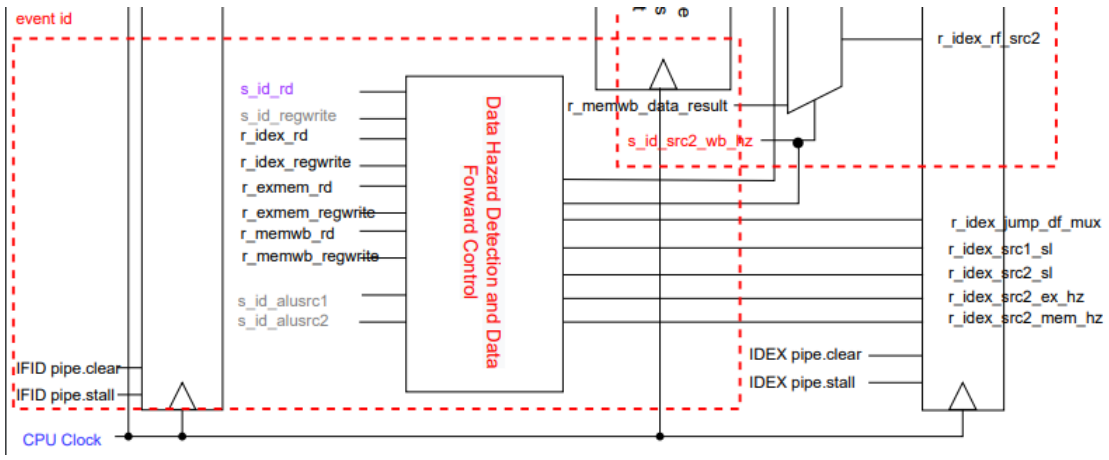


    - . To minimize the total time required in the EX stage, the data hazard detection and forwarding logic for the src1 and src2 select lines will be moved to the ID stage. This is based on the assumption that the ALU timing path will require the longest clock period of all stages due to the complexity of the ALU logic (longer clock period means slower clock frequency). Reducing the time path in the EX stage to just two functional blocks, the Input Operand Muxes and the ALU, will minimize the EX stage timing path which reduces the clock period resulting in maximizing the overall processor clock frequency.



