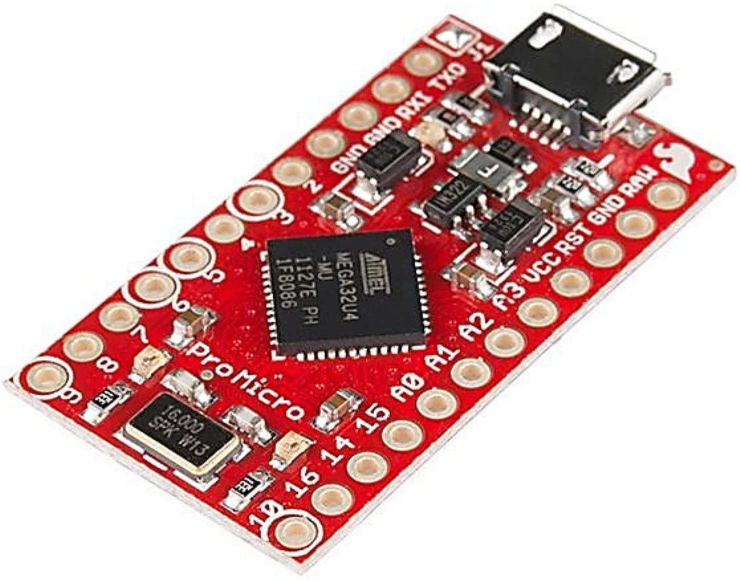
Electrical and Computer Engineering
Erik Jonsson School of Engineering & Computer Science
The University of Texas at Dallas
Dr. Tooraj Nikoubin



Project #4:
(1) Pipeline and Registers
(2) Data Hazard Stall (DHS) and Branch Detection (Br_detect)

EEDG/CE 6302
Microprocessors and Embedded Systems

Objectives:

- Understanding the meaning of pipeline in the MCU design
- Understanding different levels of pipeline and their functions: IF, DOF, EX and WB
- Designing and simulating synthesizable Registers of the MCU
- Writing testbench to test the written codes

1. Introduction:

In this project, you will understand the meaning of the pipeline in your MCU design. Then, to perform the pipeline, you write HDL codes for all registers of the MCU and test at least one of them with the related test bench.

2. Pipeline Datapath

2.1. Steps of instruction execution

To execute any instruction in any microprocessor, it is needed to do (almost) the same four steps:

1. The instruction should be read or “fetched” from memory (Instruction Fetch or IF step)
2. The fetched instructions should be “decoded” by circuitry and required control signals will be prepared. If required, needed operands must be read from Register File or Data Memory (Decode and Operand Fetch or DOF step)
3. The instruction can now be “executed” (Execution or EX step)
4. The results of execution of instructions must be written to specified locations (Write Back or WB step)

After WB, next instruction will be called to IF and this procedure continues until the end of the program.

2.2. pipeline vs non-pipeline datapath

There are two ways to do these steps. First you can run all steps continuously. As you can see in figure (1), in a non-pipeline datapath (or non-pipeline), 12 clock cycles are needed to perform 3 instructions.

Clock cycle	1	2	3	4	5	6	7	8	9	10	11	12
	IF	DOF	EX	WB	IF	DOF	EX	WB	IF	DOF	EX	WB
Instruction				1				2				3

Figure (1): Non-pipeline Datapath

On the other hand, in a pipeline datapath (or pipeline), each step of the pipeline can perform independent jobs related to consequential instruction. As it is illustrated in figure (2), when the pipeline is used, 7 instructions run in 10 clock cycles. Thus, pipeline dominates non-pipeline microprocessor in speed and performance.

