Computer Organization
5-stage RISCV32I Processor
Phase 6: Data Hazard Detection and Forwarding
Fall 2022

Due Dates:

- This is a 1-week phase
- Due date: Monday, October31st, 2022 at 11:59pm

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 tlehman4.xml to tlehman6.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.

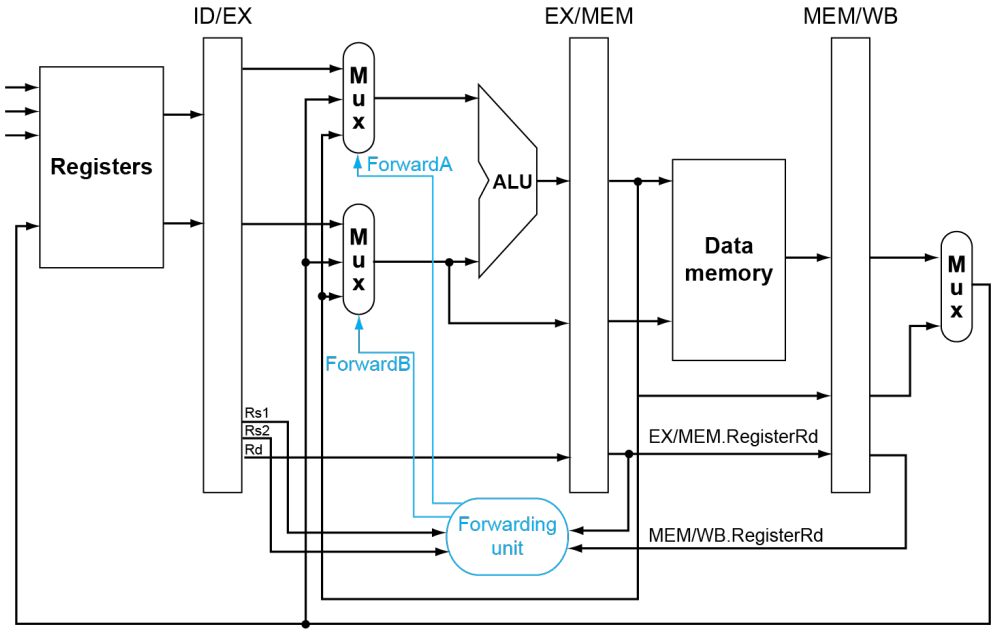


Figure 1

- 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.

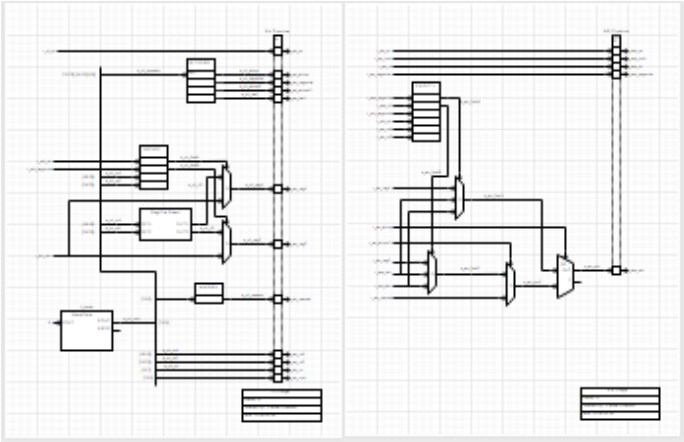


Figure 4

- 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 Phase 6, you will be using the completed project from Phase 5. If you have not completed Phase 5, work with your instructional staff to complete the phase.
- Import the hazard_detection_test project using the following git clone command
  - git clone https://github.com/CompOrg-RISCV/hazard_detection_test.git

- Changes to **the ca_defines.hcodal** file:
  - In the ID section create a new enum (rff_fwd) to control the data forwarding into the decode (ID) stage. You will need two values for register file forwarding (RFF): RFF_REG (register file output) and RFF_WB (MEM/WB pipeline latch value).
  - In the EX section create a new enum (dhf_fwd) to control the data forwarding in the execute (EX) stage. You will need three values for the data hazard forwarding (DHF): DHF_REG (register file output), DHF_ME (EX/MEM pipeline latch value), DHF_WB (MEM/WB pipeline latch value).
  - At the top of the file where the other widths definitions are, create the **two** new widths that can be composed from the two enums you just created (i.e. #define RFF_W bitsizeof(enum rff_fwd))
- Changes to the **ca_resources.codal** file:

