CSE 210
Computer Architecture Sessional
Assignment 3: MIPS Design and Simulation

6 December 2024

Group: 02
Section: A1

Members of the group:

- 2105007 - Nafis Nahian
- 2105008 - Sk. Ashrafuzzaman Nafees
- 2105009 - Abrar Zahin Raihan

1 Introduction

A processor, also known as a processing unit, is a digital circuit that carries out operations on data from external sources, such as memory or data streams. While the term "processor" can refer to various types of processing units, it is most commonly associated with the Central Processing Unit (CPU) of a system. The CPU serves as the core of a computer, executing instructions from programs and performing essential computational tasks.

The CPU is composed of intricate electronic circuitry designed to handle fundamental operations, including arithmetic calculations, logical decision-making, control processes, and input/output (I/O) operations. These tasks are executed based on the instructions defined in the computer program, making the CPU central to the functionality of the system.

Among the CPU's primary components is the Arithmetic Logic Unit (ALU), which is responsible for performing mathematical computations and logical comparisons. Additionally, processor registers play a crucial role by providing data to the ALU for processing and storing the results of these operations. Complementing these components is the control unit, which orchestrates the entire process by fetching instructions from memory, interpreting them, and coordinating the activities of the ALU, registers, and other parts of the CPU. This coordinated process ensures that instructions are executed in the correct sequence and timing, maintaining the logical flow of operations within the system.

There are many types of processor design implementation. MIPS (Microprocessor without Interlocked Pipelined Stages) is a reduced instruction set computer (RISC) instruction set architecture (ISA). The Processor Design implementing MIPS ISA is called MIPS processor.

The processor components that we implemented are as follows:

1. **Program Counter and Instruction Memory:** The program counter (PC) is an 8-bit register which activates at the falling edge of the clock signal. After every clock cycle, it adds 1 to its previous address. The value it stores is used as the Instruction Memory address.

2. **Register File:** The Register File is a bank of seven registers. These are denoted as `$zero`, `$t0`, `$t1`, `$t2`, `$t3`, `$t4`, `$t5`, and `$sp`. The `$sp` register acts as the stack pointer, while the `$zero` register always holds the value zero. The other registers serve as general-purpose registers.

3. **ALU:** The ALU performs all calculations. It is controlled by a 3-bit `ALUop` code that determines the type of calculation it performs. It operates on two 8-bit binary numbers.

4. **Control Unit:** The Control Unit decodes the instruction by providing selection inputs to all multiplexers, the Register File, the Data Memory, and the ALU.

5. **Data Memory:** The Data Memory stores stack values and functions as the main memory. It stores data as 8-bit values.

Instruction pipelining is a method used to enhance the efficiency of instruction execution in a processor by dividing the process into distinct stages. Each stage performs a part of the instruction cycle, allowing multiple instructions to overlap in execution. This increases the overall instruction throughput, enabling the processor to execute several instructions simultaneously without reducing the time required to complete an individual instruction.

Generally, each instruction is divided into five stages:

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execution and Address Calculation (EX)
4. Data Memory Access (MEM)
5. Write Back (WB)

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called hazards. Hazards are categorized into three types:

1. **Data Hazards:** Occur when an instruction depends on the result of a previous instruction that has not yet completed its execution.
2. **Control Hazards:** Arise when the pipeline is unable to predict the flow of control, typically caused by branch instructions.
3. **Structural Hazards:** Happen when multiple instructions require the same hardware resource simultaneously, leading to resource contention.

The processor we designed supports pipelined datapath for a subset of MIPS instruction set. We considered EX hazard, MEM hazard, load use, load store and double data hazard. We also considered Control/branch hazard, and data hazard during the branching.

2 Instruction Set

2.1 Instruction Set With Instruction ID

Instruction ID	Category	Type	Instruction
A	Arithmetic	R	add
B	Arithmetic	I	addi
C	Arithmetic	R	sub
D	Arithmetic	I	subi
E	Logic	R	and
F	Logic	I	andi
G	Logic	R	or
H	Logic	I	ori
I	Logic	R	sll
J	Logic	R	srl
K	Logic	R	nor
L	Memory	I	sw
M	Memory	I	lw
N	Control	I	beq
O	Control	I	bneq
P	Control	J	j

