Computer Organization
5-stage RISCV32I Processor
Phase 7: Branch and Jump Instructions
Fall 2022

Due Dates:

- This is a 1-week phase
- Due date: Monday, November 14th, 2022 at 11:59pm

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

- Update the diagrams.net schematics from phase 6 to include all the components necessary for the branch and jump instructions.
- Update all the specified Codasip files to implement all branch and jump (jal and jalr) instructions. The files that we will be modifying in this phase are ca_decoder.codal, ca_defines.hcodal, ca_resources.codal, ca_pipe_stage2_id.codal, ca_pipe_stage3_ex.codal, ca_pipe_stage4_me.codal and ca_pipe_control hcodal.

Schematic Instructions:

- ♦ Copy the schematic standardname6.xml to standardname7.xml.
- ♦ Open the schematic in diagrams.net.
- ♦ Adding the conditional branch instructions follows the logic shown in Figure 4.49 of the textbook, reproduced below as Figure 1. The functions in the three red rectangles must be added.
- ♦ In the EX stage, add the Adder component which adds the PC value to the correct immediate value. The "Shift Left 1" function in Figure 1 is not necessary, as this function is handled in the IMMGEN logic. The output of the adder (s_ex_target_address) will be used below. Add the zero signal coming out of the ALU.
- ♦ The JAL instruction jumps to the address which is a J-type immediate added to the current PC. This path already exists in the schematic for the branch instructions, although IMMGEN will create a different immediate value based on s_id_immsel. No schematic changes are necessary for this piece of the instruction.
- ♦ The JALR instruction jumps to the address which is the sum of a register and an I-type immediate. The ALU will be used in this case to compute the branch target address. The ALU already has the correct inputs for this operation to work as expected, no additions are needed on the schematic for this piece of the instruction.
- ♦ Add a 2-1 multiplexor in the EX stage which selects between the output of the address adder and the output of the ALU and produces the final s_ex_target_address, which is pipelined to the ME stage. You will use the signal s_id_jump_instr, produced by the decoder, to control this mux and pipeline it to the EX stage.
- ♦ The jump logic is not included in any of the block diagrams in the textbook because the mini RISC-V processor of the book doesn't have these instructions.

- ♦ Both JAL and JALR write the current PC value plus 4 into the destination register of the instruction. Implementing this requires a multiplexor selecting the write data to the Register File but this needs to be done in the ALU to make sure that forwarding works as expected.
  - Add a 2-1 Multiplexor component in the EX stage after the ALU. One input is the current ALU output (s_ex_alu_result). Add an Adder component which adds 4 to the current PC value (r_id_pc) and connect it to the second input of the 2-1 mux (this addition is for the jump instructions).
  - Add a new Decoder output s_id_rfwtsel to control the write data select. Pipeline this signal to the 2-1 mux select.
- ♦ Logic in the ME stage determines whether the conditional branch is taken or not. The zero output from the ALU (s_ex_zero) is asserted when the ALU output s_ex_alu is zero. This signal needs to be pipelined to the ME stage, and is the only signal required to determine whether a conditional branch is taken or not.
- ♦ A new control signal s_id_branchop is required from the Decoder in the ID stage to determine how the zero indicator is used. Pipeline this signal to the ME stage. Add the control block BRNCH which creates the s_me_take_branch signal which selects the next PC source, based on branchop and zero.