- In the ID section create the two new select variables (signals) for the two muxes that will control the forwarding of the data from the register file or the MEM/WB pipeline register. You need one per source operand (s_id_fwd1 an s_id_fwd2). Assign the corresponding variables widths.
- In the ID section create the two variables your will need for the output of the two muxes. You will need to create two new variables (s_id_src1_val and a_id_src2_val) to hold either the forwarded value or the register file output value. Assign the corresponding variable widths.
- In the ID section add two new pipeline registers to hold the names of the input operands (r_id_rs1 and r_id_rs2) to be able to implement the hazard detection logic. Assign the corresponding variable widths.
- In the EX section create two new mux select variables to control the data forwarding for the two ALU input operands (s_ex_fwd1, s_ex_fwd2). Assign the corresponding variable widths.
- In the EX section create two new variables to hold the output of the two muxes (s_ex_fwd1_val and s_ex_fwd2_val). These two variables will be the new inputs to the ALU in the ca_pipe3_ex.codal file. Assign the corresponding variable widths.

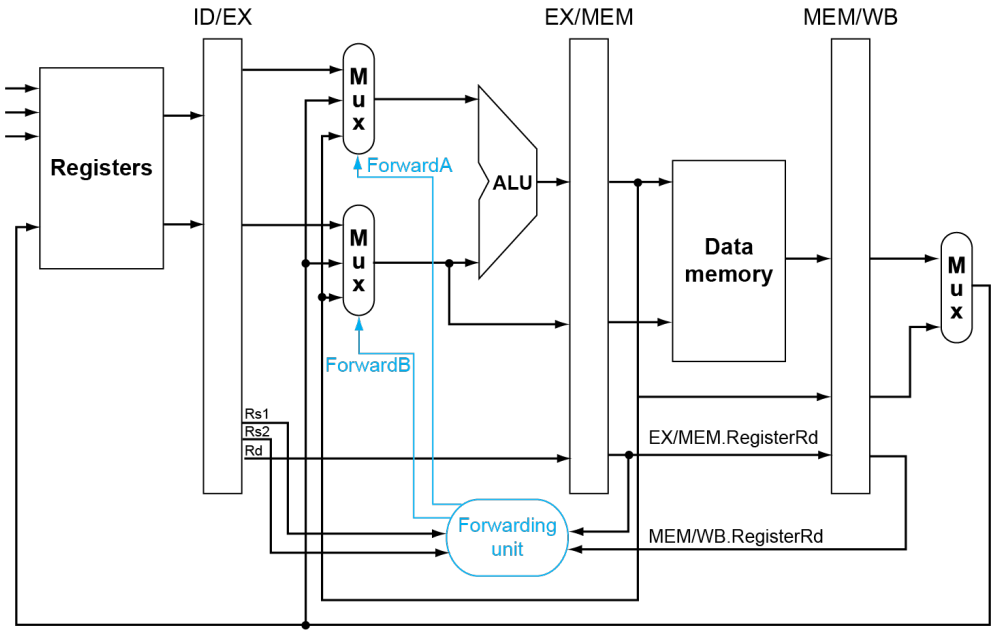
- Changes to the **ca_pipe2_id.codal** file:
  - After reading the register file into the corresponding signals, add the hazard logic detection to identify when the destination register that will be written in the writeback stage in this cycle is the same value that will be read from the register file for the both input operands. The code below is just the beginning of the logic. Complete the rest of the code on your own.

```
If ((r_mem_rd == s_id_rs1) && (r_mem_rd !=0) && r_mem_regwrite == true)){
        (assign the corresponding value for the select variable for the mux)
}
If((r_mem_rd …) …
```

  - Create the 2 switch statements for the 2 muxes that will control the flow of the values. You should have already created all the variables needed in the ca_resources.codal file.
  - Update the pipeline register assignments as needed. You will have to add two need ones and you may need to update two others.
- Changes to the **ca_pipe3_ex.codal** file:
  - At the beginning of the execute stage add the hazard detection logic and assign the mux select variables accordingly. You will need to do this for both r_id_rs1 and r_id_rs2. The code below is the beginning of the code you will have to write. Complete the rest of the code on your own.

```
If ((r_ex_rd == r_id_rs1) && (r_ex_rd !=0) && r_ex_regwrite == true)){
        (assign the select valriable to choose the value from the EX/MEM
        pipeline register)
}
else If((r_mem_rd …) …
```

