**BEQ, BNE, J MIPS Operations**

**CSC342/43**

**Professor Izidor Gertner**

**Anthony Ramos**

**April 18th 2021**

**Contents**

[**Objective** 3](#_Toc69425301)

[**Introduction** 4](#_Toc69425302)

[**Components** 5](#_Toc69425303)

[*Instruction Register* 5](#_Toc69425304)

[*Register File* 6](#_Toc69425305)

[*32-Bit Comparator* 7](#_Toc69425306)

[*Program Counter* 8](#_Toc69425307)

[*Sign Extender* 8](#_Toc69425308)

[*Adder* 9](#_Toc69425309)

[*2:1 Multiplexer* 10](#_Toc69425310)

[*Components Package* 11](#_Toc69425311)

[*NAL Unit* 12](#_Toc69425312)

[**BEQ Simulation** 12](#_Toc69425313)

[*Verification 1: Not Equal Scenario* 13](#_Toc69425314)

[*Verification 2: Equal Scenario* 13](#_Toc69425315)

[*Verification 3: Equal Scenario* 13](#_Toc69425316)

[**BNE Simulations** 15](#_Toc69425317)

[*Verification 1: Equal Scenario* 15](#_Toc69425318)

[*Verification 2: Not Equal Scenario* 15](#_Toc69425319)

[*Verification 3: Not Equal Scenario* 15](#_Toc69425320)

**Objective**

The objective of this lab is to apply the knowledge gained from previous self-check labs (SCL) regarding comparators, multiplexers, and registers to implement the data paths for MIPS instructions beq, bne, and j. Namely, by constructing a Next Address Logic (NAL) Unit that will determine the next address of the instruction (i.e. new program counter value) to be executed whether or not a branching occurs. Figure 2 below shows the design to be implemented.

# **Introduction**

Prior to designing anything, we must first understand how the *I-Type* MIPS 32-Bit IR works. Figure 1 shows this breakdown. The first 6-bits are the opcode field which define what arithmetic operation will be performed. Bits 25 down to 16 define the address indicies RS and RT which will store the data we wish to perform an arithmetic operation on. Lastly, bits 15 down to 0, define the immediate value. For the purposes of this lab, we do not need to know the value of the Opcode since there is no instruction that will be performed.

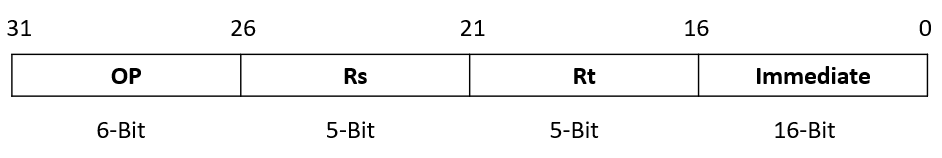


Figure 1: I-Type Instruction Syntax

The data path below makes use of various components (to be discussed in detail below) to compute the address of the next instruction. The required inputs for this unit is a 32-bit instruction, a program counter (PC) value, and clock signal. For the purposes of this lab, the instruction and PC values are arbitrary. The 32-bit instruction is fed into the Instruction Register (IR) component where the address indices Rs and Rt, along with the Opcode and 16-Bit Immediate field are decoded. The Rs and Rt address indices serve as inputs to the Register File component which use the address indices to locate the 32-Bit data stored at those addresses. The data stored at these address indices are also arbitrary and are fed to outputs BusA and BusB. These outputs in turn, are fed to a comparator to set a flag called EqualCond. That is, if BusA = BusB, then the EqualCond flag is set to 1. The significance of this value depends on the branching instruction we are implementing and will be discussed later. With this flag now set, we now have to compute the two possible addresses using the PC value and signed extended immediate value. Using two adder components will give these addresses. Lastly, the addresses are fed into a 2:1 multiplexer where the CondEqual flag set earlier serves as the selector to choose the appropriate address of the next instruction (i.e. new PC value). This (final) value of the PC is then fed back into the PC component and updated to this new value.

