Microprocessor Design Project

Project Objectives

- Design a simple 8-bit MCU
- Improve your understanding of the inner working of an MCU
- Re-enforce your understanding of VHDL/Verilog HDL
- Understand the conversion between assembly instructions and machine code

1 - Introduction

This project will cover the design of a basic microcontroller (MCU). Over the course of the project we will design each of the important functional blocks in an MCU using a hardware description language. Each design will be simulated, and eventually connected together to make a functional MCU. Once a complete MCU is designed, we will implement the MCU on an FPGA.

The MCU we will design will be an an 8-bit, reduced instruction set microcontroller (RISC). Though we are only making an 8-bit MCU, this design can easily be scaled up to include more bits (such as 16- or 32-bit systems). Practically, it is easier to design and implement an 8-bit MCU on an FPGA due to a limited number of functional blocks on an FPGA. This also decreases the time required to simulate, synthesize, and implement the design. Our design will follow a Harvard architecture (meaning that the instruction and data memory and busses will be separate).

Your MCU will be limited to 32 instructions. To make each MCU unique, you will be assigned specific functions to implement in your MCU (no two projects will implement the same two functions). You will not necessarily have 32 instructions, but your processor will have upwards of 16 instructions. Each processor will have basic functionality such as loading and storing from data memory as well as some more advanced functions (multiplication for example). Additionally, each MCU will have basic input/output functionality (IO) and potentially timers and interrupts to differentiate it from a CPU. The design we will use will be based on a 32-bit CPU design from *Logic and Computer Design Fundamentals* by Mano, Kime, and Martin [1].

2 - Functional Components

Your MCU will have at least main components:

- Program Counter
- Instruction Memory
- Instruction Decoder
- Register File
- Arithmetic Logic Unit (ALU)
- Data Memory
- IO Module (potentially serial and/or parallel IO)
- Stack Pointer (Optional)

Each of these components will be connected together using control signals in order to create your MCU. We will now look at the function of each of these components in detail.

2.1 - The Program Counter

The program counter is relatively simple. This is a counter that controls the instruction memory. The value of the program counter is the address of the next instruction to be executed. Generally, the program counter increments by one after each clock cycle. In the case of branches and jumps, the program counter will change to a new value that is either an offset from the current program counter or the value of a register. In the case of this MCU, the program counter will be an 8-bit register, meaning our MCU will be limited to 256 (2^8) instructions.

2.2 - Instruction Memory

The instruction memory stores the program that the MCU will run. Each instruction will be 17-bits in order to accommodate the number of instructions we need. This will be a ROM of size 17-by-256 bits to accommodate 256 instructions. It will take the program counter as an input and output the corresponding instruction (e.g. If the program counter is 00000101_2, then the instruction in location 101_2 (5_{10}) will be output.

2.3 - Instruction Decoder

The instruction decoder takes the instruction that is output by the instruction memory and created the needed control signals for the MCU to execute the required function(s). For example, if the instruction was to add two numbers together, then the instruction decoder would create the signals to do tell the other functional units to do the following:

- Select the proper registers from the register file to add and save the result into
- Tell the ALU to add two numbers
- Select the proper ALU inputs

- Use the output of the ALU to save the data
- etc.

This makes having a properly functioning instruction decoder vital to proper operation of an MCU. Even if every other component works correctly, a malfunctioning instruction decoder can lead to complete malfunction of your MCU.

2.4 - Register File

MCUs use registers to perform functions. The register file is the term used for the group of registers that the MCU has access to for saving data. We will have a total of 8 registers in our MCU. One of these registers will always be equal to zero. All functions will use at least one register (up to 3) to execute every instruction. For example, an instruction could add two registers together and then store the result in a third register. When more than the available registers are required to perform computations, values stored in registers can stored in or loaded from data memory.