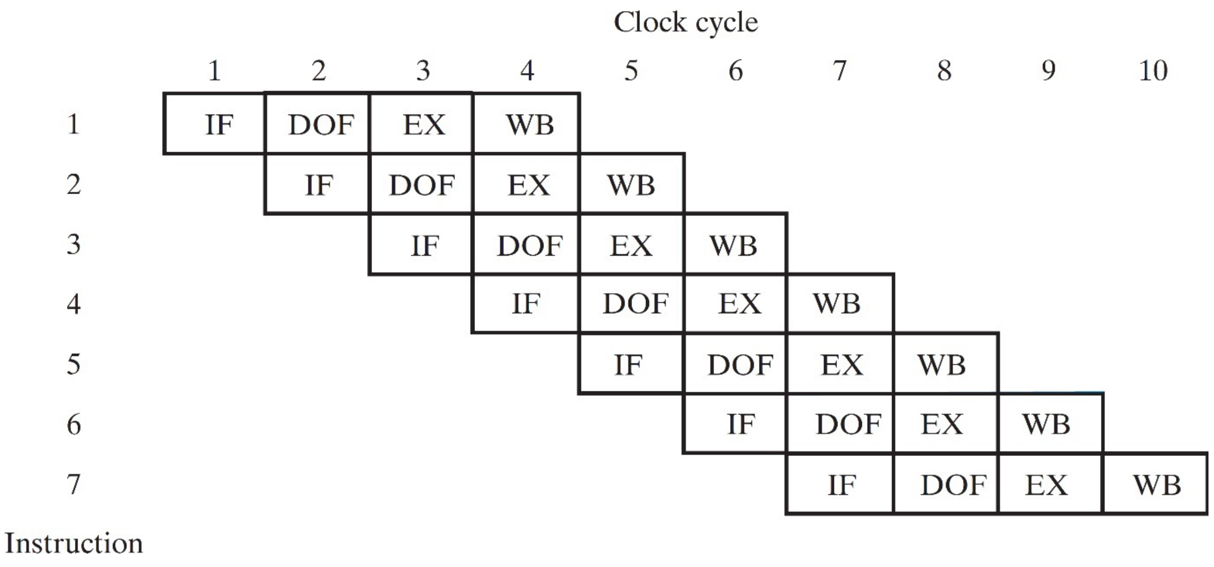


Figure (2): Pipeline Datapath

In this project, we will use pipeline. Pipeline steps are shown with dashed line in figure (6) at the end of this manual. As you can guess, each step pf the pipeline should be able to store the values required for the next step of execution of each instruction. This is called register transfer operations. Hence, we need some registers to store data.

3. Registers

3.1. Required registers

1. Program Counter (PC): For programmable systems, the instructions are usually stored in memory, either in RAM or in ROM. To execute the instructions in sequence, it is necessary to provide the memory address of the instruction to be executed. This address comes from a register called the program counter (PC). As the name implies, the PC has logic that permits it to count. In addition, in order to change the sequence of operations using decisions based on status information from the datapath, the PC needs parallel load capability. Therefore, in the case of a programmable system, the control unit contains a PC and associated decision logic, as well as the necessary logic to interpret the instruction.
2. Instruction Register (IR): Because of the multiple cycles of the modified computer, the instruction needs to be held in a register for use during its execution since its values are likely to be needed for more than just the first cycle. The register used for this purpose is the instruction register or IR. Since the IR loads only when an instruction is being read from memory, it has a load-enable signal (LE) that is added to the control word.
3. PC_1 and PC_2: These two registers hold the up to two previous value of PC. For example, if no branch is in the program and PC points to location 10, PC_1 and PC_2 have values of 9 and 8 respectively. Their contents are used in jump and branch instructions.
4. BUS_A and BUS_B: They are required to save the outputs of MUXA and MUXB.
5. IR_DOF_TO_EX: This register copies required signals from IR and its associated logic circuit in DOF step to the next step of pipeline or EX stage.

6. IR_EX_TO_WB: Like previous register, but its width is less than IR and IR_DOF_TO_EX.
7. OUT_MSB and OUT_LSB: They are used to store values for the output port.
8. D_BIT: This single bit flip-flop is used to save the “D” bit of the ALU/FU outputs. As it is depicted in figure (6), the value of D_BIT must be filled with seven zeros to be able to be applied to the MUXD input. Thus, if D_BIT==’1’, the input 2 of the MUXD will be “00000001”, and if D_BIT==’0’, it will be “00000000”.
9. F_RESULT: It save the results of ALU/FU operation. It goes to the input 0 of MUXD.
10. DATA_MEM: It saves the values read form Data Memory.