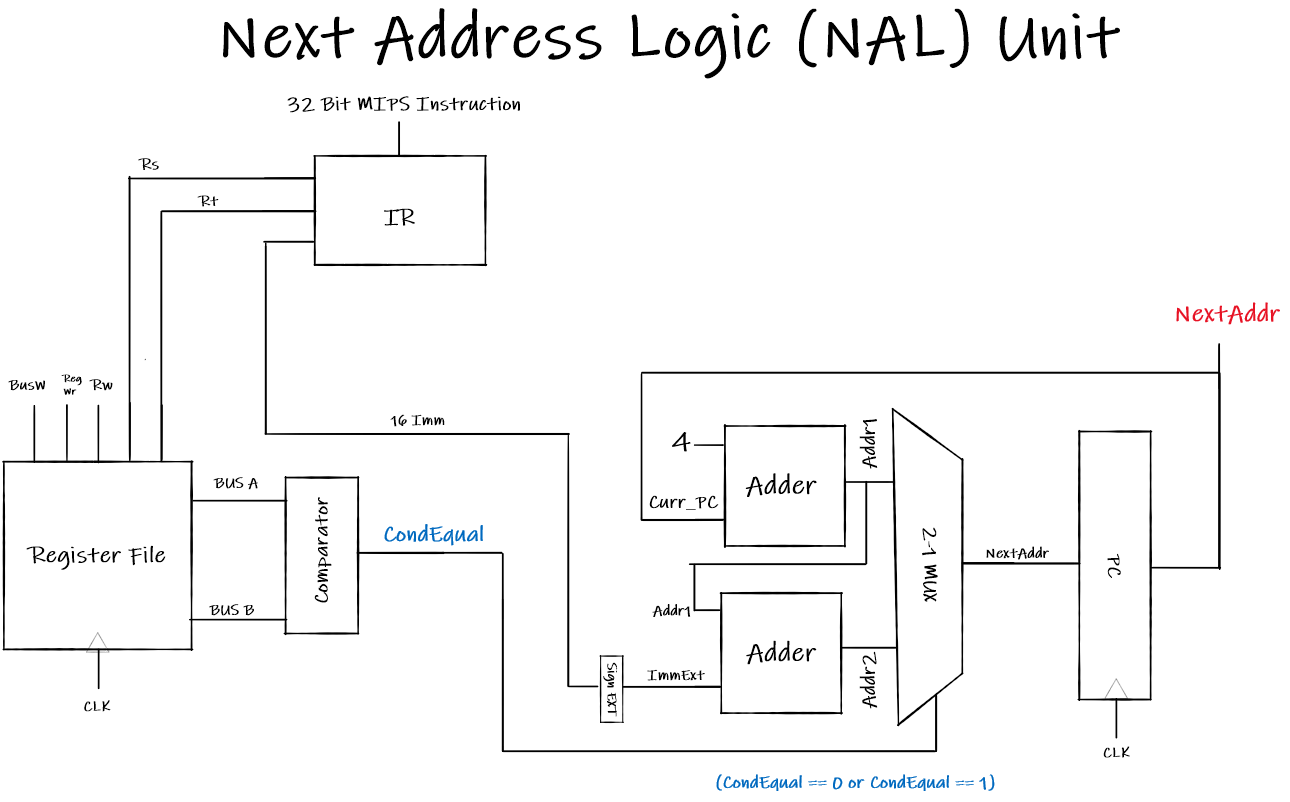


Figure 2: Datapath for BEQ & BNE instructions

# **Components**

## *Instruction Register*

The Instruction Register (IR) component is responsible for decoding a 32-Bit MIPS instruction into the necessary elements for an I-Type Instruction. Hence, it requires the instruction as the only input. The four outputs are the Opcode (which is not needed), the address indices Rs and Rt, and the 16-Bit immediate value. The VHDL code for this component is given in figure 3 below.