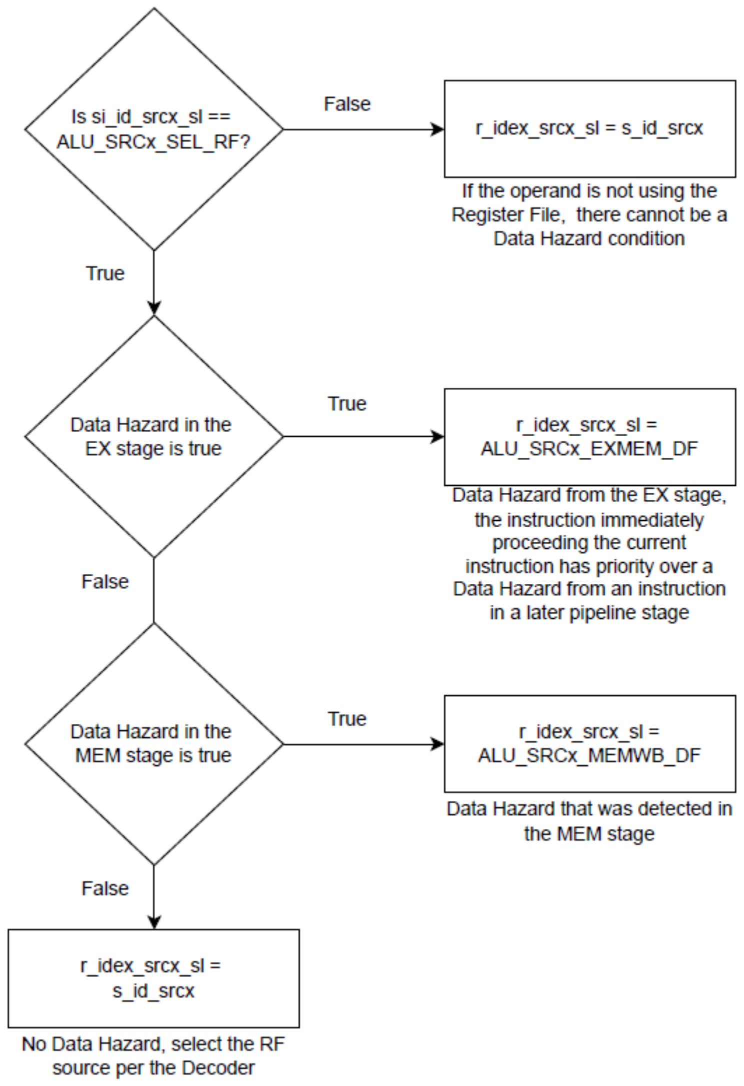
1

- The Data Hazard detection should be performed in the ID event immediately before the call to the id_output() event. You must determine the hazards before calling the id_output() event where the src1 and src2 mux select lines are evaluated to pass into the pipeline registers for the EX stage.
- In the ID section of ca_resources.codal, create four boolean **signals** (what definition will you use to set a single bit wide logic signal?) which will transfer the result of your Data Hazard detection logic using the nomenclature of this course. Using best programming practices is included in the grading rubric.
  - SRC1 EX Data Hazard: The instruction in the EX stage whose destination register, rd, is identical to the src1, rs1, register parsed in the ID stage and will be updated, written (regwrite in EX) when the EX alu result gets to the WB stage
  - SRC1 MEM Data Hazard: The instruction in the MEM stage whose destination register, rd, is identical to the src1, rs1, register parsed in the ID stage and will be updated, written (regwrite in MEM) when the WB alu result gets to the WB stage.
  - SRC2 EX Data Hazard: The instruction in the EX stage whose destination register, rd, is identical to the src2, rs2, register parsed in the ID stage and will be updated, written when the EX alu result gets to the WB stage
  - SRC2 MEM Data Hazard: The instruction in the MEM stage whose destination register, rd, is identical to the src2, rs2, register parsed in the ID stage and will be updated, written when the WB alu result gets to the WB stage



- With the **data hazard signals** defined, go into ca_pipe2_id.codal and insert your Data Hazard detection logic immediate before id_output() in the id event
  - The second part of the Data Hazard Learning Outcome is that the only information that is required from the later stages is whether it will be writing, regwrite, to the Register File and its destination register, rd.
  - Note: when doing the data hazard detection logic we want to check the pipeline register before the stage we are checking. For example, when checking for a data hazard in the EX stage, we will be looking at the rd and regwrite values of the ID pipeline register which is the active signal in the EX pipeline stage.
  - Use the c-programming ternary (?) operator or an IF statement to set each of the newly added signal resources true if the conditions are met and false if they are not
- With the Data Hazard detection logic completed, the src1 and src2 operand mux select lines must be updated if a hazard is indicated (true). Now find the r_id_src1_sel and r_id_src2_sel signals in id_output() event.
  - Another important planning tool is flowcharts. In this flowchart, source for either src1 or src2 will be the most recently updated value. Implement the below flowchart to properly assign the two source sel ID pipeline registers to select the correct input operand mux input

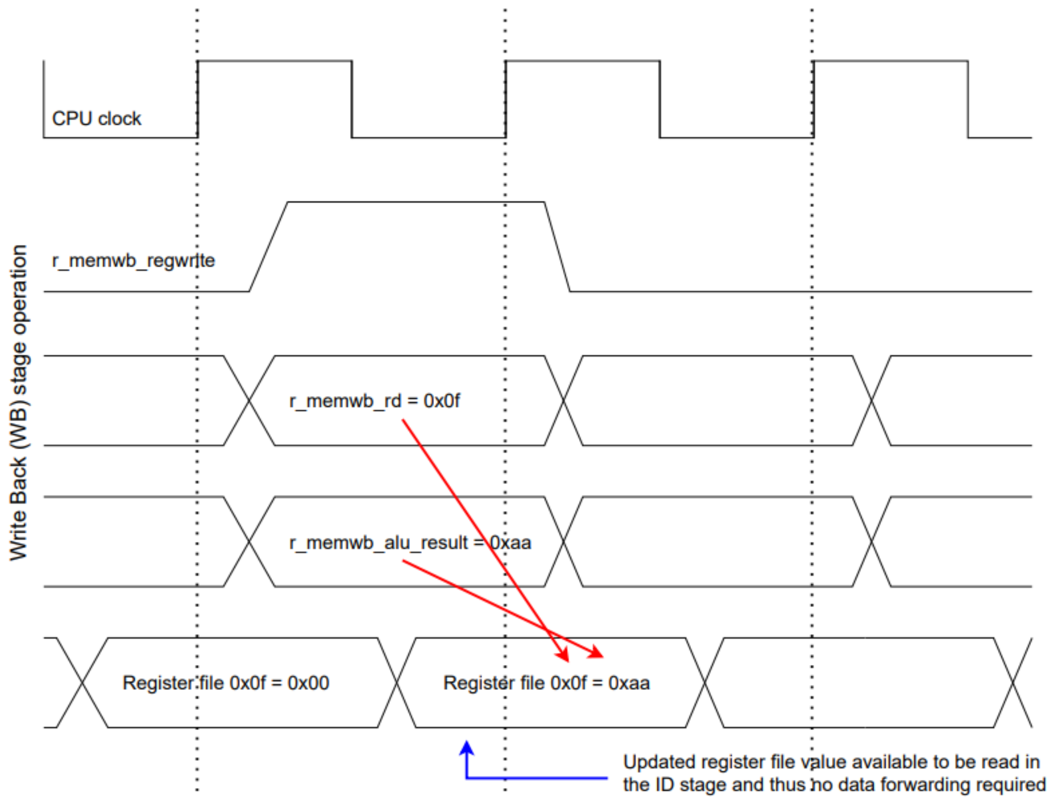
Data Forwarding Flowchart for ALU operand muxes