2.2 Instruction Set With Op-Code

Op-Code	Category	Type	Instruction
0000	Arithmetic	I	subi
0001	Control	I	bneq
0010	Memory	I	sw
0011	Control	I	beq
0100	Arithmetic	R	add
0101	Logic	I	ori
0110	Logic	R	srl
0111	Memory	I	lw
1000	Control	J	j
1001	Logic	R	or
1010	Logic	R	nor
1011	Logic	I	andi
1100	Arithmetic	R	sub
1101	Logic	R	and
1110	Arithmetic	I	addi
1111	Logic	R	sll

3 Complete Circuit Diagram

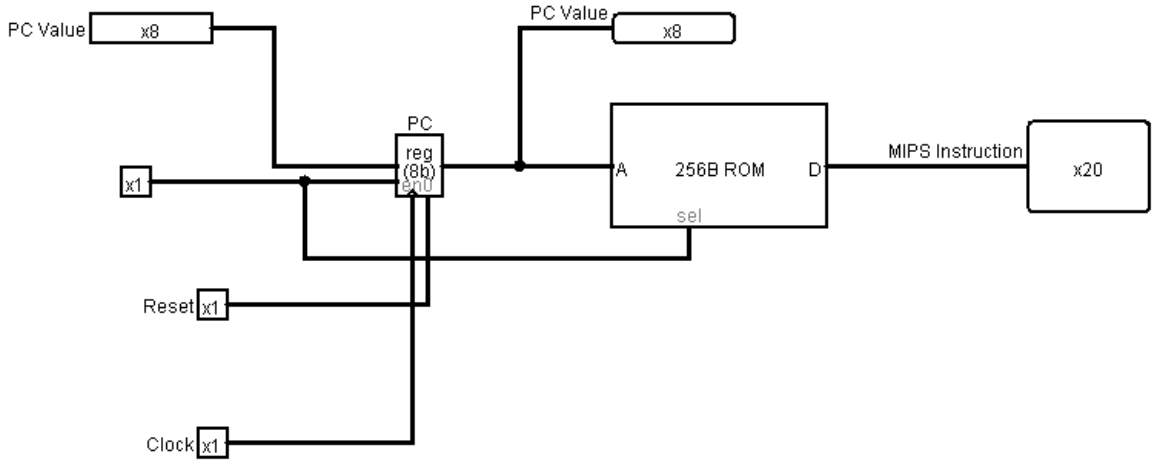


Figure 1: Instruction Memory with Program Counter