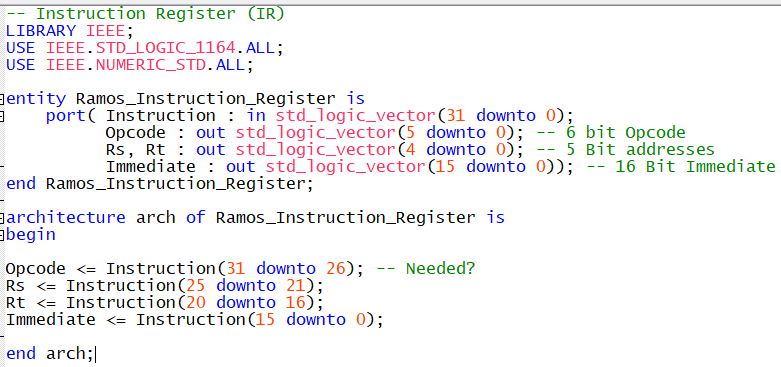


Figure 3: Instruction Register VHDL code

## *Register File*

The register file component will contain predefined arbitrary data located at the arbitrary address indices of Rs and Rt. Based on the values of Rs and Rt (which are fed from the IR), the corresponding data will be sent to outputs BusA and BusB respectively. Note that inputs WREN, RST, Waddress, and Data are present, and behaviors defined, but they will not be used to implement this lab. This is because, we are not writing to any registers and so the inputs could be removed if desired. Figure 4 shows the VHDL implementation for this component.



Figure 4: Register File VHDL code

## *32-Bit Comparator*

The 32-Bit Comparator will compare the outputs of the Register File (i.e. BusA and BusB) to set a condition flag CondEqual either to a 1 or a 0. Figure 5 shows the VHDL implementation for this component using an LPM module.

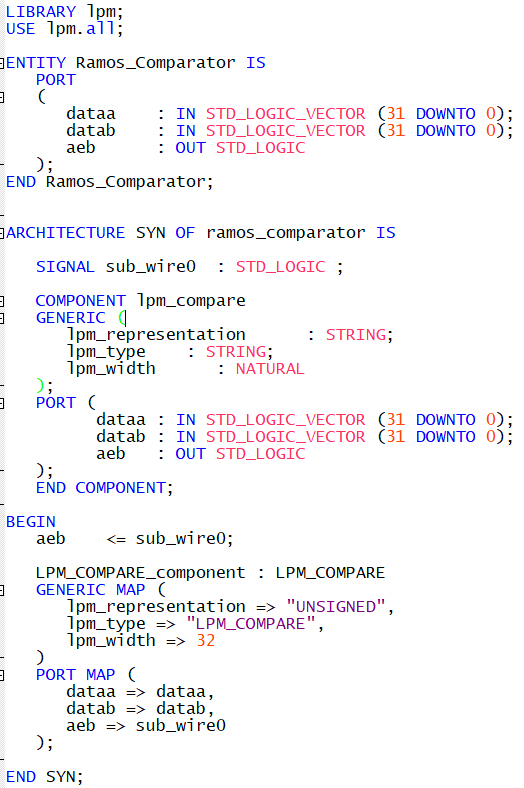


Figure 5: 32-Bit Comparator VHDL code

## *Program Counter*

The Program Counter is a 32-Bit register whose sole purpose is to update the value of the current PC. Updates to the PC value will occur at the rising edge of the clock.

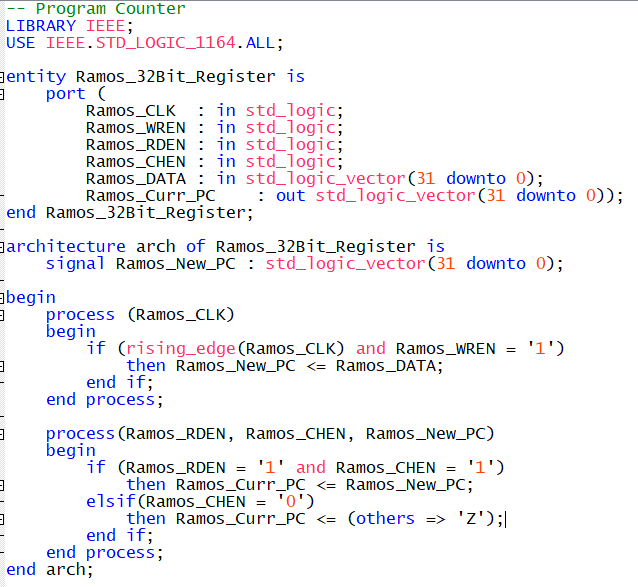


Figure 6: Program Counter VHDL code

## *Sign Extender*

The sign extender will simply convert the 16-Bit Immediate field into 32-Bits which will be necessary to determine the address of the next instruction. Depending on whether the most significant bit is positive (0 – 7) or negative (8 – F), the remaining 16 bits will be filled with either 0’s or F’s to preserve the sign of the value. Hence, the VHDL code in figure 7 is also referred to as a Zero-Sign extender.