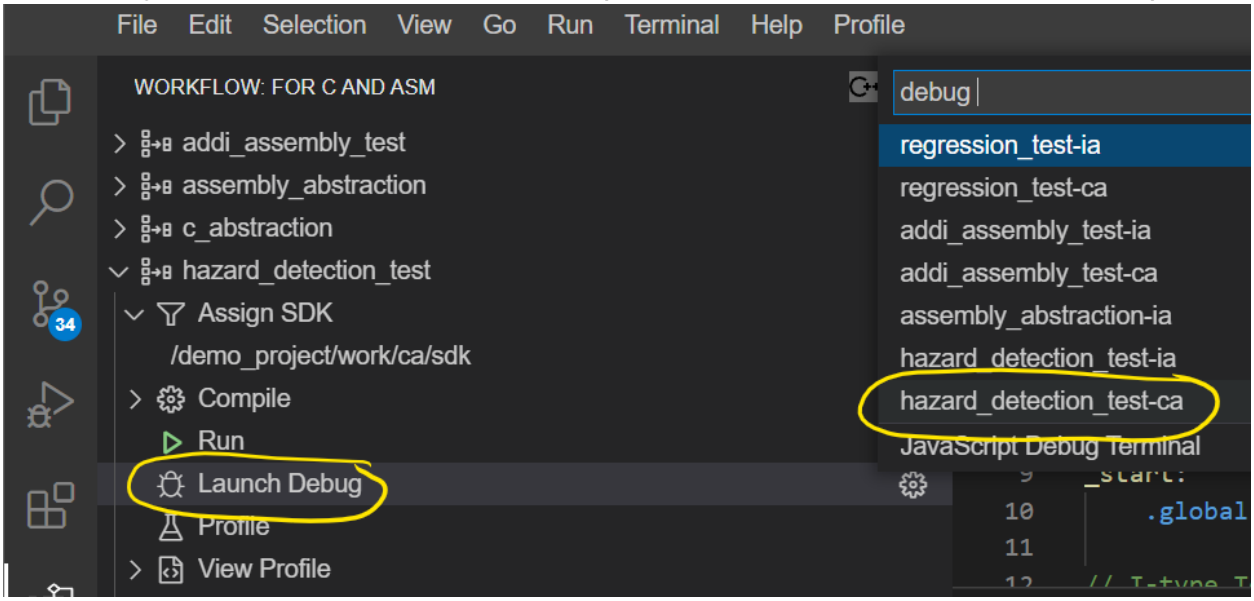
- Before calling the alu_operate function and before the muxes that select the different options for the alu input operands, update the register file values to the forwarded values as needed. To do this you will need to add the 2 switch statements to model the muxes that will control the flow of data (as shown in the schematics below). You should have already created the variables and enums necessary to complete this step and you should have assigned the select variables in the last buller point.
- Update the variable that is assigned to the ouptut of the muxes that select between the multiple options for the alu input operands to the correspoding new variables (these should be the s_ex_fwd1_val instead of r_id_rf_src1 and s_ex_fwd2_val instead of r_id_rf_src2).



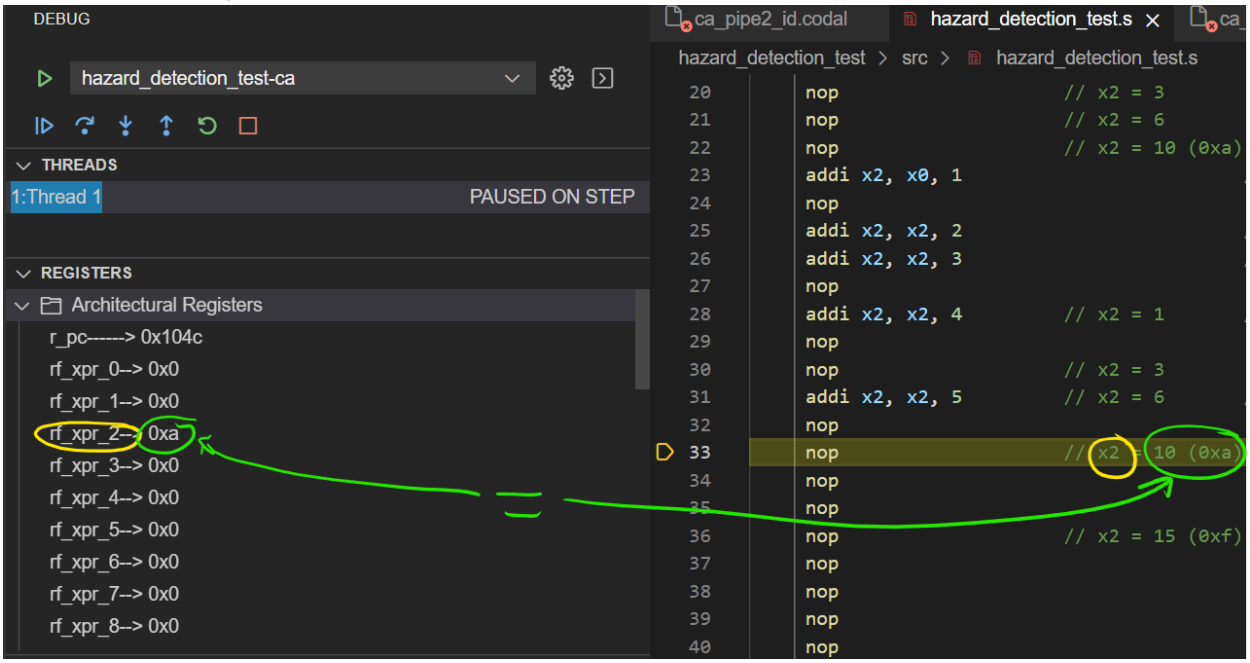
- It is time to try your data hazard and data forwarding
- Preparing your hazard_detecting_test to **Launch Debug(ger)**