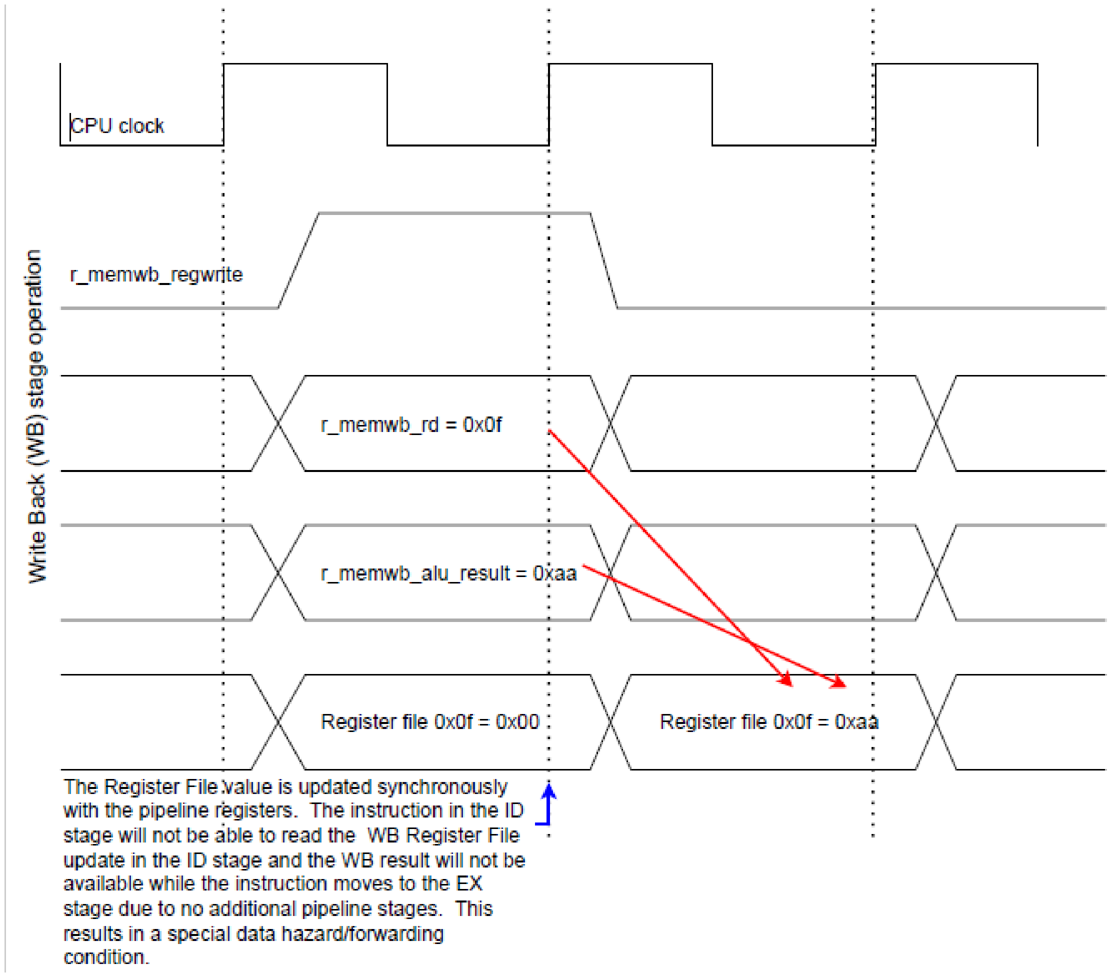
Is si_id_srcx_sl == ALU_SRCx_SEL_RF?
False
r_idex_srcx_sl = s_id_srcx
If the operand is not using the Register File, there cannot be a Data Hazard condition
True
Data Hazard in the EX stage is true
True
r_idex_srcx_sl = ALU_SRCx_EXMEM_DF
Data Hazard from the EX stage, the instruction immediately proceeding the current instruction has priority over a Data Hazard from an instruction in a later pipeline stage
False
Data Hazard in the MEM stage is true
True
r_idex_srcx_sl = ALU_SRCx_MEMWB_DF
Data Hazard that was detected in the MEM stage
False
r_idex_srcx_sl = s_id_srcx
No Data Hazard, select the RF source per the Decoder