3.2. Implementation

Look again to the block diagram of the MCU in figure (6) at the end of this manual. Registers are shown crosshatched in blue. Table (1) summarizes the function of each output.
All registers should have reset (rst) and clock (clk) inputs. They should be sensitive to positive edge of “clk” and active low of “rst”.
As it is shown in table (1), some registers, such as PC and IR, have a control signal named “Load Enable” or “LE”. If LE is activated, a new data will be loaded to registers. Otherwise, the register must retain its current value. Those registers not having LE have to be updated at each positive edge of the clock pulse.

	Width	Save data from	Transfer data to	Load Enable
PC	8	EX	IF	Y
PC_1	8	IF	DOF	Y
IR	17	IF	DOF	Y
PC_2	8	DOF	EX	Y
IR_DOF_TO_EX	18	DOF	EX	N
BUS_A	8	DOF	EX	N
BUS_B	8	DOF	EX	N
IR_EX_TO_WB	6	EX	WB	N
OUT_MSB	8	EX	WB	Y
OUT_LSB	8	EX	WB	Y
D_BIT	1	EX	WB	N
F_RESULT	8	EX	WB	N
DATA_MEM	8	EX	WB	N

Table (1): List of registers

4. Data Hazard Stall and Branch Detection

We saw that pipelines makes the microprocessors faster. However, there are other problems with pipeline operation that may reduce its throughput. We will examine two such problems: data hazards and control hazards. Hazards are timing problems that arise because the execution of an operation in a pipeline is delayed by one or more clock cycles from the time at which the instruction containing the operation was fetched. If a subsequent instruction tries to use the result of the operation as an operand before the result is available, it uses the old or stale value, which is more likely to give a wrong result. There are two solutions for hazard prevention: software-based and hardware-based. We only concentrate on hardware-based solutions.

4.1. Data Hazard Stall (DHS)

The simplest solution to data hazards is inserting NOPs automatically in the pipeline. In other words, when an operand is detected at the DOF stage that has a data hazard and it also not been written back yet, the associated execution and write-back are delayed by “stalling” the pipeline flow in IF and DOF for one clock cycle. Then the flow resumes with completion of the instruction when the operand becomes available, and a new instruction is fetched as usual. The delay of one cycle is enough to permit the result to be written before it is read as an operand.
When the actions associated with an instruction flowing through the pipeline are prevented from happening at a given point, the pipeline is said to contain a bubble in subsequent clock cycles and stages for that instruction. If the pipeline flow is held in IF and DOF for an extra clock cycle, the pipeline is said to be stalled, and if the cause of the stall is a data hazard, then the stall is referred to as a data hazard stall (DHS).

4.2. Branch Detection (Br_Detect)

Another type of hazards in the MCU is called control hazards. Control hazards are associated with branches in the control flow of the program. Imagine we want to run following program:

```
1       BZ       R1, 18
2       MOVA     R2, R3
3       MOVA     R1, R2
4       MOVA     R4, R2
20      MOVA     R5, R6
```

First column shows the address of instruction in the memory. If R1==0, then a branch to the instruction in location 20 (recall the operation of BZ instruction) is to occur, skipping the instructions in locations 2 and 3. Assume that the branch is taken to location 20 because R1 is equal to zero. But R1==0 is not detected until EX in cycle 3 of the first instruction and the PC is set to 20 on the clock edge at the end of cycle 3. On the other hand, the MOVA instructions in locations 2 and 3 are into the EX and DOF stages, respectively, after the clock edge. Therefore, incorrect jump will occur unless corrective action is taken. The problem can be seen in figure (3).