2.5 - Arithmetic Logic Unit (ALU)

The arithmetic logic unit is the portion of the MCU that performs mathematical and logical operations. This could be addition, subtraction, multiplication, division, logical shifts, AND, OR, etc. Your ALU will have a limited number of functions compared to most modern MCUs, but it will still be able to perform many useful functions. Additionally, your ALU will output 'flags'. Flags are additional information about the result output by the ALU. Your ALU will have four flags: zero, carry, negative, and overflow.

2.6 - Data Memory

The data memory (usually called RAM on modern computers) is where you can load and store values with your MCU. Your MCU will have instructions to read from and write to the data memory.

2.7 - IO Module

The IO module will be used to interface your MCU with the outside world. This could be connecting the MCU to a keyboard or mouse, a screen, push-buttons, switches, LEDs, or even another MCU.

2.8 - Stack Pointer

The stack pointer is used to implement a stack. A stack is a special way to store information in the data memory. We will cover the stack in more-depth later, but the stack pointer stores the memory address either immediately proceeding or the actual address of the most recently stored value in the stack.

3 - The Pipeline

The architecture of this MCU will consist of a 4-stage pipeline with the following stages:

1. Fetch
2. Decode
3. Execute
4. Writeback

To create this pipeline, certain data will need to be passed between each stage. This will be accomplished using registers. For example, the program counter needs to be passed through to the execute stage in order to properly perform branches. We will look at what happens in each stage of the pipeline now. Remember that each stage in the pipeline happens simultaneously, so each stage of the pipeline can handle its own part of a different instruction simultaneously.

3.1 - The Fetch Stage

The fetch stage of the pipeline is where the next instruction to execute is obtained. The program counter is used to obtain the next instruction from program memory. When the clock pulses, the program counter is incremented by 1 and the instruction is passed to the decode stage. In the event of a branch or jump, the program counter will be replaced with a different value based on control logic.

3.2 - The Decode Stage

In the decode stage, the instruction passed along by the fetch stage is decoded using the instruction decoder. The instruction decoder will send the proper control signals to the register file and any other control logic in order to prepare the data needed in the next stage. The control signals may be used for the execute and writeback stages. Additionally, these signals may be passed back to the fetch stage in order control branching. In addition to the control signals required in the other stages of the pipeline, the inputs to the ALU (values stored in registers and potentially an immediate value from the instruction itself) will also be passed on to the next stage of the pipeline, the execute stage.

3.3 - The Execute Stage

The execute stage of the pipeline is where computation and memory reads and writes take place. Additionally, the address to branch to is calculated in this stage. Even if a branch will not be taken, a dedicated adder is required to calculate the branch address (since the ALU is used to calculate branching conditions such as a value being zero). Once a computation is complete, or a value is loaded or stored from memory, the required control signals as well as the necessary data is passed on to the writeback stage.

3.4 - The Writeback Stage

The writeback stage is the final stage in the pipeline. The only function of the writeback stage is to store data in the register file after computation. Note that since each stage happens

simultaneously, there is a possibility that the data to be saved in a register is needed in an earlier stage in the pipeline. This is called a data hazard (since the data required is not yet saved in the register file). We will address data hazards more in depth later.

4 - MCU Instructions

4.1 - Instruction Formats

We will now cover the various instructions and formats that the processor will need to use. As mentioned above, we will use a 17-bit instruction format. Each instruction will start with an OPCODE. The OPCODE is the portion of the instruction which indicates what function the MCU will perform (e.g. addition, subtraction, branching, jumping, shifting, or logic functions). Our MCUs will have a 5-bit opcode, allowing for 32 unique functions. After the OPCODE will come 2-3 other register addresses. Each register address will be a 3-bit value, allowing selection of any of the 8 registers in the register file. Finally, in the instructions with 2 registers, a 6-bit immediate value is at the end. The immediate value can be used for branching, jumping, or functions in the ALU and can be unsigned or signed (2's compliment). Figure 4.1 shows the different instruction formats mentioned.