- The Data Hazard Detection and ALU operand Data Forwarding is now complete

- What about the prior instruction in the WB stage? Does it need to be detected as a hazard and be forwarded?
  - The Data Hazard detection in the WB stage does need to be detected and forwarded, but in this implementation, it is a special case
  - There are implementations where the write to the Register FIle occurs on the falling edge of the CPU clock instead of the rising edge that the pipeline registers utilize. In these implementations, the data written into the register file on the falling edge would be available to be read out in the ID stage and written into the ID pipeline register on the next positive rising CPU clock edge, resulting in non-data hazard condition


CPU clock
r_memwb_regwrite
Write Back (WB) stage operation
r_memwb_rd = 0x0f
r_memwb_alu_result = 0xaa
Register file 0x0f = 0x00
Register file 0x0f = 0xaa
Updated register file value available to be read in the ID stage and thus no data forwarding required

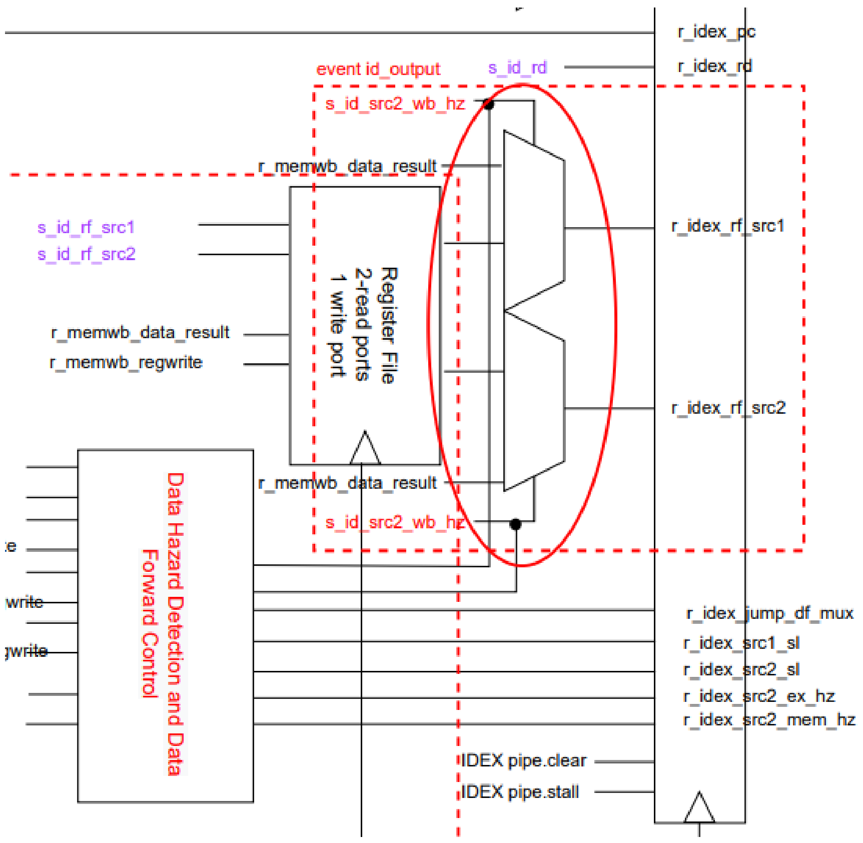
- Our implementation is based on a Register File that updates on the rising edge of the CPU clock, synchronous with the pipeline registers. When the instruction in the ID stage passes into the EX stage, the instruction in the WB stage will have completed (just updated the Register File) and is no longer available to forward back into the ALU src1 and src2 operand muxes. In this design, the Register File is written when the pipeline registers are updated.