A solution to this problem is named “Branch Detection.” During normal operation, when branches are not taken, instructions will be fetched and decoded, and then operands will be fetched based on the addition of 1 to the value of the PC. If the branch is taken, the instructions following the branch instruction need to be canceled. The cancellation is done by inserting bubbles into the execution and write-back stages for these instructions. This is illustrated in the figure (4) for the abovementioned example. Although instructions 2 and 3 have entered the pipeline, they are not executed and ignored by the branch prediction logic. These bubbles have the same effect as two NOP instruction in the pipeline.

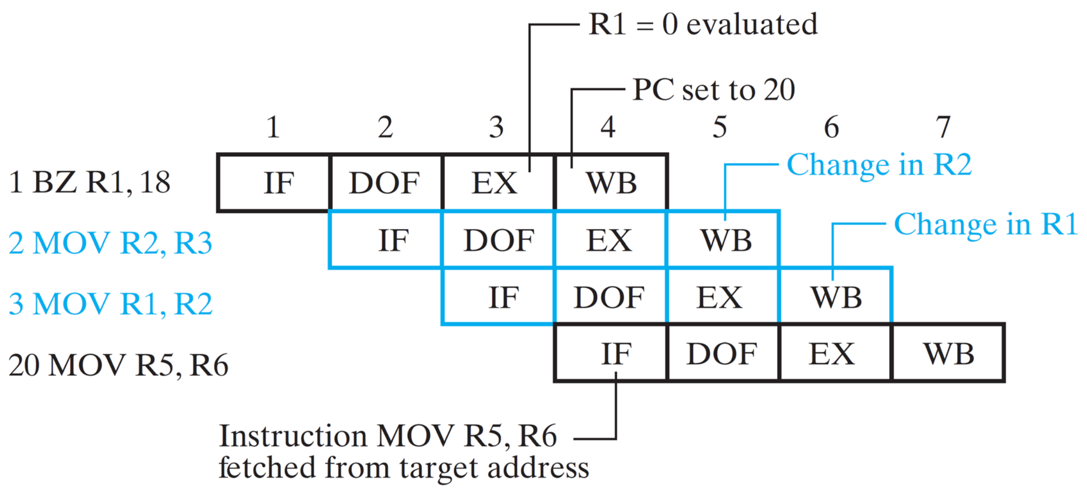


Figure (3): Control Hazard

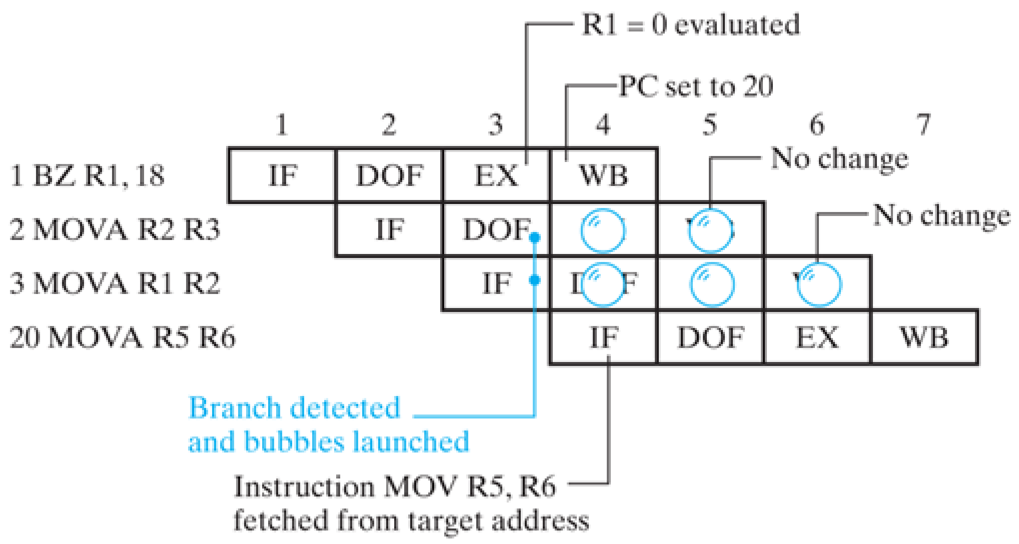


Figure (4): Branch Prediction when a branch is taken

The control organization of branches is based on the appropriate selection of the available values for the PC. This section is done in MUXC. In Figure (5), you can see the logic used to generate BS and PS bits in branch detection control process.