1) Generate the SDK for the ia model
2) Generate the SDK for the ca model
3) Assign your processor's project SDK(ia) to the hazard_detection_test software project
4) Set the Compiler Configuration to run an assembly test
5) Set the Debugger Configuration to stop after 0 instructions after startup
6) Compile the hazard_detection_test
7) Assign your processor's project SDK(ca) to the hazard_detection_test
8) Launch Debug to run your CA model

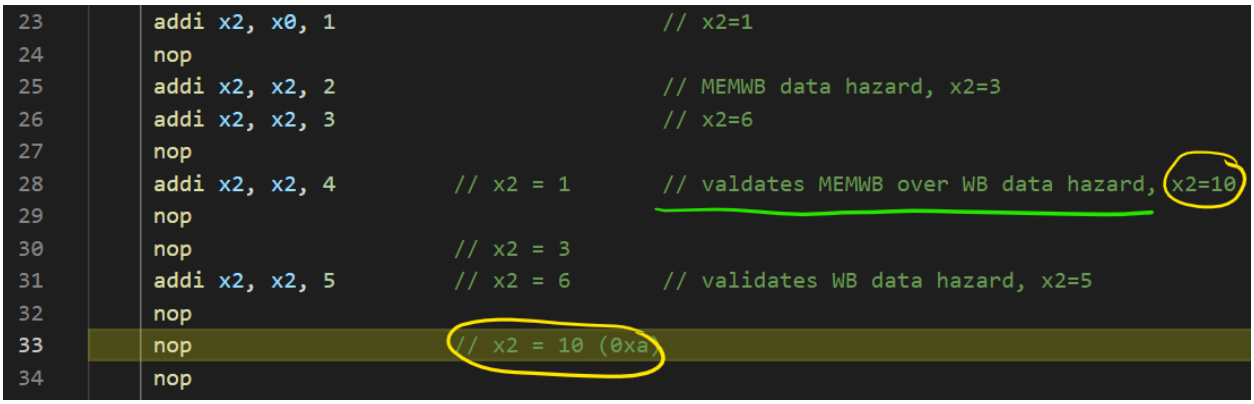
- **Note that if you if you change the code in any of the CA directory files you will only have to repeat number 2 alone (Re-Generate the SDK for the CA model).**