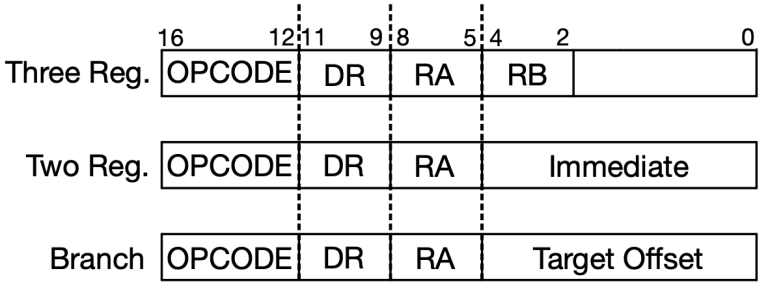

Figure 4.1: The three instruction formats of the MCU.

You will be free to implement other types of instructions with approval from the TA or professor. Notice that the format of the two register and branch instructions is the same. The target offset is a 6-bit, signed number and the immediate can be signed or unsigned (depending on the instruction). The three registers, DR, RA, and RB represent the data register, register A, and register B, respectively. Generally, DR is used for storing the result of instructions, and registers RA and RB are used for computation. For example, the addition of two numbers would take the sum of the values in RA and RB and store the results in RD.

4.2 - MCU Functions

In order for your MCU to function properly, each instruction will require several instructions. **The functions required for every MCU** are listed in Table 4.1.

Note: All logical operations are bitwise and the terms $se\ IM$ and $(0||IM)$ refer to a sign extended immediate and 0 extended immediate, respectively. To refer to the data memory, we will used the following notation $M[RA]$, where this example means the memory location

at the address stored in RA (e.g. if $RA = 5$, then this is equal to the value stored at memory location 5). For branching and jump instructions, PC represents the program counter. For shift instructions SH represents the number of bits to shift.

Operation	**Symbolic Notation**	**Action**
No Operation	NOP	*None*
Move	MOV	$DR \leftarrow RA$
Add	ADD	$DR \leftarrow RA + RB$
Compliment	NOT	$DR \leftarrow \overline{RA}$
Add Immediate	ADI	$DR \leftarrow RA + (se\ IM)$
Load	LD	$DR \leftarrow M[RA]$
Store	ST	$M[RA] \leftarrow DR$
Set if Less Than	SLT	If $RA < RB$, then $DR \leftarrow 1$, else $DR \leftarrow 0$
Branch if Zero	BZ	If $SA = 0$, then $PC \leftarrow PC + 1 + se\ IM$
Branch if Nonzero	BNZ	If $SA \neq 0$, then $PC \leftarrow PC + 1 + se\ IM$
Jump Register	JMR	$PC \leftarrow RA$
Jump	JMP	$PC \leftarrow PC + 1 + se\ IM$
Jump and Link	JML	$PC \leftarrow PC + 1 + se\ IM, DR \leftarrow PC + 1$
Input	IN	$DR \leftarrow IP$
Output	OUT	$OP \leftarrow OP$

Table 4.1: Required instructions and their functions.

In addition to the required instructions, there are several other instructions to potentially implement. Each student will be assigned several of these instructions to implement, but you are free to implement more instructions with the approval of your TA or professor. Some of these instructions are listed in Table 4.2

Operation	**Symbolic Notation**	**Action**
Subtract	SUB	$DR \leftarrow RA + \overline{RB} + 1$
AND	AND	$DR \leftarrow RA$ AND RB
OR	OR	$DR \leftarrow RA$ OR RB
Exclusive-OR	XOR	$DR \leftarrow RA$ XOR RB
Subtract Immediate	SBI	$DR \leftarrow RA + \overline{se\ IM} + 1$
AND Immediate	ANI	$DR \leftarrow RA$ AND $(0\|\|IM)$
OR Immediate	ORI	$DR \leftarrow RA$ OR $(0\|\|IM)$
Exclusive-OR Immediate	XRI	$DR \leftarrow RA$ XOR $(0\|\|IM)$
Add Immediate Unsigned	AIU	$DR \leftarrow RA + (0\|\|IM)$
Subtract Immediate Unsigned	SIU	$DR \leftarrow RA + \overline{(0\|\|IM)} + 1$
Logical Shift Right	LSR	$DR \leftarrow$ lsr RA by SH
Logical Shift Left	LSL	$DR \leftarrow$ lsl RA by SH

Table 4.2: Additional instructions.