Definition of Control Fields BS and PS

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

Figure (5): PS and BS bits generation

4.3. Implementation

The logic required to perform DHS and Br_Detect is shown in the areas shaded in light blue in figure (6).
DHS requires two comparators. If two inputs of the comparators are the same, they generate ‘1’ at their outputs. Also not that 3-bit DA signals go to an OR gate. It means that all bits of DA must be ORed together to generate one bit result. You may use reduction operator in HDLs. A single bit DHS should be inverted to generate $\overline{\text{DHS}}$ and then, applied to LE of four registers in the MCU.
Br_Detect requires a simpler logic and an adder. When a branch is taken, it flushes the value of IR by writing all zeros into this register and prevents running the previously fetched instruction. The MUXC will select the next correct value for the PC. Note that although the MUXC is shown on top of the MUC block diagram in figure (6), it is logically locate in EX area. The MUXC has two selectors: S[1] and S[2]. S[1] is directly connected to BS[1] and S[0] requires extra logic to be created. S[1] and S[0] are ORed to create Br_Detect which is inverted to produce $\overline{\text{Br\_Detect}}$.
Notice that there are four AND gates at the outputs RW, DA, BS and MW of the Instruction Decoder. For signals such as BS which has 2 bits, you need to AND the $\overline{\text{DHS}}$ and $\overline{\text{Br\_Detect}}$ to all bits of that signal. For example,

In Verilog: BS & {2{DHS_N}} & {2{Br_Derect_N}}
In VHDL: BS and (others=> DHS_N) and (others=> Br_Derect_N)

Where DHS_N and Br_Derect_N mean negated of DHS and Br_Derect, respectively.

5. Project requirements:

- Write codes in an HDL for all registers, DHS and Br_Detect.
- Write a testbench for to evaluate the PC. To show that the Constant Unit is working properly, consider all conditions in your testbench.
- Take a screenshot of the report of Xilinx ISE (or any other HDL synthesizers) which shows your registers do contain only flip-flops and no latches. Insert the screenshot in your report.
- Each student should submit a report no more than 4 pages (excluding cover page, TOC, appendix, etc).
- Do not include your codes in the report. Submit your HDL codes along with their testbenches separately.
- Answer to the questions at the end of this manual in the report.

5. Questions:

1. What is the difference between active high and active high control signals?
2. What is the difference between positive-edge-triggered and negative-edge-triggered control signals?
3. What is the difference between a latch and a flip-flop?
4. Why do all registers need reset signal? When is reset usually applied to the microprocessors? Explain in 2 to 3 lines.
5. Some flip-flops or latches have preset input. If it is activated, the module will be filled with all ones. Can we have “preset” signal instead of reset in the MCU? Explain in 2 to 3 lines.
6. Imagine there is no hardware or software solution for hazards in the MCU. Explain why runing following codes leads to an incorrect result after entering the pipeline:

		1	2	3	4	5	6
MOVA R1, R5	R1 ← R5	IF	DOF	EX	WB		
ADD R2, R1, R6	R2 ← R1 + R6		IF	DOF	EX	WB	
ADD R3, R1, R2	R3 ← R1 + R2			IF	DOF	EX	WB