- 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



  - 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



```
23      addi x2, x0, 1                          // x2=1
24      nop
25      addi x2, x2, 2                          // MEMWB data hazard, x2=3
26      addi x2, x2, 3                          // x2=6
27      nop
28      addi x2, x2, 4          // x2 = 1       // valdates MEMWB over WB data hazard, x2=10
29      nop
30      nop                     // x2 = 3
31      addi x2, x2, 5          // x2 = 6       // validates WB data hazard, x2=5
32      nop
33      nop                     // x2 = 10 (0xa)
34      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 phase.

- Completing the phase
    - Clean all binaries from your project.
        - Go to the workflow
        - Under the cu_riscv32i_cycle_accurate project click on Clean.
    - Export your processor phase
        - On the Explorer tab, right click on the cu_riscv32i_cycle_accurate project and select Download
    - Submit your exported tar file into the Phase 6 assignment on Canvas.

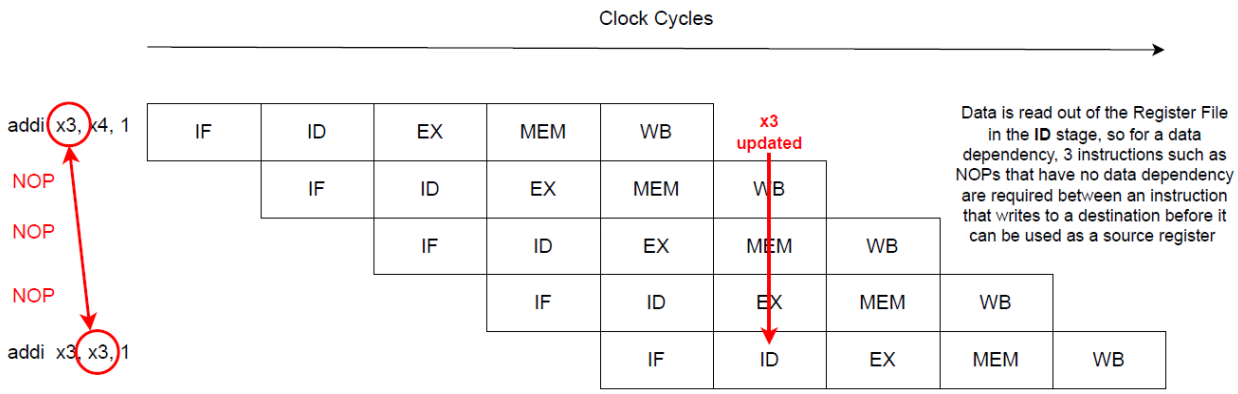
Extra Information If Needed:

- 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 phase 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 phase, 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).