4.3 - Assembly Format

We will use the following format to write assembly code:
[label:] INS [RD,] [RA,] [RB] #comment
In this format the label is just a convention to make it easier to keep track of important portions of your code. Additionally, it is easier to refer to labels in your comments than it is to explain specific branch offsets. Here is an example of adding register 1 and 2 together and storing it in register 3:
start: ADD R3, R1, R2 #R3 = R1 + R2
Keep in mind that you should **include comments for every line of assembly code** you write due to the higher difficulty of understanding assembly when compared to higher level languages. Also, note that the "start" is the label for this line. This is a pointer to this instruction and an assembler would use this label as an address for instructions.

5 - Creating Your MCU

For this project, you will be assigned different portions of your MCU to create. These portions, which will be referred to as checkpoints, will be shown to your TA every 1-2 weeks. Additionally, you will need to provide a short write-up (around 1 page) explaining what you have done to accomplish the checkpoint. Before creating anything, you will meet with your TA to discuss the specific functions of your MCU. You will also have the opportunity to suggest changes or additional features you would like to incorporate into your MCU with your TA. **All changes and additions to your MCU must be approved by the TA or professor!**. Table 5.1 shows the approximate schedule we will follow for designing the MCU.

Week(s)	Components	Additional Information
0		Watch Verilog videos and complete verilog examples.
1	Program Memory, Data Memory	Select your OPCODES
2	Register File, ALU	
3	Instruction Decoder	
4-5	Basic MCU	Connect all functional units together in a 4-stage pipeline
6	Data Hazard, Branch Detection, and Stack (optional)	
7	Input and Output (IO)	

Table 5.1: The approximate schedule of the MCU project.