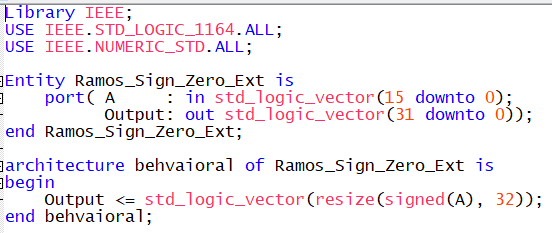


Figure 7: Zero-Sign Extender VHDL code

## *Adder*

Two adder components will be used to compute the two possible PC values as described earlier. Figure 8 shows the VHDL code for a 32-Bit adder using an LPM module.

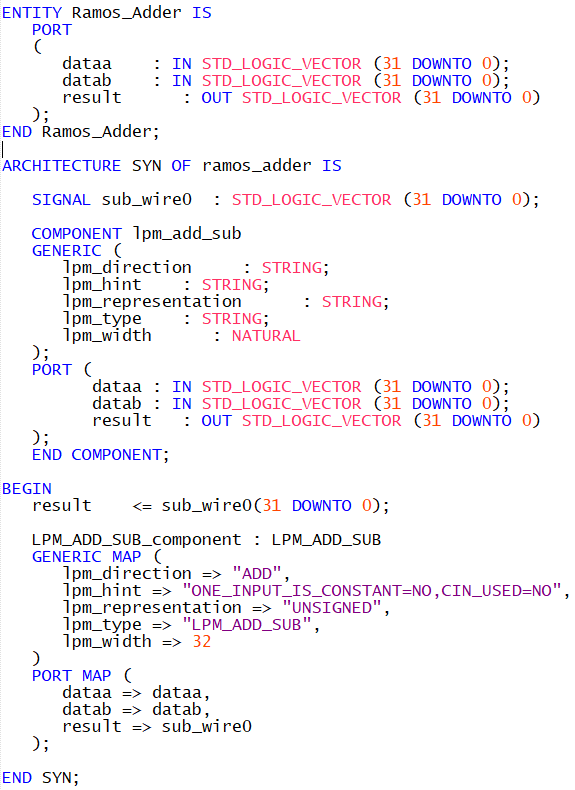


Figure 8: 32-Bit Adder VHDL code

## *2:1 Multiplexer*

The final required component is a 2:1 Multiplexer which will use the CondEqual flag to select one of the two computed PC values to feed back into the PC and give the final PC value.