Write Back (WB) stage operation
CPU clock
r_memwb_regwrite
r_memwb_rd = 0x0f
r_memwb_alu_result = 0xaa
Register file 0x0f = 0x00
Register file 0x0f = 0xaa
The Register File value is updated synchronously with the pipeline registers. The instruction in the ID stage will not be able to read the WB Register File update in the ID stage and the WB result will not be available while the instruction moves to the EX stage due to no additional pipeline stages. This results in a special data hazard/forwarding condition.

- For this special data hazard/forwarding conditions, the forwarding must be done in the ID stage and **not** the EX stage.

- For this implementation that uses synchronous writes to the Register File on the rising clock edge, the ALU result in the WB stage must be forwarded in the ID stage.
    - Go to the ca_resources.codal and create two new control signals (boolean) for a WB hazard detection for the src1 operand and for the src2 operand
    - In the id event in ca_pipe2_id.codal, after the instruction has been parsed, determine if a WB hazard has been detected for src1(rs1) and src2 (rs2)
    - In the id_output() event, update the assignment of the r_id_src1_rf and r_id_src2_rf pipeline registers accordingly. If there is no WB data hazard, continue to assign the output of the register file to the appropriate pipeline register values. If there is a data hazard, create a new data-path by assigning these pipeline registers with the current MEM ALU result pipeline register.

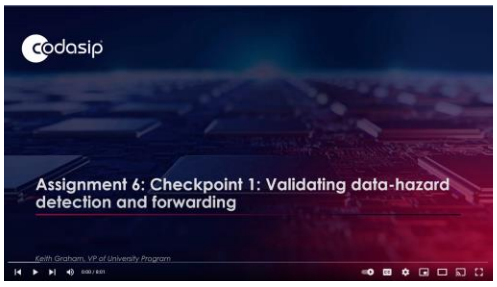
- For example, if there is a WB data detected in src1, you would assign r_id_src1_rf = r_mem_data_result.



  - We now know that all data hazard conditions have be taken accounted for because we have a detection circuit for each of the later pipeline stages and a data forwarding path from each of these later pipeline stages

---

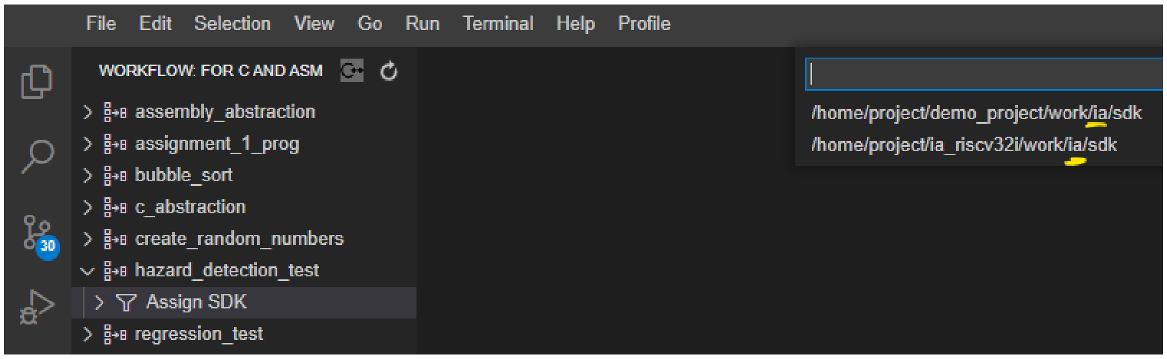
Checkpoint 1: Validating data-hazard detection and forwarding


codasip
Assignment 6: Checkpoint 1: Validating data-hazard detection and forwarding
Keith Graham, VP of University Program

Assignment 6: Checkpoint 1: Validating data-hazard detection and forwarding: Assignment 6: Checkpoint 1 video describes the test coverage of the checkpoint 1 regression test, hazard_detection_test.s, demonstrates the steps to launch the test in the debugger, and how to use the test's comments to focus the debug of your processor's Cycle Accurate (CA) project.