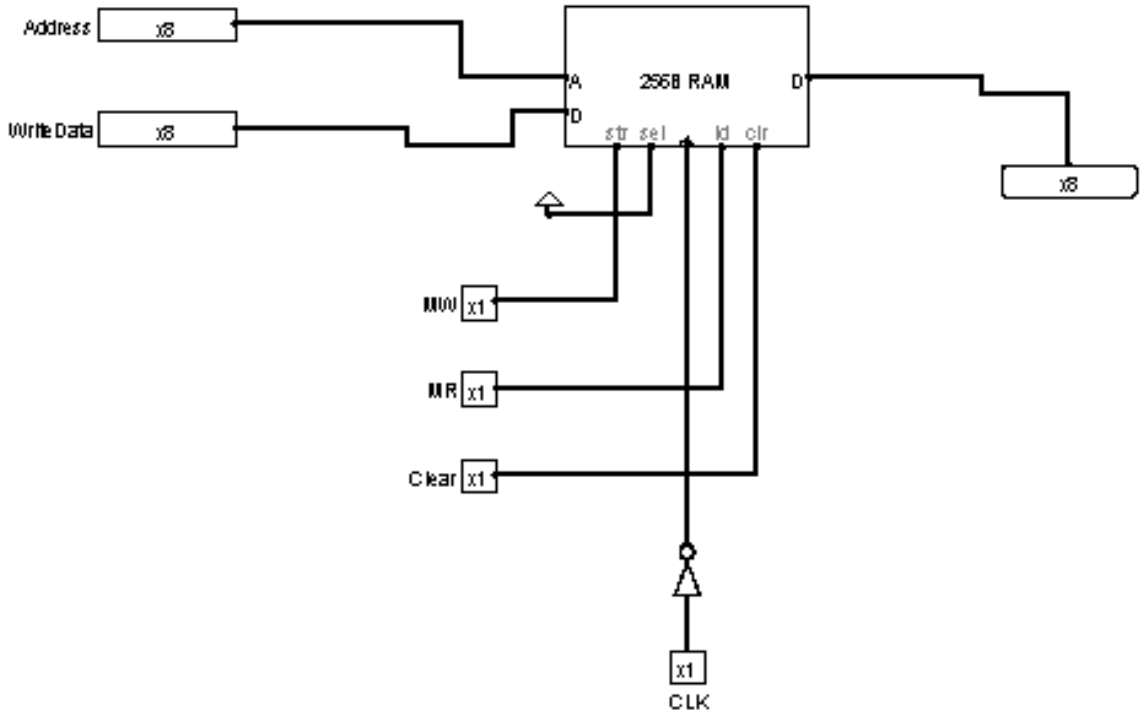


Figure 2: Data memory with stack

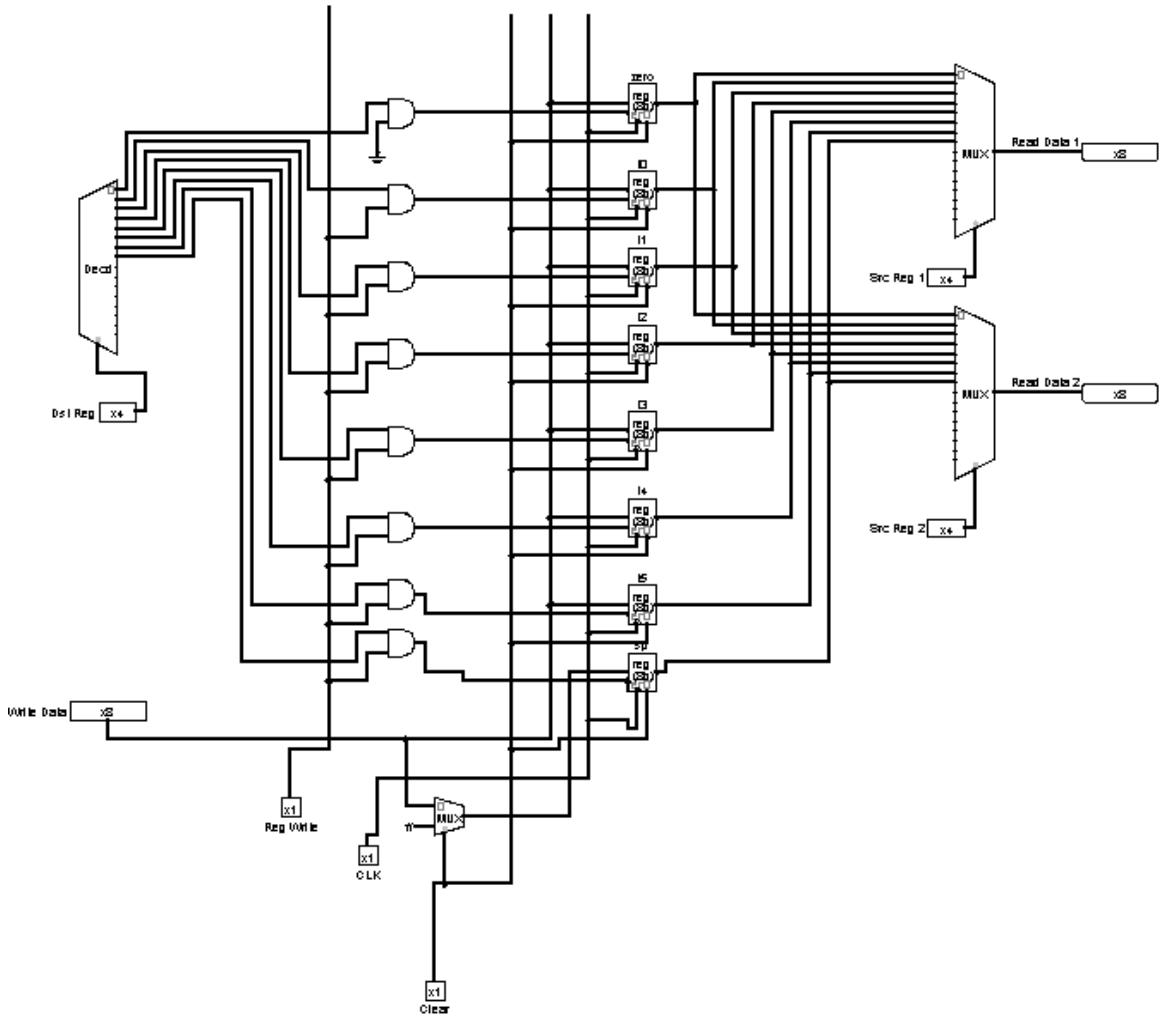


Figure 3: Register Files