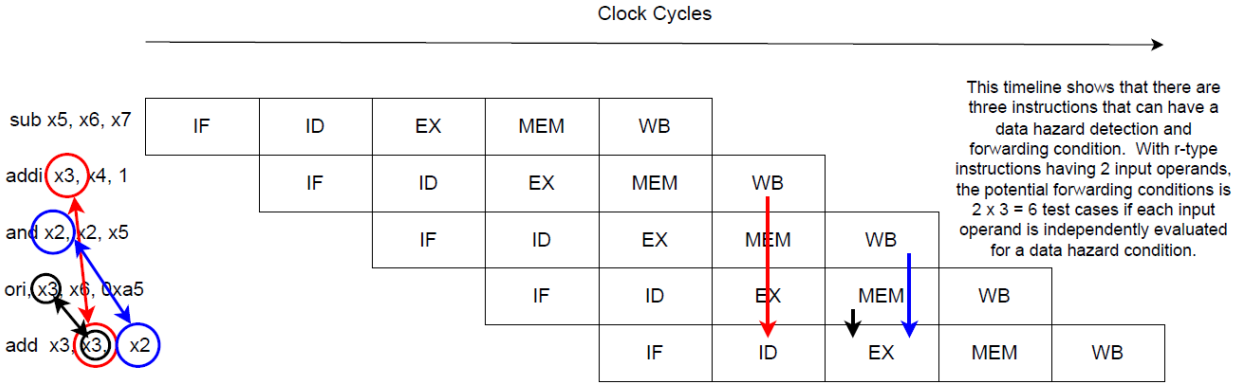
Key Learning Outcomes of this phase:

- 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.



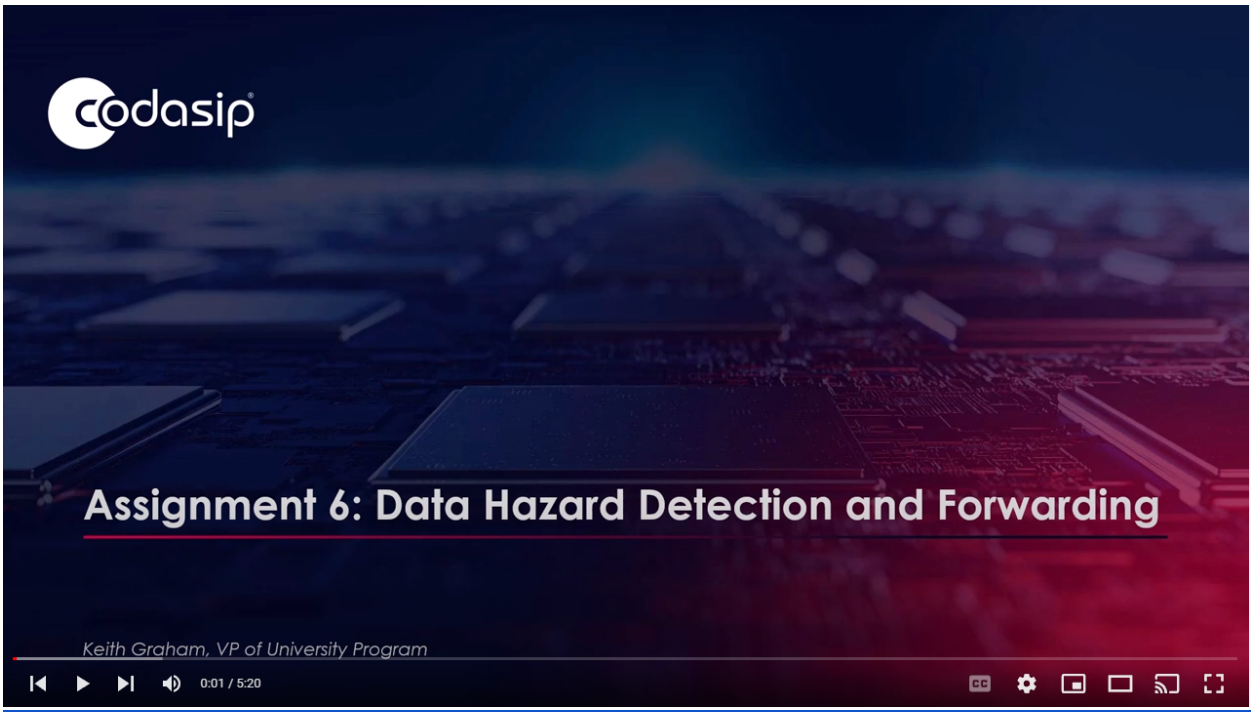
Standard 5-stage pipeline flow with no stalls

- 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.



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

- Click on the below screenshot to be taken to a YouTube video introducing Phase 6



- 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: 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

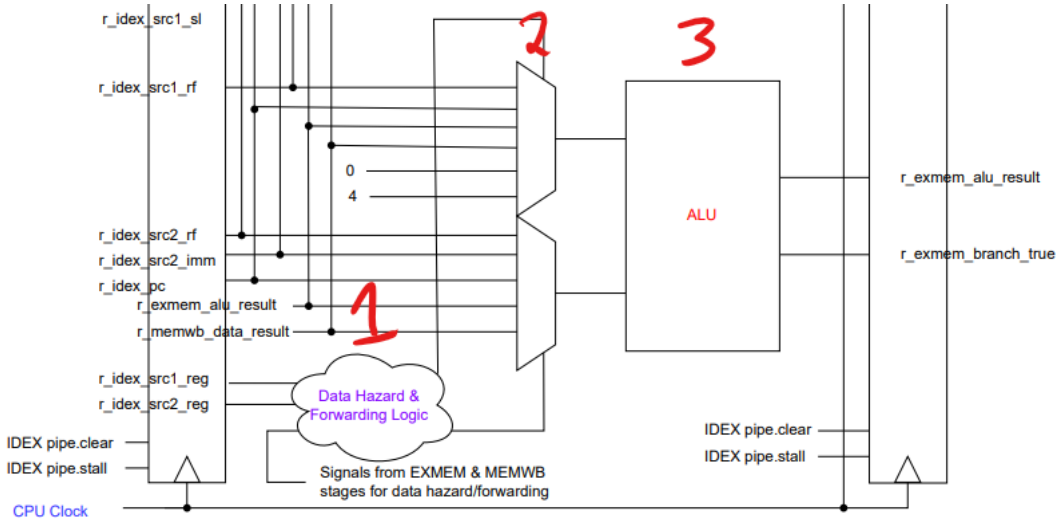
```
#define ALU_SRC1_SEL_W          bitsizeof(enum alu_src1_sel)
#define ALU_SRC2_SEL_W          bitsizeof(enum alu_src2_sel)
```

bitsizeof operator example from ca_defines.codal

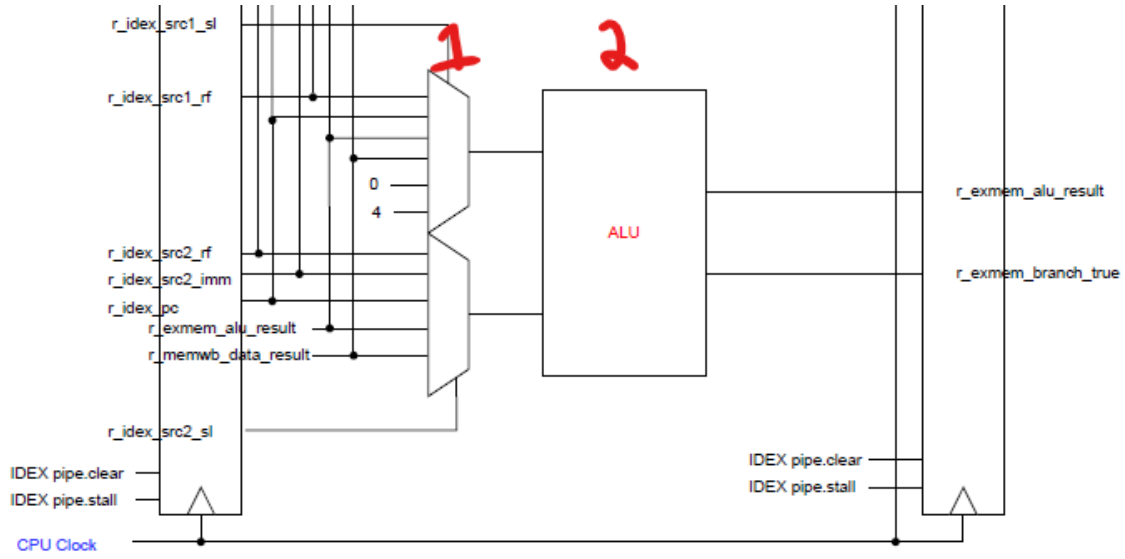
```
// Signals generated by the Instruction Decoder
signal bit[CNTL_BIT]        s_id_regwrite;          // Write to rd (write register) if true
signal bit[ALU_SRC1_SEL_W]  s_id_alusrc1;           // src1 operand mux select line
signal bit[ALU_SRC2_SEL_W]  s_id_alusrc2;           // src2 operand mux select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