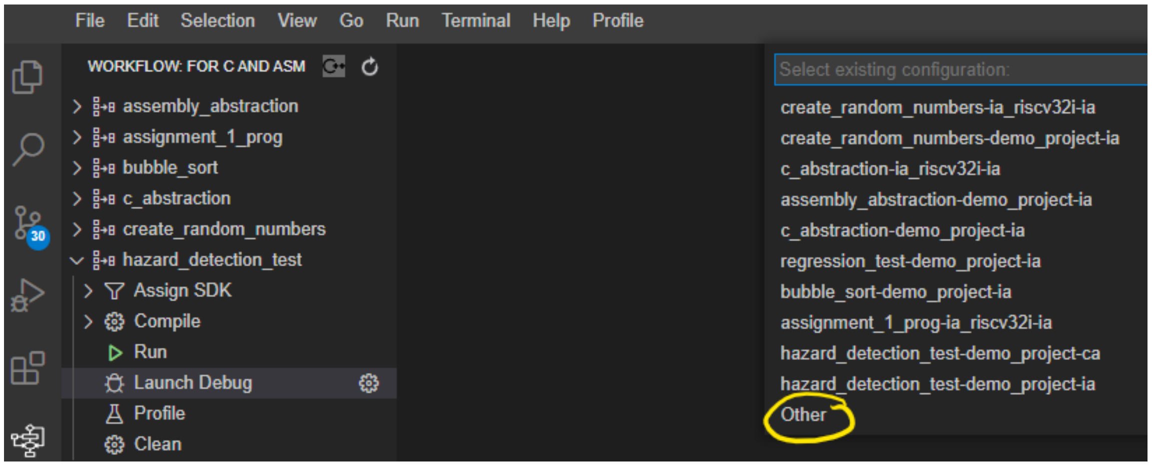
- It is time to try your data hazard and data forwarding
- Preparing your hazard_detectin_test to **Launch Debug(ger)**
    - Assign your processor's project SDK(ia) to the hazard_detection_test software project
    - Set the Compiler Configuration to run an assembly test
    - Set the Debugger Configuration to stop after 0 instructions after startup
    - Compile the hazard_detection_test

- In the latest version of Codasip Studio, you will not be assigning the CA SDK(ca) model to the software project. You will select the CA SDK(ca) model to debug through the Launch Debug dialog box.

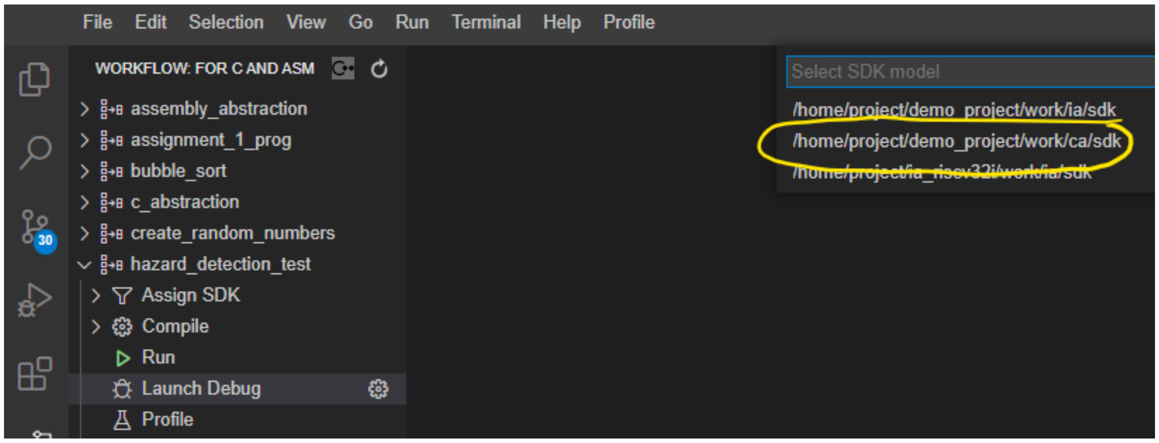


/home/project/demo_project/work/ia/sdk
/home/project/ia_riscv32i/work/ia/sdk

- To launch the debugger, you will click on Launch Debug. The dialog box will present a history of previous debug sessions, if you do not see the CA model for your project, select "**Other**," and options for new debug sessions will be presented



Select existing configuration:
create_random_numbers-ia_riscv32i-ia
create_random_numbers-demo_project-ia
c_abstraction-ia_riscv32i-ia
assembly_abstraction-demo_project-ia
c_abstraction-demo_project-ia
regression_test-demo_project-ia
bubble_sort-demo_project-ia
assignment_1_prog-ia_riscv32i-ia
hazard_detection_test-demo_project-ca
hazard_detection_test-demo_project-ia
Other