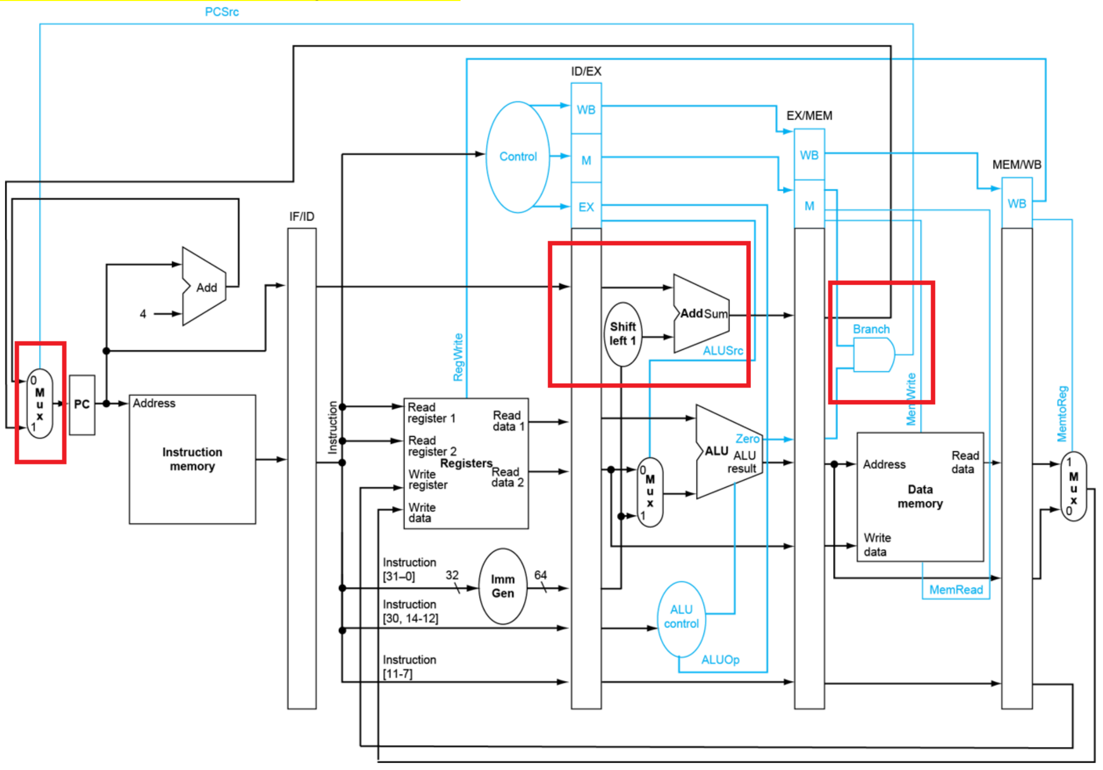


Figure 1

- ♦ Although not required, it is recommended that you submit the updated schematic as soon as it is complete, as standardname7.xml. This will allow for feedback prior to creating the Codasip code. The scoring will reward early submission of the schematic.

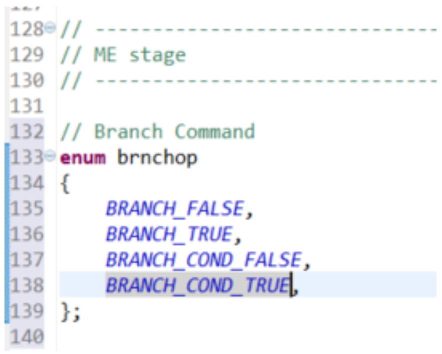
Instructions for Codasip:

1) You will continue to build upon your phase 5 project. If required, import your phase 5 processor CodAL project.
2) The following branch and jump instructions will be implemented: BEQ, BNE, BLT, BGE, BLTU, BGEU, JAL, JALR.
3) Changes to the **ca_defines.codal** file
    - Add the enum values and width defines for the new control signal, s_id_brnchop,, as shown in the figure below. The four values required are as follows:
      BRNCH_FALSE – never branch (which must be the default and thus first in the list)

      BRNCH_TRUE – always branch (used for jump instructions)

      BRNCH_COND_TRUE – branch if the ALU output is zero (for conditional branches)

      BRNCH_COND_FALSE – branch if the ALU output is not zero (for conditional branches)