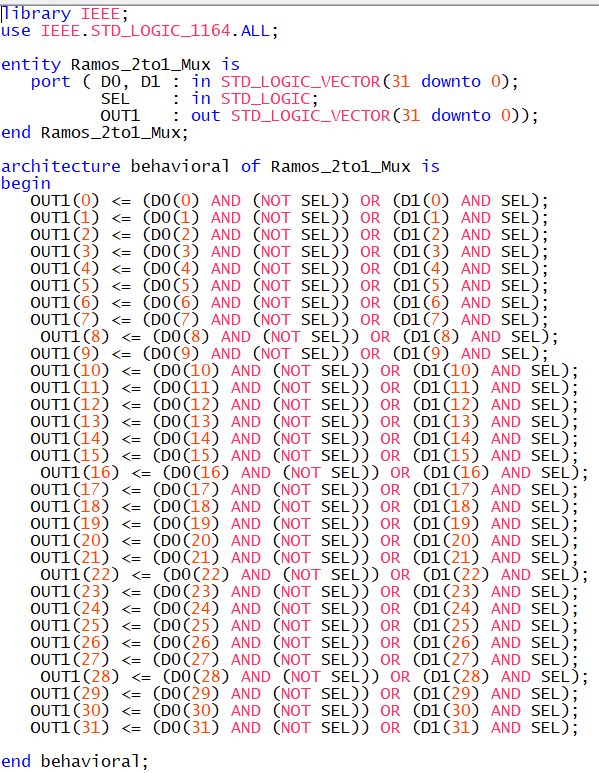
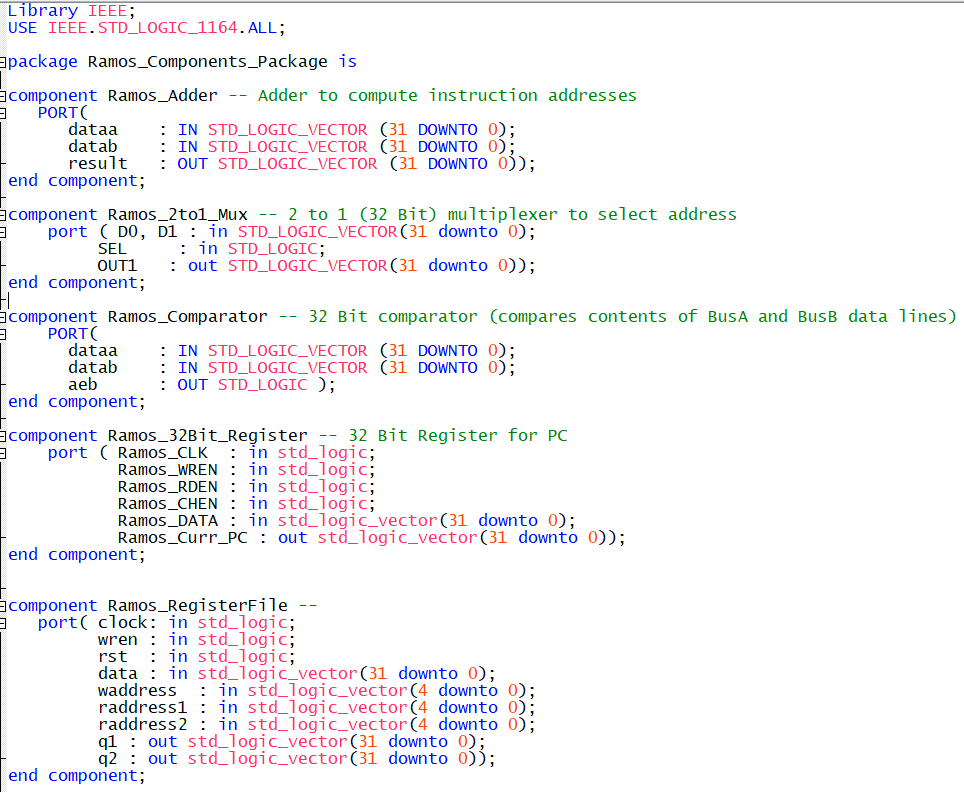


Figure 9: 2:1 (32-Bit) Multiplexer VHDL code

## *Components Package*

A package is created to store all components.



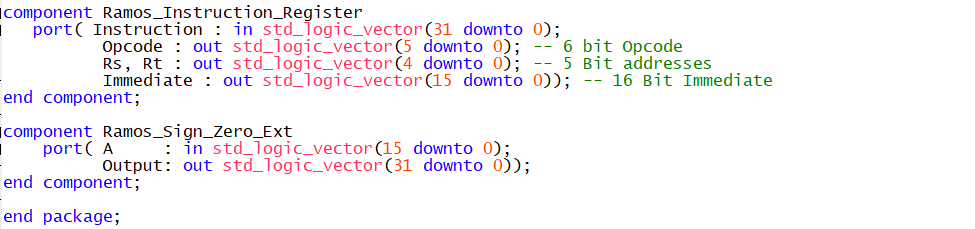


Figure 10: Components Package

## *NAL Unit*

With the utilization of each component, we create a symbol so that we can realize the NAL unit in a block diagram as shown in Figure 11a. Using the components package, the VHDL code for the NAL unit is realized in figure 11b.