The basic design of the MCU we are designing is shown is shown in Figure 5.1

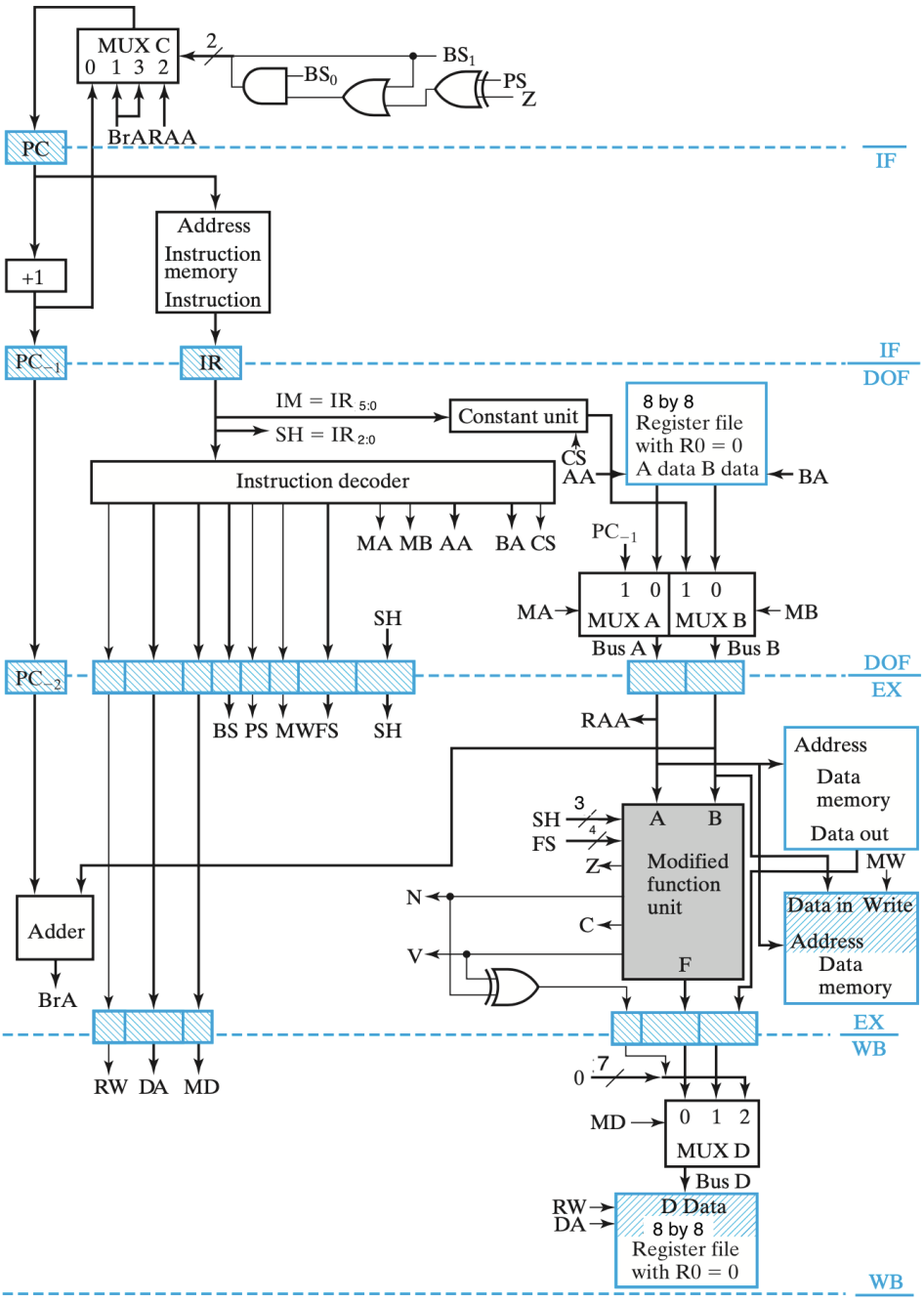

Figure 5.1: The base design of your MCU [1].

We will now look in detail at the requirements for each week's checkpoints.

5.1 - Week 0: Verilog Practice

This week is the best time to get a refresher on Verilog. There are several videos covering the Verilog HDL on eLearning. You should watch each of these videos and then complete the example Verilog designs on eLearning. **If you are having trouble, ask your TA for help**

as soon as possible. Verilog is the backbone of this project so you need to understand it. The designs in this project are not particularly complex, but the better you understand Verilog from the start of the project, the easier it will be.

5.2 - Week 1: Memories and OPCODEs

This week will focus on both the memory units and selecting your OPCODEs. Though you will not use your OPCODEs yet, it is important to have them in one place.

5.2.1 - Program Memory

The program memory is where the instructions will be stored for your program. It will have the following inputs and outputs:

- **Inputs:** Address (8 bits)
- **Outputs:** Instruction (17 bits)

The program memory will contain an 256x17 bit memory. It will use the program counter as the address and output the corresponding 17-bit instruction from the memory.

5.3 - Data Memory

The data memory will be used to store data created by the MCU program. It will have the following inputs and outputs:

- **Inputs:**
  - Clock (1 bit)
  - Address (8 bits)
  - Write Enable (1 bit)
  - Data Input (8 bits)
- **Outputs:** Data Out (8 bits)

The data memory will be a 256 byte memory. The address input will be used to determine the address being written to or read from. At the edge of the clock, the value of the data input will be written to the memory address **only if the write enable input is set to the correct level**. Regardless, the data output will output the byte located in the memory at the address being input.

5.4 - OPCODE Selection

For selecting your OPCODEs, randomly assign OPCODEs to each of your instructions. **No two students should have the same OPCODEs**. Create a CSV file with two columns, the instructions (using the symbolic notations from Tables 4.1 and 4.2) and the 5-bit, binary OPCODEs. Below is an example format for the table:

Instruction	OPCODE
ADD	00000
...	...
SUB	11111

NOTE 1: You will not use all of the available OPCODEs, but you may find that assigning only OPCODEs in order results in more complex logic for other parts of your design down the line.
NOTE 2: In Excel, you can use the a custom format with the type "00000" so that it displays the leading zeros of your OPCODEs. **NOTE 3:** The one exception to assigning random OPCODEs is the NOP OPCODE. To make a later part of the project easier, **the OPCODE of NOP should be 00000**.