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

- To utilize the data forwarding, data hazard detection for both the EX 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.



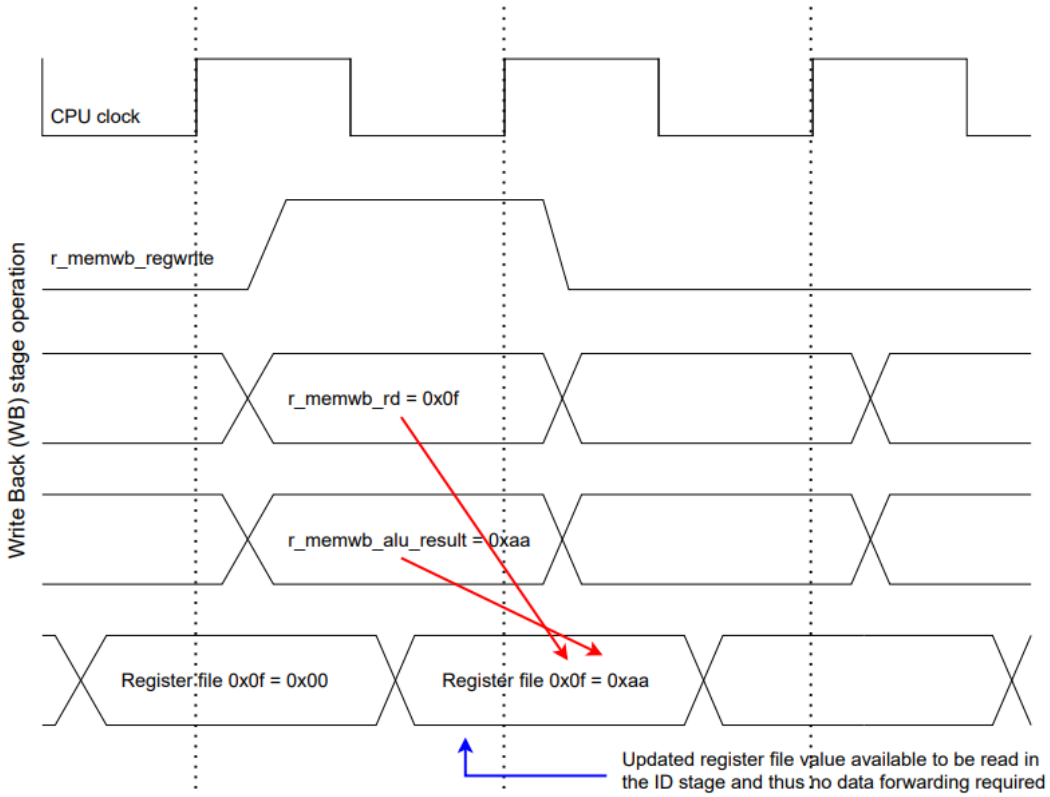
    - . To minimize the total time required in the EX stage, the data hazard detection and forwarding logic for the src1 and src2 select lines will be moved to the ID stage. This is based on the assumption that the ALU timing path will require the longest clock period of all stages due to the complexity 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.