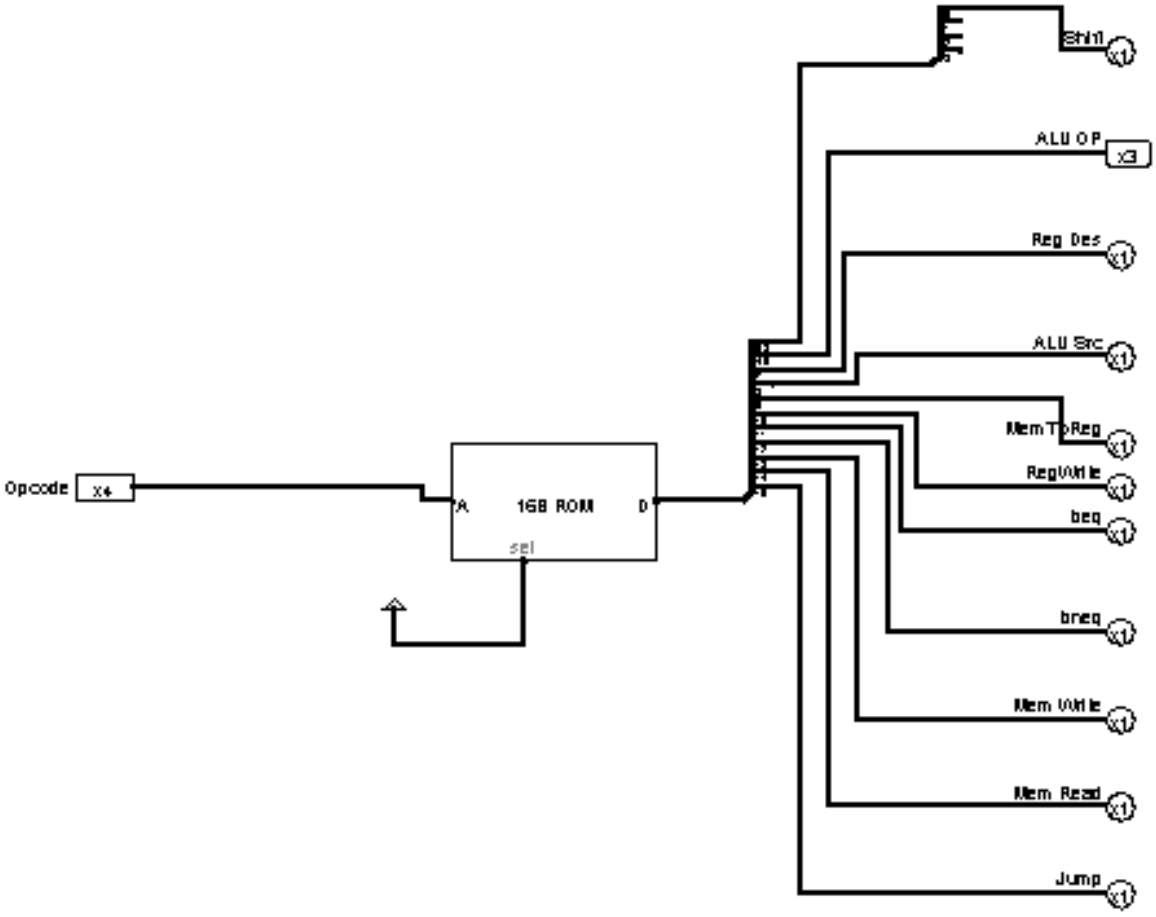


Figure 4: Control Unit

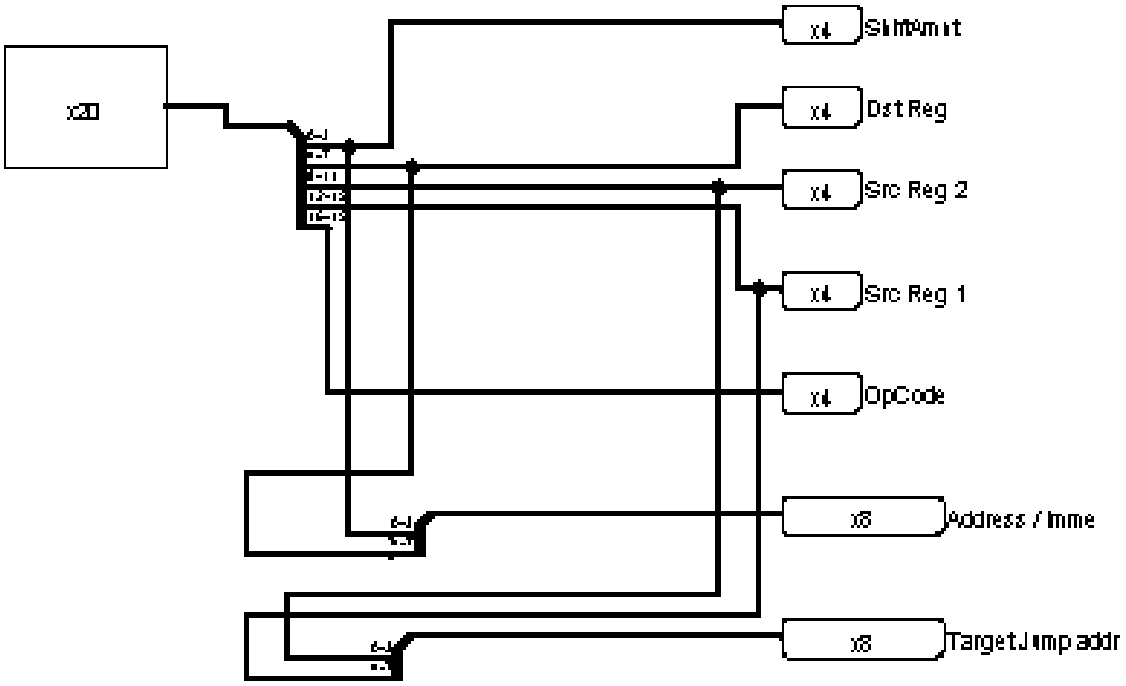


Figure 5: Instruction Splitter