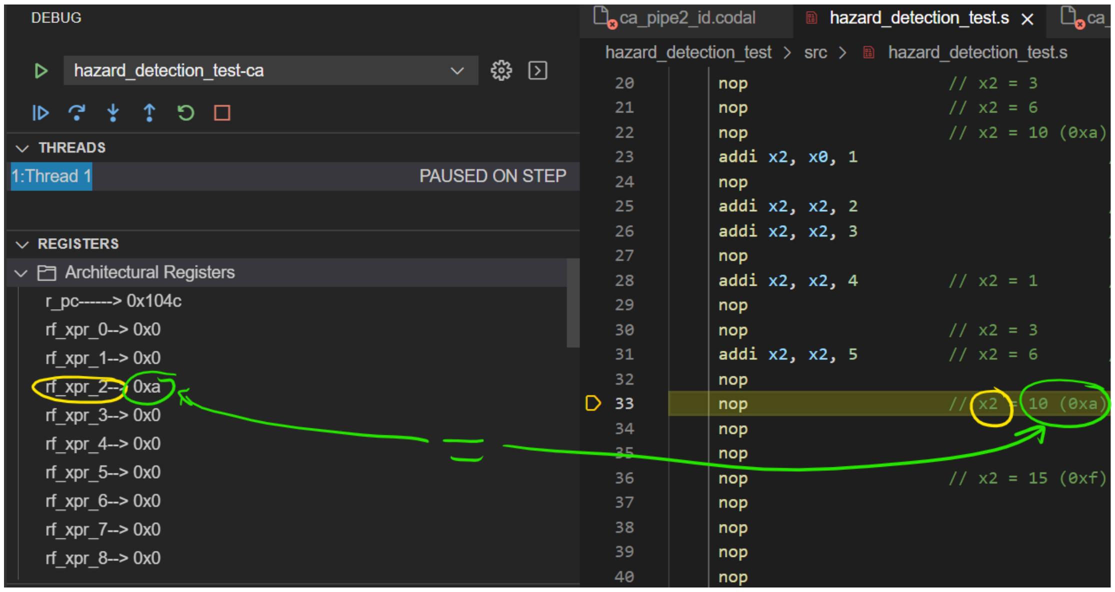


```
Select SDK model
/home/project/demo_project/work/ia/sdk
/home/project/demo_project/work/ca/sdk
/home/project/ia_riscv32i/work/ia/sdk
```

FAQ: Finding your CA model to debug: The Launch Debug task under each software project in the Workflow Perspective brings up a history of past debug sessions. The first time you attempt to run a debug session with your Cycle Accurate (CA) model, you will need to select "Other" to request a dialogue box to locate your CA model.

After you have run the debugger once with your CA model, the next time you attempt to launch the debugger, your CA model will be in the debug history to select.

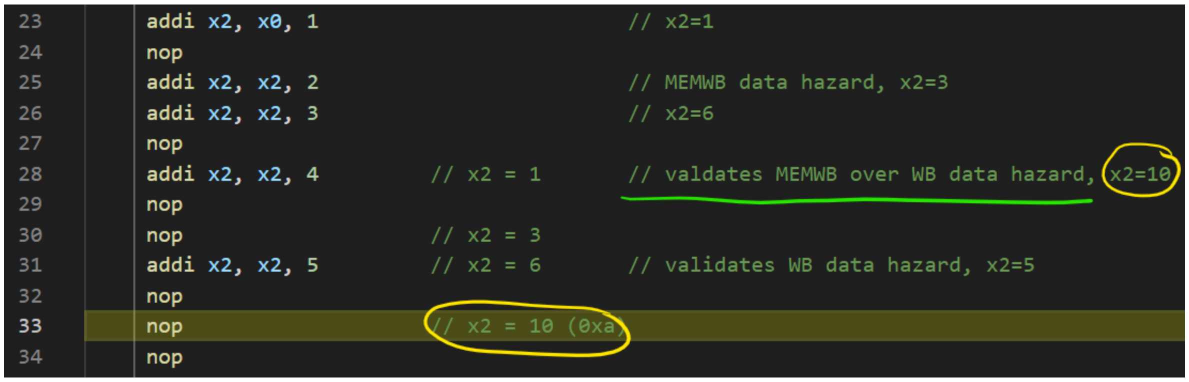
- Step through the hazard_detection_test. This test is designed to test all possible conditions
  - Use the //comment statements that begin mid-screen to determine what the register value should be at that line number



```
r_pc------> 0x104c
rf_xpr_0--> 0x0
rf_xpr_1--> 0x0
rf_xpr_2--> 0xa
rf_xpr_3--> 0x0
rf_xpr_4--> 0x0
rf_xpr_5--> 0x0
rf_xpr_6--> 0x0
rf_xpr_7--> 0x0
rf_xpr_8--> 0x0
```

```
20    nop                       // x2 = 3
21    nop                       // x2 = 6
22    nop                       // x2 = 10 (0xa)
23    addi x2, x0, 1
24    nop
25    addi x2, x2, 2
26    addi x2, x2, 3
27    nop
28    addi x2, x2, 4            // x2 = 1
29    nop
30    nop                       // x2 = 3
31    addi x2, x2, 5            // x2 = 6
32    nop
33    nop                       // x2 = 10 (0xa)
34    nop
35    nop
36    nop                       // x2 = 15 (0xf)
37    nop
38    nop
39    nop
40    nop
```