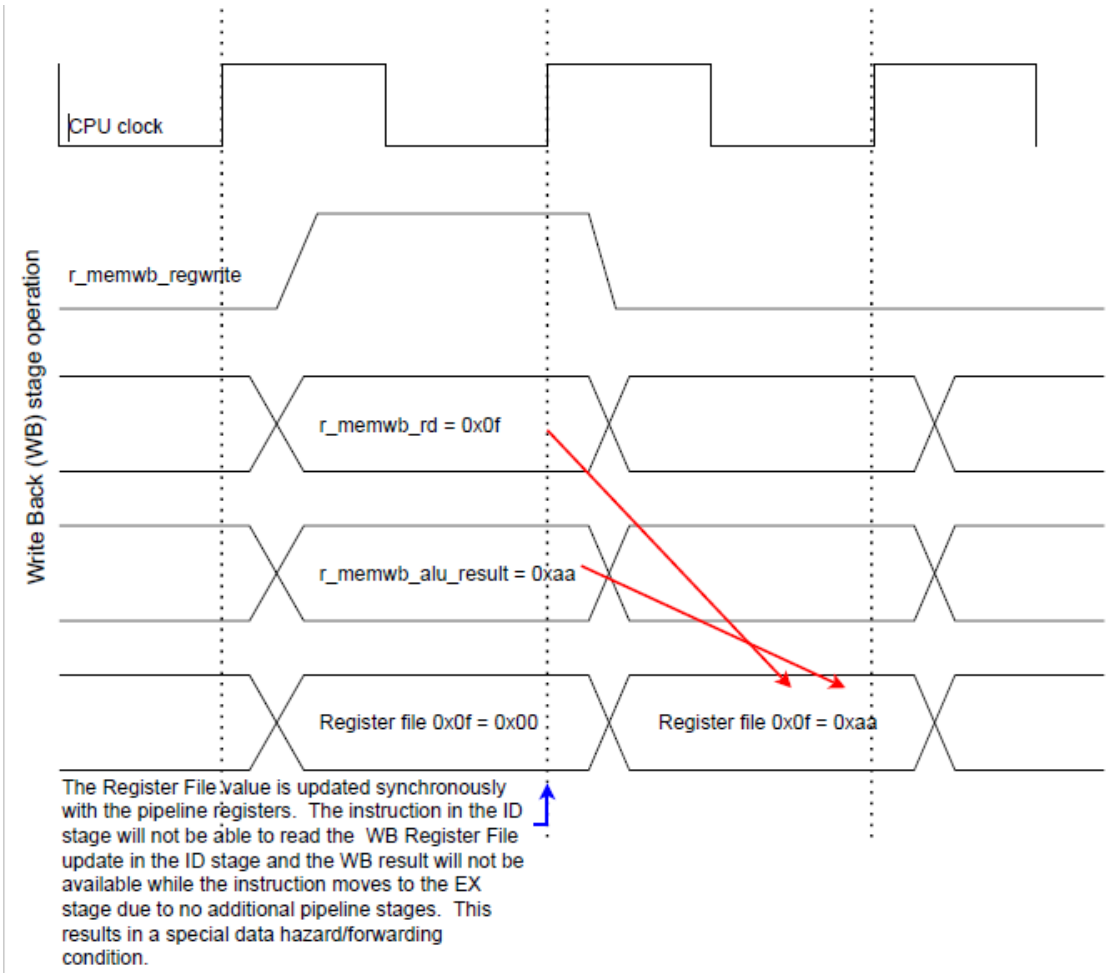
1

- 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.

- 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



- 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.



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

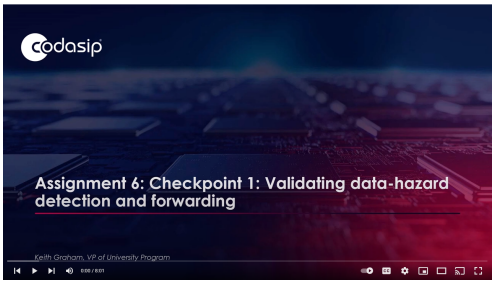
- We now know that all data hazard conditions have be taken accounted for because we have a detection circuit for each of the later pipeline stages and a data forwarding path from each of these later pipeline stages

---

Checkpoint 1: Validating data-hazard detection and forwarding

Phase 6: Checkpoint 1: Validating data-hazard detection and forwarding: Phase 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

-

Appendix A: YouTube videos for Phase 6

Phase Videos:

- **Phase 6: Data Hazard Detection and Forwarding**
  - In pipelining a processor, Data Hazard conditions can exist that can negatively impact the performance of the processor. If an instruction in the pipeline requires the result of a prior instruction that has not made the result available to the instruction through the register file, the processor will need to stall until the data is available for the instruction to proceed. This phase explores detecting these data hazard conditions and instead of negatively impacting the processor performance via a stall, you will be forwarding the result of the prior instruction not through the register file, but through a multiplexer in the Execute or Decode stage.
  - https://www.youtube.com/watch?v=BbuqFgAJM_A&list=PLTUn6Ox9e6q2ienoqI3KClFtRPMqO28uj&index=11&t=14s
- **Phase 6: Checkpoint 1: Validating data-hazard detection and forwarding**
  - Phase 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 phase, 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