7. Another hardware solution for dta hazard is called “data forwarding“. Explain it briefley in 3 to 5 lines.

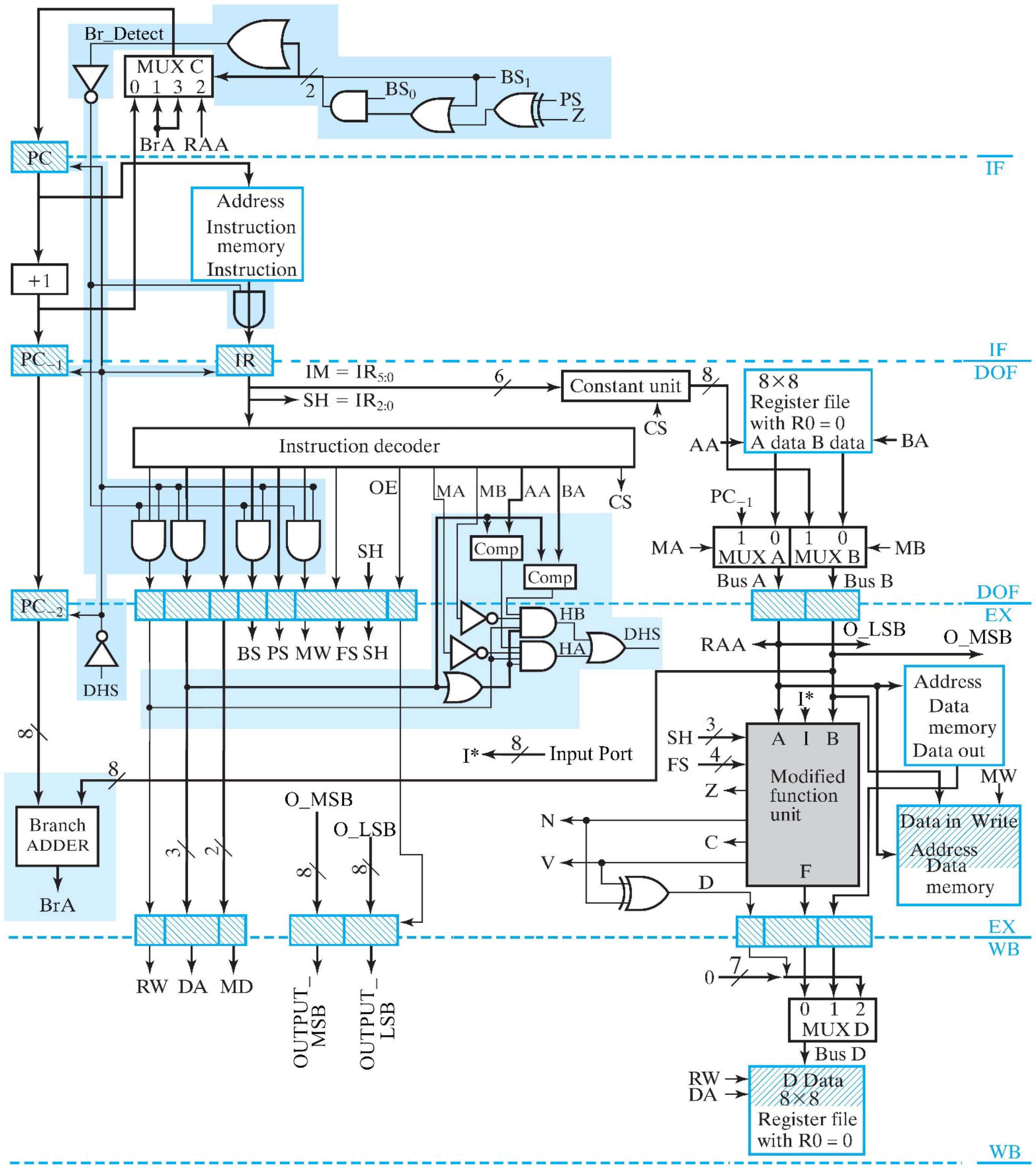


Figure (6): MUC Block Diagram

Reference:
Mano, M. M. AND Kime, C. R. AND Martin M. “*Logic and Computer Design Fundamentals*“, 5th ed. Hoboken, NJ: Pearson Higher Education Inc., 2015, pp 585-615.