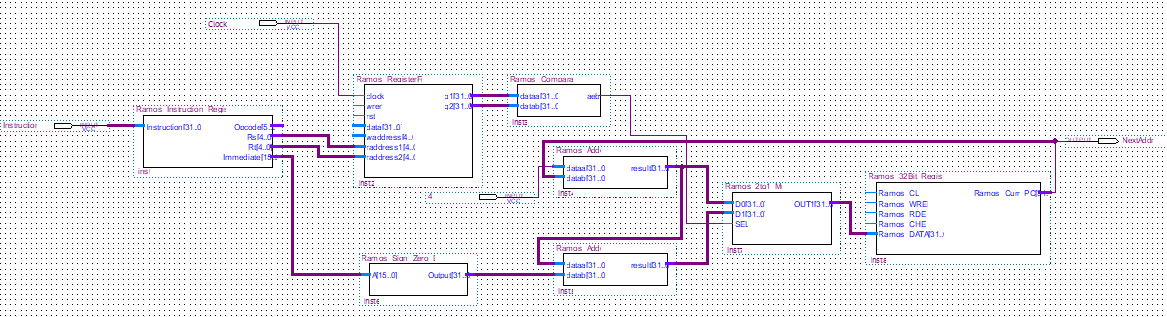


Figure 11a: NAL Unit Block Diagram

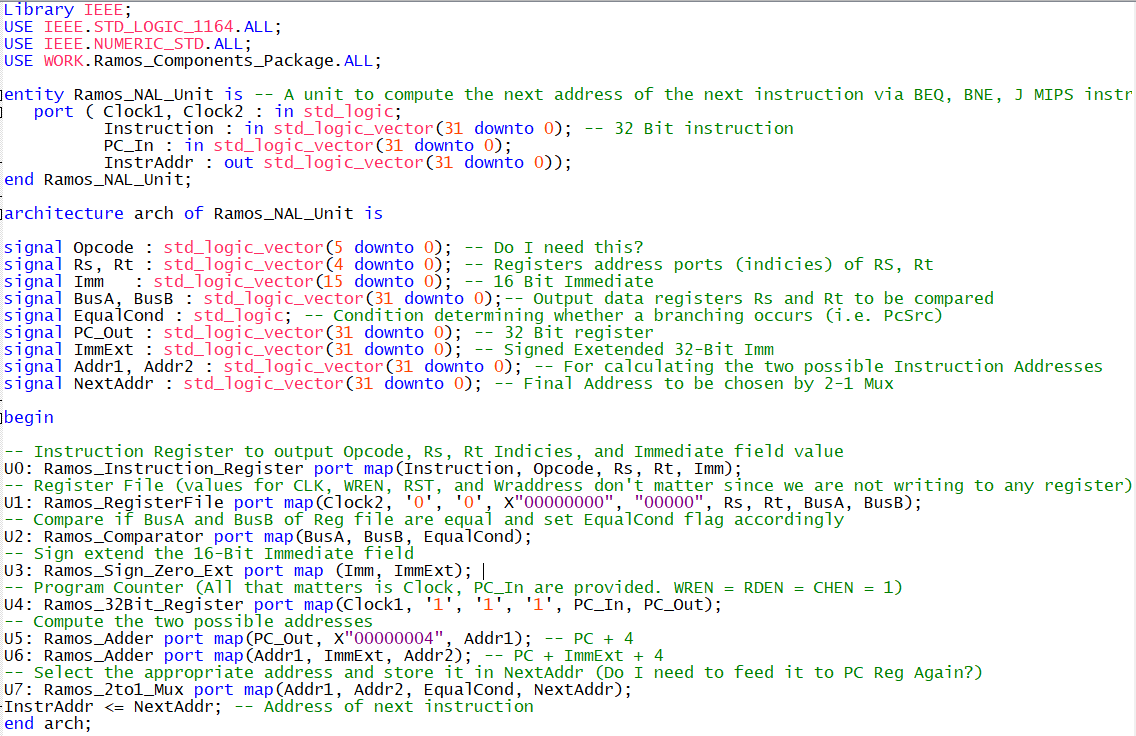


Figure 11b: NAL Unit VHDL Code

# **BEQ Simulation**

The BEQ instruction is a branching instruction that will jump or branch to a label *L* if the contents stored at register R[rs] is equal to the contents of register R[rt]. The address of the next instruction (i.e. the new PC value) is the sum of the current PC value, the sign extended immediate, and 4. If the register contents are not equal, the current PC value is just incremented by 4. Programming-wise, the logic is as follows:

CondEqual <= (R[rs] == R[rt])

If (CondEqual) New\_PC <= Curr\_PC + SignExt(Imm16) + 4

Else New\_PC <= Curr\_PC + 4

To verify the correctness of the unit, we shall perform three verifications. For each verification, a 32-Bit arbitrary instruction is fed into the Instruction Register (IR) where the Rs and Rt address indices as well as the 16-bit Immediate are decoded. These indices serve as inputs to the *register file* which store predefined arbitrary 32-Bit data at those indices. The 16-Bit Immediate is then sign extended. This data is outputted to BusA and BusB which are compared via a comparator to set a condition flag (1 if equal, 0 if not). With a predefined arbitrary program counter value, the two possible addresses are computed using an adder and are fed to a 2:1 multiplexer which will select the final address of the next instruction based on the condition flag set earlier (i.e. select address 2 if condition flag is 1, else select address 1).

## *Verification 1: Not Equal Scenario*

We fed an arbitrary 32-Bit instruction 0x74BB4AD3 into the IR. The IR decodes this instruction to yield indices Rs = 0x05 and Rt = 0x1B along with 16-Bit Immediate field = 0x4AD3. The signed extended ImmExt is 0x00004AD3. We define an arbitrary 32-Bit PC value 0x10000004. The arbitrary 32-Bit data (stored in the register file) for indices Rs and Rt are 0x3FFD02E1 and 0x224CCC80 respectively. This data is fed to BusA and Bus B respectively. It can be seen that data stored at these buses are not equal and so the condition flag is set to 0. Hence the expected output (i.e. the address of the next instruction) is PC + 4 (i.e. InstrAddr = 0x10000008). This is verified in figure 12 below.

## *Verification 2: Equal Scenario*

In the second verification, we fed an arbitrary 32-Bit instruction 0xF2AD52C0 into the IR. The IR decodes this instruction to yield indices Rs = 0x15 and Rt = 0x0D along with 16-Bit Immediate field = 0x52C0. The signed extended ImmExt is 0x000052C0. The PC value is unchanged. The arbitrary 32-Bit data (stored in the register file) for indices Rs and Rt are 0xFFFFEEEE and 0xFFFFEEE respectively. This data is fed to BusA and Bus B respectively. It can be seen that data stored at these buses are equal and so the condition flag is set to 1. Hence the expected output (i.e. the address of the next instruction) is PC + ImmExt + 4. Thus, the address of the next instruction NextAddr is 0x100052C8. This is verified in figure 12 below.

## *Verification 3: Equal Scenario*

In the third verification, we fed an arbitrary 32-Bit instruction 0xED2080F2 into the IR. The IR decodes this instruction to yield indices Rs = 0x09 and Rt = 0x00 along with 16-Bit Immediate field = 0x80F2. The signed extended ImmExt is 0xFFFF80F2. The PC value is unchanged. The arbitrary 32-Bit data (stored in the register file) for indices Rs and Rt are 0x00C3A23B and 0x00C3A23B respectively. This data is fed to BusA and Bus B respectively. It can be seen that data stored at these buses are equal and so the condition flag is set to 1. Hence the expected output (i.e. the address of the next instruction) is PC + ImmExt + 4. Thus, the address of the next instruction NextAddr is 0x0FFF80FA. This is verified in figure 12 below.