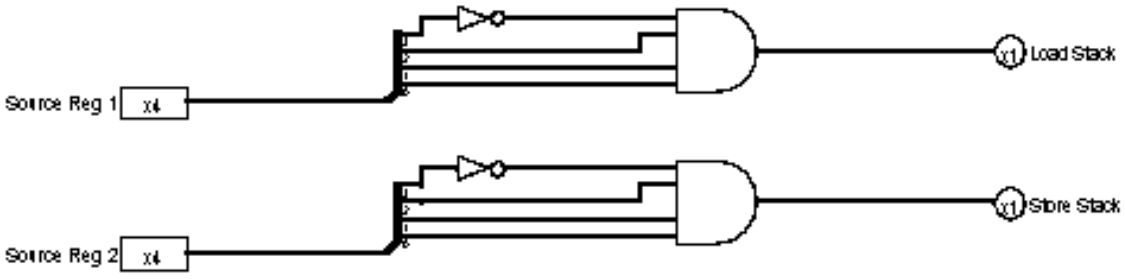


Figure 6: Stack Logic

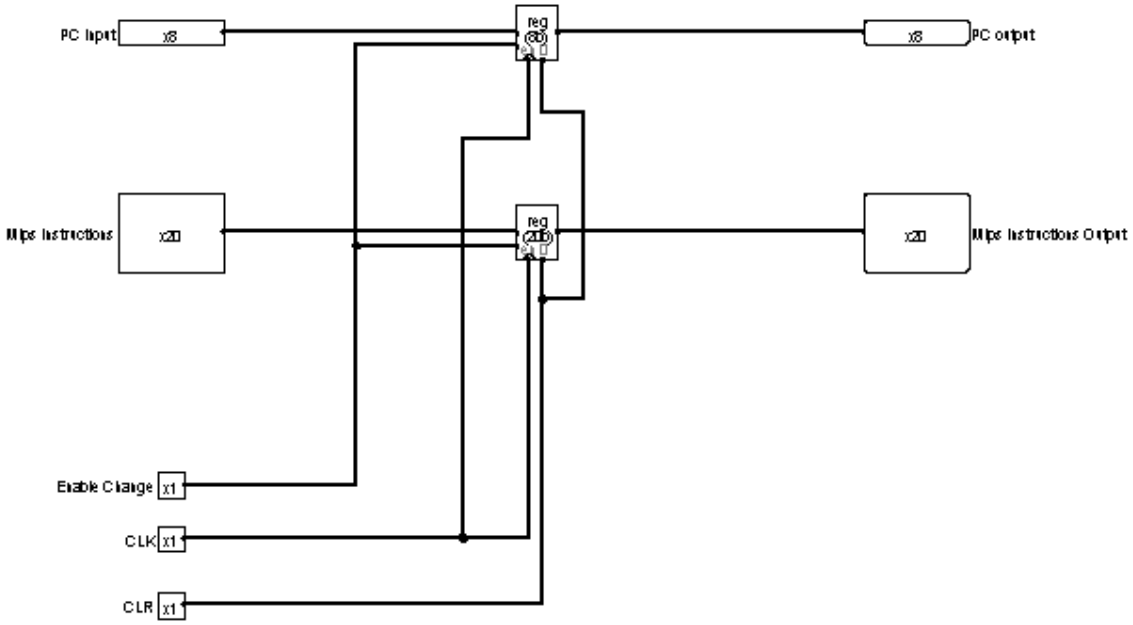


Figure 7: IF/ID Register

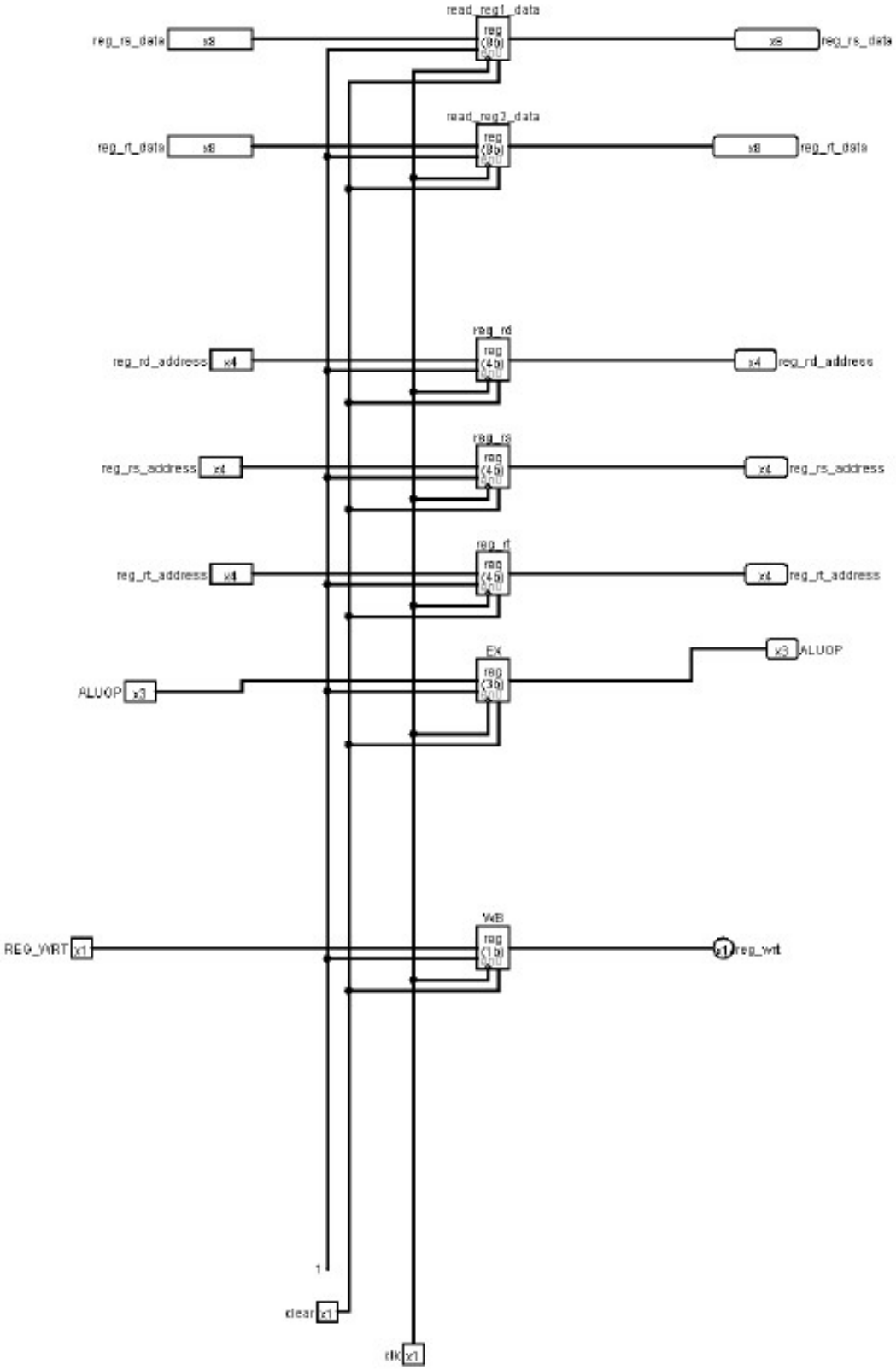