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

// Branch Command
enum brnchop
{
    BRANCH_FALSE,
    BRANCH_TRUE,
    BRANCH_COND_FALSE,
    BRANCH_COND_TRUE,
};
```

    - Create the enum and width defines for the EX stage multiplexor that selects which value to write to the register file (r_ex_rfwtsel).
        - WB_ALU: selects the output of the alu to write into the register file
        - WB_PC: selects the current pc + 4 value to write into the register file (this is for the JAL and JALR instructions)
    - The branch/jump target address unit must be updated to provide the jump address calculated address for jump instructions
        - Create an enum and corresponding width definition in the EX section for the EX stage mux that selects how to calculate the branch target address. The mux options are:
            - BRADD_ADDR: calculate the branch target address by adding the immediate to the PC (all branch instructions and the JAL)
            - BRADD_ALU: get the branch target address from the output of the ALU (JALR)

- Changes to **ca_resources.codal:**
    - Add definitions for the branch operation (both signals and registers) to pipeline this value all the way through to the memory stage.

    - Add the definition for the ALU zero output (both signal and registers) to pipeline the value all the way through to the memory stage. s_ex_zero is a 1-bit control signal and can be defined with BOOLEAN_BIT.
    - Add the signal definition to keep the branch target address. The corresponding pipeline register has already been provided.
    - Add the definition for the control signal for the register file write select value (create both signals and registers to be able to pipeline this value from the decode to the execute stage.
    - Add a new signal in the execute stage to hold the value that will be written into the register file. Note that this will be different from the existing alu_result value since jal and jalr do not use the alu_result value to write to the register file and instead use the PC+4.
- Changes to **ca_pipe2_id.codal**
    - Pipeline the new control signals from the decoder
- Changes to **ca_pipe3_ex.codal**
    - Inside the alu_operate event, after the aluop switch statement, add the multiplexor which selects among the branch target address calculation choices based on the control value for r_ex_jump_inst. Inside each case statement add the corresponding logic to compute the address, s_ex_target_address, for both branches and jumps accordingly. If r_ex_jump_instr is set then the target address is taken from the ALU output, otherwise add the immediate value to the current pc.
    - Inside the alu_operate event, after the aluop switch statement, add the generation of s_ex_zero, which is asserted when the ALU output s_ex_alu is zero.
    - Inside the alu_operate event, after the aluop switch statement, add a mux that selects between the alu_result and the PC+ 4 to pass down to write into the register file. You can save this value into a new signal value that will be pipelined to the existing r_ex_alu_result pipeline register.
    - Pipeline all required values in ex_output.
- Changes to **ca_pipe4_me.codal**
    - Create a new event, branch_logic, and ensure that you declare this event (as done with the me_output event) inside the me event and **call the event** before me_output()

```
event branch_logic : pipeline(pipe.ME)
{
}
```

    - A branch will occur based on the following logic:
        - r_ex_branchop == BRANCH_COND_TRUE, then if r_ex_zero == 1
        - r_ex_branchop == BRANCH_COND_FALSE, then if r_ex_zero == 0
        - r_ex_branchop == BRANCH_TRUE
    - Based on the logic above, assign the corresponding value to the signal s_me_take_branch. The result will depend no only on the value of the r_ex_branchop but also the value of the r_ex_zero register.
        - You can use a switch statement is follows:

```
switch(r_ex_branchop)
        {
            case BRANCH_COND_TRUE:
s_me_take_branch = r_ex_zero;
                break;
```

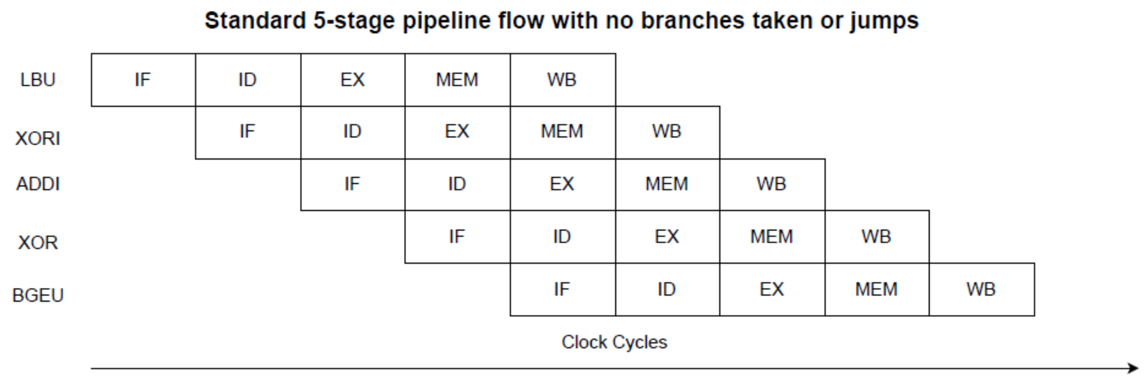
- Pipeline all required values in me_output.

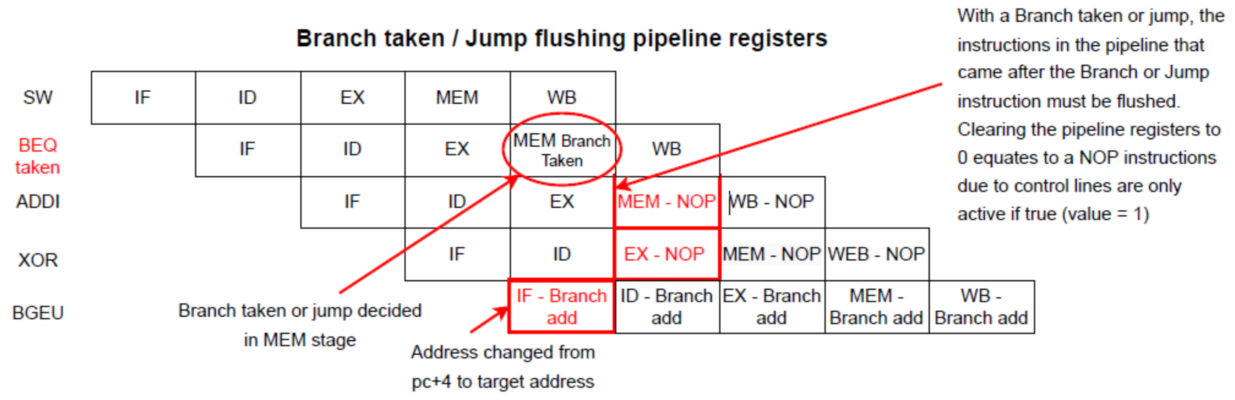
4) Changes to **ca_pipe5_wb.codal:**
    - No changes required for this phase.

5) Changes to **ca_decoder.codal**:
    - Add the assignment of any new control signals (s_id_brnchop, s_id_rfwtsl) to every existing instruction group. Since none of the existing groups ever branch, assign BRANCH_FALSE for s_id_brnchop in each case.
    - Search for btype until you find the element i_hw_btype_branches and uncomment the complete i_hw_btype_branches element
    - Set all control signals:
        - s_id_regwrite: branch instructions do not write to the register file
        - s_id_alusrc1 and s_id_alusrc2: branch instructions use registers for both inputs.
        - s_id_imm_gen_sel: use the corresponding branch immediate value.
        - s_id_branch_inst: this signal indicates if we need to use the PC to calculate the target address, so set it to true for all branch instructions.
        - s_id_jump_inst: this signal indicates if we need to use a register to compute the target address. Set this value to false for all branch instructions.
        - s_id_mem_ops: since these are not memory instructions set this value to MEM_NOP as with all the prior elements.
        - s_id_memread: since these instructions are not memory instructions set this value to false.
        - s_id_halt: set this value to false.
        - s_id_rfwrtsl: since write enable is false we don't care about this value for branches (DONT_CARE)
        - s_id_branchop: assign the correspoding branch operation based on the following logic:
            - BEQ: branch taken if the result of the ALU is zero.
            - BNE: branch taken if the result of the ALU is not zero.
            - BLT/BLTU: branch taken if the result of the ALU is not zero.
            - BGE/BGEU: branch taken if the result of the ALU is zero
    - the i_hw_jtype_jlink and i_hw_itype_jlreg elements
        - First, you will need to **uncomment** these element frameworks
        - For these elements, s_id_branchop will be set to BRANCH_TRUE
        - For the **jal** instruction, set s_id_branch_inst = true and the s_id_jump_instr = false since the target address will be calculated in a similar manner as the branch instructions (PC+IMMED).

        - For the **jalr** instruction, set s_id_branch_inst = false and the s_id_jump_instr = true to indicate that the target address will come from the register file plus the immed (SRC1+IMMED).
        - For the aluop control signal, set it to ALU_ADD since we are going to be adding the two values for jalr. The jal instruction won't be using the ALU so this signal's value does not matter.
        - You will need to assign a value to the remaining control signals defined in the instruction decoder
    - Add alll of these elements to the inst_decode set at the top of ca_decoder.codal (line 36)

6) All the updates to the model have been to determine whether a branch should occur based on an ALU compare operation and calculating the target branch address, changing the program flow. The changes to the IF stage have already been provided.
7) The last piece to implement is the flushing of pipeline stages in case of a branch misprediction (we are doing static branch prediction to always predict not taken), and it is performed in the ca_pipe_control.codal file
    - **Changes to the ca_pipe_control.codal file:**
    - Inside the pipeline_control event, add an IF statement based on the signal that that identifies when a branch has been taken. (s_me_take_branch).


Standard 5-stage pipeline flow with no branches taken or jumps
LBU
XORI
ADDI
XOR
BGEU
IF
ID
EX
MEM
WB
Clock Cycles


Branch taken / Jump flushing pipeline registers
SW
BEQ taken
ADDI
XOR
BGEU
MEM Branch Taken
MEM - NOP