5.5 - Week 2: Register File and ALU

The focus of this week is to create the register file and the ALU. The register file will be used for each computation of the MCU as well as loading from and storing to the data memory. The ALU will perform all logical and arithmetic operations for the MCU (such as addition, subtraction, and logical shifting).

5.5.1 - Register File

The register file functions very similarly to the data memory. The main difference is the size (we will only have eight 8-bit registers) and the input/output functionality. It will have the following inputs and outputs:

- **Inputs:**
  - Clock (1 bit)
  - A Address (3 bits)
  - B Address (3 bits)
  - D Address (3 bits)
  - Data In (8 bits)
  - Write Enable (1 bit)
- **Outputs:**
  - A Data (8 bits)
  - B Data (8 bits)

There will be a zero register and 7 general purpose registers. The zero register will always have a value of 0 that cannot be changed (this will be index 0). The 7 remaining registers (indices 1-7) will potentially be overwritten. The A and B Data outputs are controlled by the A and B Address inputs, respectively. Whatever the value of A Address is determines the register value that will be output to the A Data output. The same applies to the B

Address input and the B Data output. The D Address, Data In, and Write Enable inputs will be used to control the writing of data to the registers. If writing is enabled by the Write Enable input, then when the clock pulses the value from Data In will be stored in the register associated with the D Address value (with the exception of the zero register). For example, if the D Address input is '010', then the value of Data In will be stored in register 2.

5.6 - Week 3: Instruction Decoder

The instruction decoder is the final main component needed to create your MCU. The instruction decoder takes the instruction and determines the correct control signals to create in order to have the MCU function properly. The inputs and outputs for the instruction decoder are shown below:

- **Inputs:** Instruction (17 bits)
- **Outputs:**
  - RW - Register Read/Write (1 bit)
  - DA - Data Address (3 bits)
  - MD - MUX D Select (2 bits)
  - BS - Branch Select (2 bits)
  - PS - Zero Toggle (2 bits)
  - MW - Memory Write (1 bit)
  - FS - ALU Function Select (4 bits)
  - MA - MUX A select (1 bit)
  - MB - MUX B select (1 bit)
  - AA - Register A Address (3 bits)
  - BA - Register B Address (3 bits)
  - CS - Constant Select (1 bit)

We will cover the function of each of the instruction decoder's outputs in detail below, but it is your job to use the instruction input to determine the value of each output. Below are the outputs and their functions:

- **Register Read/Write:** This control bit tells the register file if data should be written to the register selected by the data address
- **Data Address:** The data address is used to select the register that is written to.
- **MUX D Select:** This control bus is used to control MUX D, selecting the data that will be written to a register in the writeback stage.
- **Branch Select:** This control signal is used (in combination with logic gates) to control MUX C, and ultimately determine what the next program counter value is.

- **Zero Toggle:** This bit is used to differentiate between branching on zero and branching on not zero. Use Table 5.2 to determine the value of this output.

Register Transfer	BS Code	PS Code	Comments
$PC \leftarrow PC + 1$	00	X	Increment PC
$Z : PC \leftarrow BrA, \overline{Z}PC \leftarrow PC + 1$	01	0	Branch on Zero
$\overline{Z} : PC \leftarrow BrA, ZPC \leftarrow PC + 1$	01	1	Branch on Nonzero
$PC \leftarrow R[AA]$	10	X	Jump to Contents of R[AA]
$PC \leftarrow BrA$	11	X	Unconditional Branch

Table 5.2: The branch logic for the MCU [1].