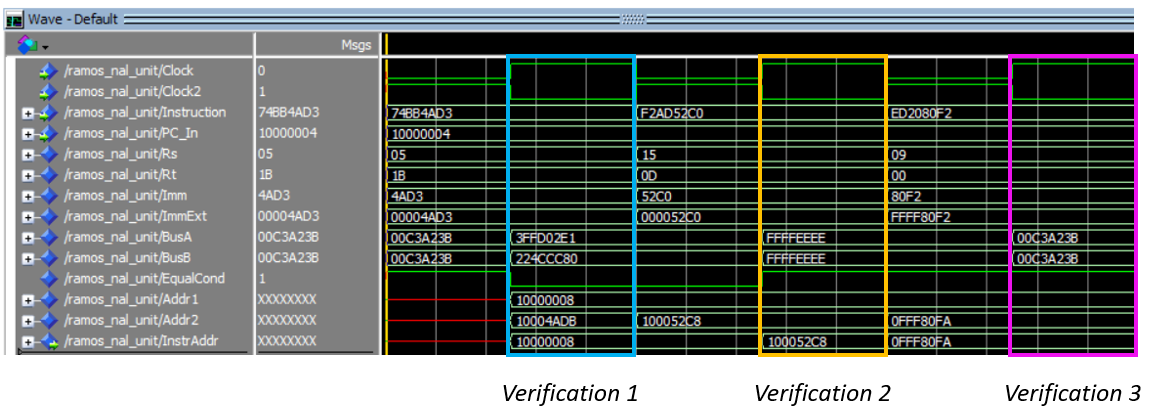


Figure 12: ModelSim Simulation for BEQ instruction

# **BNE Simulation**

The BNE instruction is a branching instruction that will jump or branch to a label *L* if the contents stored at register R[rs] are *NOT* equal to the contents of register R[rt]. To implement the BNE instruction using the data path of the BEQ instruction, the behavior of the 2:1 multiplexer is the only component to be modified. Previously, it selected the sum of the current PC value, the sign extended immediate, and 4 when the EqualCond flag was 1. Instead, this address will be selected when EqualCond flag is 0. This implies that when the EqualCond flag is 1, the current PC value will be incremented by 4. Programming-wise, the logic is as follows:

CondEqual <= (R[rs] == R[rt])

If (CondEqual) New\_PC <= Curr\_PC + 4

Else New\_PC <= Curr\_PC + SignExt(Imm16) + 4

Again, we perform three verification using the same input values to distinguish the working of the BEQ and BNE instruction.

## *Verification 1: Equal Scenario*

We fed an arbitrary 32-Bit instruction 0x74BB4AD3 into the IR. The IR decodes this instruction to yield address indices Rs = 0x05 and Rt = 0x1B along with 16-Bit Immediate field = 0x4AD3. The sign extended immediate (ImmExt) is 0x00004AD3. We define an arbitrary 32-Bit PC value 0x10000004. The arbitrary 32-Bit data (stored in the register file) for addresses indices Rs and Rt are 0x3FFD02E1 and 0x224CCC80 respectively. This data is fed to BusA and Bus B respectively. It can be seen that data stored at these buses are not equal and so the condition flag is set to 1. The address of the next instruction is then PC + ImmExt + 4 = 0x10004ADB. This is verified in figure 13 below.

## *Verification 2: Not Equal Scenario*

In the second verification, we fed an arbitrary 32-Bit instruction 0xF2AD52C0 into the IR. The IR decodes this instruction to yield address indices Rs = 0x15 and Rt = 0x0D along with 16-Bit Immediate field = 0x52C0. The sign extended immediate (ImmExt) is 0x000052C0. The PC value is unchanged. The arbitrary 32-Bit data (stored in the register file) for address indices Rs and Rt are 0xFFFFEEEE and 0xFFFFEEE respectively. This data is fed to BusA and Bus B respectively. The data stored at these buses are equal and so the condition flag is set to 0. The address of the next instruction is PC + 4 = 0x10000008. This is verified in figure 13 below.

## *Verification 3: Not Equal Scenario*

In the third verification, we fed an arbitrary 32-Bit instruction 0xED2080F2 into the IR. The IR decodes this instruction to yield address indices Rs = 0x09 and Rt = 0x00 along with 16-Bit Immediate field = 0x80F2. The signed extended (ImmExt) is 0xFFFF80F2. The PC value is unchanged. The arbitrary 32-Bit data (stored in the register file) for address indices Rs and Rt are 0x00C3A23B and 0x00C3A23B respectively. This data is fed to BusA and Bus B respectively. The data stored at these buses are equal and so the condition flag is set to 0. The address of the next instruction is PC + 4 = 0x10000008. This is verified in figure 13 below.



Figure 13: ModelSim Simulation for BNE instruction