WB - NOP
EX - NOP
MEM - NOP
WEB - NOP
IF - Branch add
ID - Branch add
EX - Branch add
MEM - Branch add
WB - Branch add
Branch taken or jump decided in MEM stage
Address changed from pc+4 to target address
With a Branch taken or jump, the instructions in the pipeline that came after the Branch or Jump instruction must be flushed. Clearing the pipeline registers to 0 equates to a NOP instructions due to control lines are only active if true (value = 1)

        - If true:
            - pipe.ID.clear(): This command sets the clear input signal to the ID pipeline register as true for this clock cycle.

                Transforming the instruction in the ID pipeline stage to a NOP operation.
            - pipe.EX.clear(): This command sets the clear input signal to the EX pipeline register as true for this clock cycle. Transforming the instruction in the EX pipeline stage to a NOP operation.
        - else false:
            - No requirement to disable the pipeline clear. CodAL disables clear input to the pipeline register any clock cycle that it is not actively specified to be clear
        - pipe.MEM.clear() is not included in the flushing of instructions because the instruction in the MEM, memory, stage is not an earlier instruction than the branch or jump. It is actually the branch or jump instruction.
    - The pipeline control is now complete for the branch operations

---

Validating branch operations

- Import the control_inst_assembly_test into your Codasip workspace
    - git clone https://github.com/CompOrg-RISCV/control_inst_assembly_test.git
    - After building both the IA and CA SDK, right click the Assign SDK for this project in the workflow perspective and assign your project(ia) model
- Set the standard assembly **Compiler Configuration** to:
    - No Startup or Default Lib (-nostdlib)
- After a successful **compile**, assign the CA SDK to the control_inst_assembly_test
- **NOTE**: If you will need to recompile the test after assigning the CA model to your project, you will first need to reassign the IA model to the software project
- Set the standard assembly **Debug Configuration** to Stop at Startup after 0 instructions and set a breakpoint on the second instruction of the test source file.
- **Launch** the debugger for control_inst_assembly_test
- Debug control_inst_assembly_test
    - Place breakpoints on the following lines:
        - 35 and 161: If the program reaches these breakpoints/halts, the branch operations have failed
        - 52: If the program reaches this breakpoint, the branche BEQ appear to be working
            - Confirm successful branches by comparing your register results with the program's comment statements
    - Step through the program
        - If you reach the halt after the FAIL label, a branch occurred when it should not have
            - Time to debug
        - If you reach the halt after PASS2 label, your branch, BEQ, successfully completed

- Verify that your NOP bubbles were correctly implemented
    - x3 = 1, a branch not taken did not create a NOP bubble
    - x4 = 0, EX stage NOP bubble occurred as required
    - x5 = 0, ID stage NOP bubble occurred as required

- For validating the jal and jalr instructions, you will continue to use the control_inst_assembly_test project.
- Comment out the halt instruction after you branched to the PASS2 label, line 52. This will enable the code to reach the jump test code
    - You will need to reassign your IA project to the control_inst_assembly_test to compile the updated test with the halt statement on line 52 commented out
    - No need to reassign the project to CA after the compile. Once the software project is aware of the CA model, it does not need to be reassigned
- Place breakpoints at:
    - 35, 104, 124, 161: These halt statements will indicate a jump or branch failure
    - 86: Reaching this breakpoint with register x10 = 1 indicates that the branches and jumps in this routine successfully executed