- **Memory Write:** This control bit is used to determine if data will be written to your data memory.
- **Function Select:** The function select bus determines the function that the ALU will run.
- **MUX A Select:** This control signal selects the inputs of MUX A, which determines the data that will be passed along the pipeline in Bus A.
- **MUX B Select:** This control signal selects the inputs of MUX B, which determines the data that will be passed along the pipeline in Bus B.
- **Register A Address:** This bus selects which register in the register file will output as register A.
- **Register B Address:** This bus selects which register in the register file will output as register B.
- **Constant Select:** This control bit determines if the constant unit will sign extend the immediate value, or if the additional bits will be filled with 0's.

You will need to look closely at Figure 5.1, Tables 4.1 and 4.1, and the OPCODES you selected in order to determine the correct outputs for your instruction decoder. Additionally, some of the outputs may be dependant on the design choices you have made for you previously designed modules (such as the write bit for your register file).

Note: For the time being, you do not need to worry about the input and output instructions. These instructions will be added later in the project.

5.7 - Weeks 4-5: Creating Your MCU

Creating your MCU involves connecting each of your different modules together. Use Figure 5.1 to guide you in connecting the modules. Make sure you have the proper logic to connect the various signals down the pipeline. It may also be helpful to have your memory and register writes occur on the opposite clock edge that the pipeline is updated. You will need to create a few additional components (such as MUXs A-D, the constant unit, and the branch

adder), but most of these modules are rather simple when compared to the others you have made for this project.

Note: Vivado will not operate properly if your design does not have any outputs. For the time being, you can create an output that connects to Bus D in the writeback stage. This should make your design simulate properly. Later, we will add IO functionality to the MCU.

5.8 - Week 6: Data Hazard and Branch Detection

5.8.1 - Data Hazard Stall

One problem with a pipeline architecture is that it takes a few clock cycles in order to write back data into the register file. This means that before one instruction saves the result in the register file, another instruction may be executing that requires this result. When this occurs, it is called a data hazard. To prevent this, we will implement a stall. A data hazard stall stops the program counter and instruction register from changing and prevents incorrect data from being written back to the register file or data memory. Once the correct data has been written back to the register file, the program continues. Effectively, this means an additional clock cycle is required to perform the operation with the correct data. Alternatively, the data can be sent backwards in the pipeline, before being written back to the register file. This process is called data forwarding, but we will not cover this in our project. The correct logic to be implemented is highlighted in Figure 5.2.

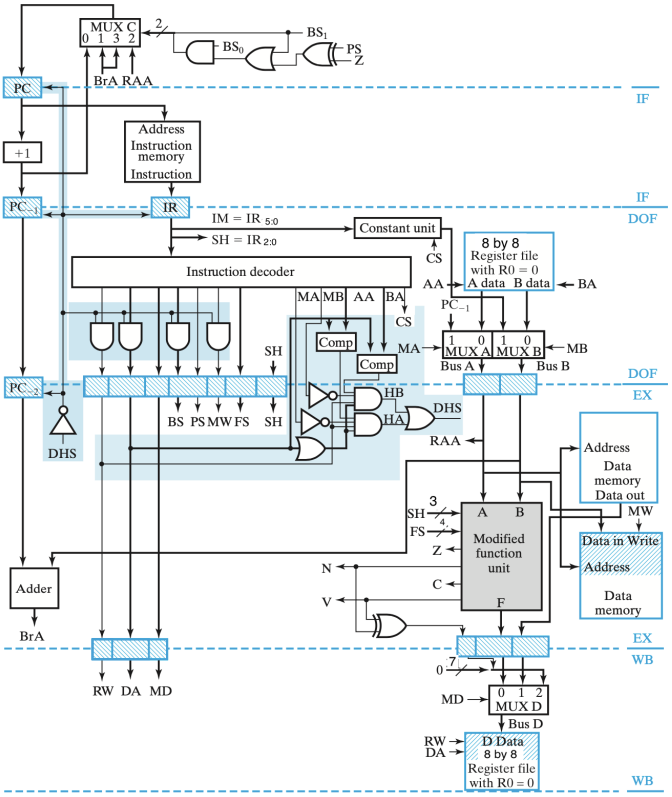

Figure 5.2: The implementation of data hazard stalls.