- If the value **does not match** the expected value, use the comments further up and further to the right for a hint on the possible test case which is failing



```
addi x2, x0, 1                                  // x2=1
nop
addi x2, x2, 2                                  // MEMWB data hazard, x2=3
addi x2, x2, 3                                  // x2=6
nop
addi x2, x2, 4          // x2 = 1              // valdates MEMWB over WB data hazard, x2=10
nop
nop                     // x2 = 3
addi x2, x2, 5          // x2 = 6              // validates WB data hazard, x2=5
nop
nop                     // x2 = 10 (0xa)
nop
```

- If needed, set breakpoints within your processor codAL project to debug your data hazard logic and data path

- Once you get to the halt at the bottom of this assembly routine without any errors (miss matches) with the expected results, you have successfully completed Checkpoint 1 and this assignment.

- Completing the assignment
  - Complete the assignment worksheet
  - Clean your CA model
  - Export your processor assignment (if you don’t clean it first, it will be >1GB)
  - Submit your exported assignment on Canvas for grading

Appendix A: YouTube videos for Assignment 6

Assignment Videos:

- **Assignment 6: Checkpoint 1: Validating data-hazard detection and forwarding**
  - Assignment 6: Checkpoint 1 video describes the test coverage of the checkpoint 1 regression test, hazard_detection_test.s, demonstrates the steps to launch the test in the debugger, and how to use the test's comments to focus the debug of your processor's Cycle Accurate (CA) project.
  - https://www.youtube.com/watch?v=0Gi4obgUfV8&list=PLTUn6Ox9e6q2ienoqI3KClFtRPMqO28uj&index=12&t=69s

Frequently Asked Questions (FAQs) Videos

- It is intended for students to provide them **real-time support** who have been assigned a project-based learning assignment, based on the Codasip Curriculum.
- **FAQ: bitsizeof() macro**
  - Automation enables increased efficiency with less errors. The bitsizeof() macro can be used to automate the declaration of the signal width of the multiplexer select lines. This video describes what the bitsizeof() macro returns and how it is used to declare the number of multiplexer select lines. The video walks through an example where bitsizeof() is used in a processor's Cycle Accurate (CA) project files; ca_defines.hcodal and ca_resources.codal.
  - https://www.youtube.com/watch?v=SF6edheACHk&list=PLTUn6Ox9e6q1ii0fp-N_GDPZjAtkDZmqe&index=13
  -
- **FAQ: enums and switch statements**
  - enums is a level of abstraction that makes programming easier and less error prone. The enum is a list of symbols where each symbol represents a distinct number which can be used by a switch statement's case statements. A switch statement will execute whose case's value matches the switch statement variable. An important benefit of the enum is not just the abstraction of a number, but the automation of any change of enumeration's value. For example, if the enum's symbol changes value from 0 to 1, all locations throughout the design will now be evaluated as a 1 instead of 0. This automation improves efficiency while reducing errors.
  - https://www.youtube.com/watch?v=UNbDe-XOCWY&list=PLTUn6Ox9e6q1ii0fp-N_GDPZjAtkDZmqe&index=14
- **FAQ: Finding your CA model to debug**
  - The Launch Debug task under each software project in the Workflow Perspective brings up a history of past debug sessions. The first time you attempt to run a debug session with your Cycle Accurate (CA) model, you will need to select "Other" to request a dialogue box to locate your CA model.

- After you have run the debugger once with your CA model, the next time you attempt to launch the debugger, your CA model will be in the debug history to select.
- https://www.youtube.com/watch?v=YpF1G2gqiys