- Step through the code to see that the jump operations performed as programmed. Refer to the comments to understand the flow of the code and whether your project successfully completed the test

- For the last validation step, you will use your regression test.

- Comment out the **halt** statements after the immediate instructions, the r-type instructions, shift-immediates, and the data-hazard detection / forwarding test code
- Your code should now run up to and through your regression test branch and jump test code and halting before the load and store set of instructions
- Debug your regression test and step through your code until it fails or it reaches the halt statement after your branch and jump test sequences
    - If your test fails an earlier test sequence that had passed, it most likely is an error in your processor's data-path or control-signals since these tests had passed in an earlier phase
    - If your test fails within your branch and jump test sequence, you will need to evaluate whether the failure is in your regression test sequence and/or in your branch/jump CodAL implementation
- After your regression test passes all the way to the halt after your branch and jump sequence with correct results through all the tests, you are ready to submit your project for grading

---

- Complete the phase
    - Download your ca folder only
    - Rename the file to the standardname7

  - Submit both your xml schematics and your compressed file into the Canvas assignment for phase 7.

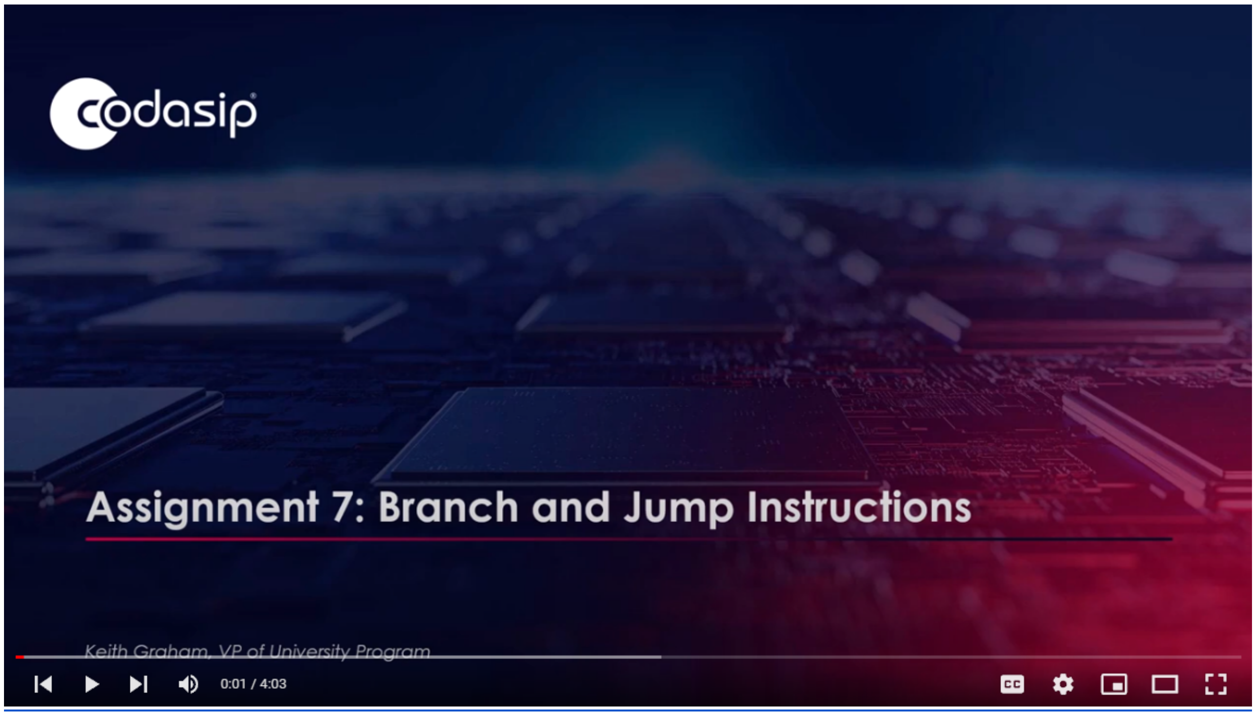
Objective: For a program to make decisions such as an IF then Else or to perform a loop, a set of control instructions need to be implemented. These control instructions are broken into branch instructions which are conditional operations while jumps are unconditional operations. A common use of branch instructions are in IF statements to perform a set of instructions based on a condition or to jump back to the top of a loop until a condition is met. The jump instructions perform the same operation as a branch operation in that they change the flow of the program but they are not conditional, it is an unconditional branch operation. In this phase, you will implement both the branch and jump instructions for a RISCV32I processor. Since these instructions change the flow of the program and require the Execute (EX) stage, any instruction in the processor's pipelines that entered after these instructions must be flushed if a branch or jump is taken. This is because they would not have entered the processor pipeline if the change of program flow could be performed immediately after the control instruction entered the processor.

For this phase, additional ALU ops (operations) will need to be added to the ALU to decide whether a branch should be taken. In the EX stage, the branch and jump target addresses must be calculated by implementing a specific ADD execution unit to calculate these addresses. There are jump instructions that use a register to determine its target address which creates the need to perform a data hazard detection and data forwarding for this branch/jump ADD unit as you did for the standard ALU operands. If a change of flow of instructions does occur, a branch is taken or a jump, all the instructions in the earlier stages of the pipeline must be flushed (become NOPs) since the change of flow results in these instructions not being taken.

Key Learning Outcomes of this phase:

- Control Instruction (Branch & Jump): Without control operations, programs would only be able to perform a single pass through the program, going from the first line of code straight to the end. Control operations change the flow of a program based on either a test condition such as whether a variable is true or false, or an unconditional jump such as making a function call. In a single-cycle processor, no pipeline, the next instruction would either be the Program Counter + 4 or the new branch/jump address. For pipeline processors, there are opportunities for **Control Hazards**. A **Control Hazard** occurs when the instructions in the pipelines are not the instructions to be executed. For a pipeline processor that is performing parallel instructions, one in each stage, if the branch/jump is taken, the instruction in the Instruction Fetch (IF) stage would be the branch/jump target address, but the pipeline would have already started the processing of the Program Counter + 4 instruction. With the control operation's change of address, the Program Counter + 4 and any other instruction that has entered the pipeline would need to be flushed (a NOP bubble) since they would not have occurred in a single cycle processor model.

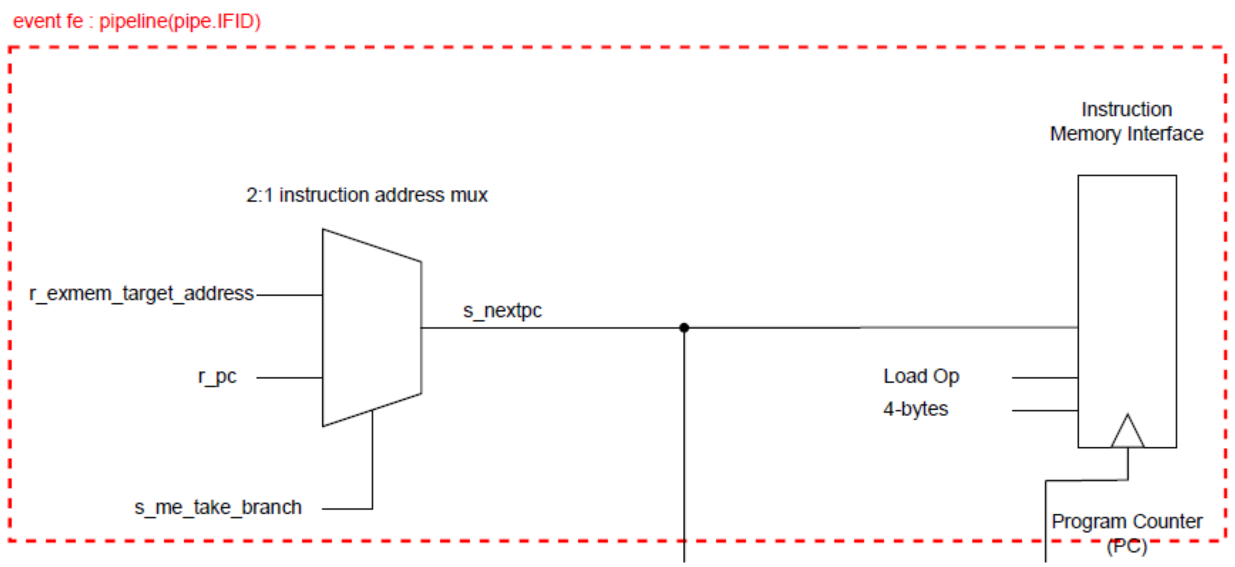
- Click on the below screenshot to be taken to a YouTube video introducing Phase 7


codasip
Assignment 7: Branch and Jump Instructions
Keith Graham, VP of University Program
0:01 / 4:03

- The decision of what each stage of a pipeline must do or operate on to complete the instruction is done in the ID stage within the instruction decoder. This results in the decoder turning on the required portions of the processor and disabling others. Turning-on or disabling these functions are done by the processor's control bits. These control bits are passed along to the next pipeline state until they reach their respective function.
    - For example, the regwrite, register write, control bit is passed to the EX, MEM, and then to the Write Back (WB) stage so that in the WB stage, a decision can be made whether to update the Register File
- With the parsing of the instruction completed in phase 5 and the data hazard detection and data forwarding completed in phase 6, the branch instructions have been added, parsed, and are ready to proceed to the EX stage
- Should the decision of taking a branch, branch_instr & branch_true, be decided in the EX or MEM pipeline stage? From the previous phase, the assumption is that the EX pipeline stage will define the clock period of the processor due to the complexity of the ALU processing element. Adding any additional logic before or after the ALU will increase the overall processor's clock period resulting in a lower clock frequency for all operations. To maximize the performance of the majority of operation, the 3rd logic block in the below block diagram, Branch Logic, will be moved to the MEM pipeline stage. In the branch timing path, there is actually a 4th element which is located in the Instruction Fetch (IF) stage. This additional element takes the control signal branch_taken to either request the next instruction pc + 4 or the branch calculated address, a 2:1 mux.

- To change the program flow, the next instruction address to fetch must be updated to the branch target address if a branch is to be taken. The current Instruction Fetch (IF) stage only requests the previous Program Counter (PC) + 4. With the branch and jump control instructions, the next instruction address to fetch must be selected using a 2:1 mux
    - Inputs to the mux
        - r_pc: previous PC address + 4
        - r_ex_target_address: Address if a branch or jump is taken
    - Mux output
        - s_if_nextpc: address used to request the next instruction memory fetch
    - Mux select line(s):
        - The signal that you have assigned in your ca_pipe4_me.codal branch_operation event that evaluates whether a branch should be taken
            - If false, no branch and use r_pc
            - If true, branch/jump taken and use r_ex_target_address

Instruction Fetch (IF) Stage Block Diagram


event fe : pipeline(pipe.IFID)
Instruction Memory Interface
2:1 instruction address mux
r_exmem_target_address
r_pc
s_nextpc
Load Op
4-bytes
s_me_take_branch
Program Counter (PC)

- This mux has been provided in the base project. It is implement using the following if/else statement:

```
// If branch, the next PC is not the previous pc + 4, but the calculate branch or jump address
if (s_me_take_branch) s_if_nextpc = r_exmem_target_address;
else s_if_nextpc = r_pc;
```

- With this mux already implemented, no changes to the IF, Instruction Fetch, stage is required

- In a single stage processor, all instructions complete in one cycle (or single stage) but performance is reduced due to lack of parallelism. If a branch is taken, the instructions immediately following the branch instruction will not be executed. In a pipeline processor whose performance is increased by executing multiple instructions simultaneously, one in each stage of the pipeline, when a branch is taken, there are instructions that need to be flushed (become NOPs) since they would not have been executed in a single stage design, a **Control Hazard**.

- The branch or jump is taken in the MEM stage which implies there are instructions in the Instruction Decode (ID) and Execute (EX) stage that must be flushed, to become NOP operations.
  - A flush occurs by "clearing" all the pipeline registers whose instruction in them would not have been executed in a single stage processor.
  - With the control signals defined as true, 1, to assert their control function such as a branch to be taken or to write to the register file, clearing these control signals, making them 0, disables their operation by setting them to false
  - The pipeline registers are defined to have a clear control input in addition to the clock signal
    - If the clear input is true, instead of latching in the input signals upon the CPU positive clock edge into the register file, all the registers effectively latch in 0s. The instruction that moves to the next stage is replaced with all zeros or equivalent to a NOP operation
- The Instruction Fetch (IF) stage does not need to be flushed because the 2:1 mux that you have added is effectively changing the IF stage's instruction request to the branch address

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.

Appendix A: YouTube videos for Phase 7

Phase Videos:

- **Phase 7: Branch and Jump Instructions**
  - We need our software programs to make decisions based on input that could be a stimulus from a touchscreen or based on a particular data set. These decisions occur at the assembly programming level through branch instructions. Branches are conditional changes of program flow. Jumps on the other hand are not conditional changes of program flow. They are used by programs to enable programming best

practices by supporting modularity and encapsulations through the use of function calls and their associated return.

    These instructions are very useful instructions, but when program flow changes in a pipeline processor, it introduces a control hazard. In this phase, you will learn how to properly change program flow as well as handle the associated control hazard.
    - https://www.youtube.com/watch?v=OWc-RzXpy38&list=PLTUn6Ox9e6q2ienoqI3KClFtRPMqO28uj&index=14
- **Phase 7: Checkpoint 1: Validating branch instructions**
    - To validate a branch instruction, you need to validate its conditional statement, whether it branches to the target address correctly (both forward and backwards), and whether it correctly handles the associated Control Hazard.

      This checkpoint will be used to validate that one branch operation, beq, branches correctly and handles its associated Control Hazard. If an error is indicated, the video comments will help indicate what the failure could be such as a lack of NOP bubbles inserted in the ID and/or EX pipeline registers.
    - https://www.youtube.com/watch?v=YJbDHQDGKQ4&list=PLTUn6Ox9e6q2ienoqI3KClFtRPMqO28uj&index=15

Frequently Asked Questions (FAQs) Videos

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