Note: The OR gate connected to the Data Address bus is an OR of each bit ($DA[2]||DA[1]||DA[0]$).

5.8.2 - Branch Detection

Another problem that can occur with your MCU is instructions being executed after a branch is taken. Since a whether or not a branch is taken is determined after the next instruction is in the pipeline, similar logic needs to be added to prevent the instruction after the branch from occurring. This is accomplished by setting the next instruction to NOP and disabling writing to the register file and program memory (similarly to how hazard stalls are handled). Figure 5.3 show the logic needed to add branch detection to your MCU.

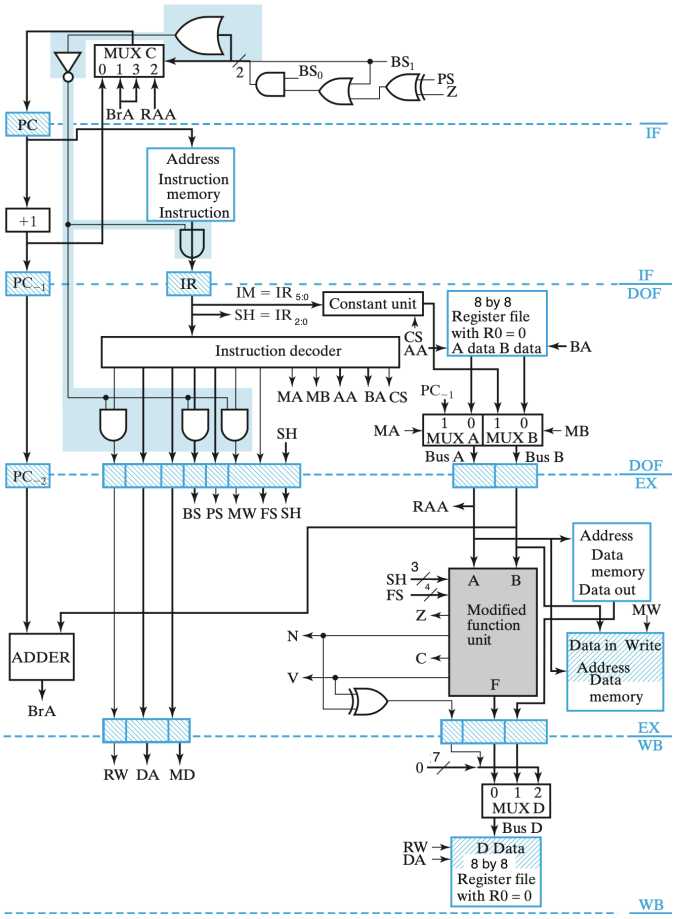

Figure 5.3: The logic needed to implement branch detection in your MCU[1].

5.8.3 - The Stack (Optional)

TO BE ADDED

5.9 - Week 7: Input and Output

A parallel input and output module will be provided to you. This module will be connected to the pin headers on your FPGA. You will be in charge of modifying your MCU in or-

der to properly connect with the module. The module will have the following inputs and outputs:

- **Inputs:**
  - MUC Output (8 bits)
  - FPGA Input (8 bits)
- **Outputs:**
  - MCU Input (8 bits)
  - FPGA Output(8 bits)

The input named "MCU Output" will be where data is passed from the your MCU to the pins of the FPGA. The input named "FPGA Input" will be connected to the input pins of the FPGA (and not to your MCU). The output named "MCU Input" will be where data is passed from the pins of the FPGA to the MCU. The output named "FPGA Output" will be connected to the output pins on the FPGA (and not to your MCU).

References

[1] M. M. Mano, C. R. Kime and T. Martin, "RISC and CISC central processing units," in *Logic and Computer Design Fundamentals*, 5th ed., M. J. Horton, Ed., Hoboken, NJ, USA: Pearson Higher Ed. Inc., 2016, pp. 540–560.