Figure 8: ID/EX Register

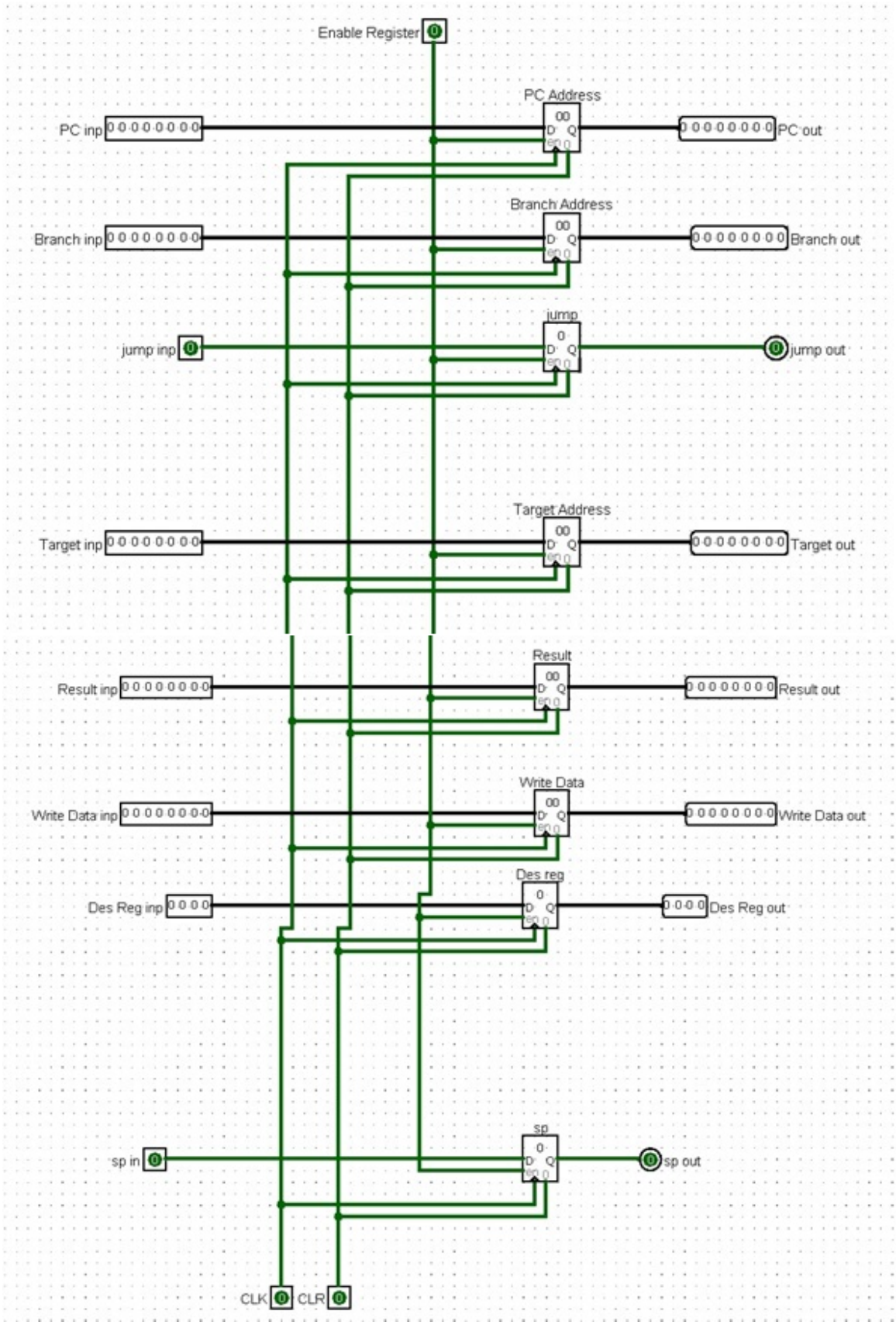


Figure 9: EX/MEM Register

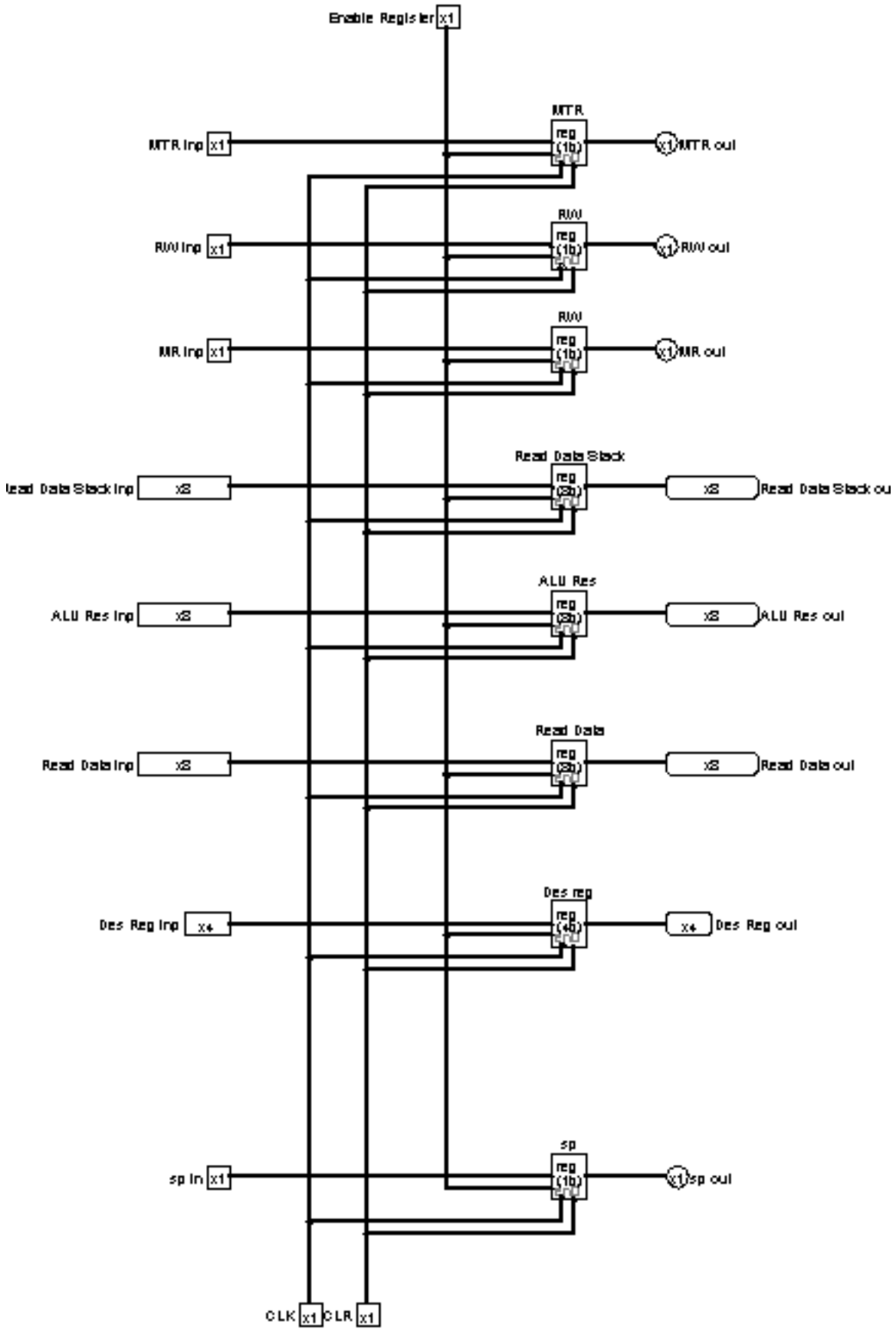


Figure 10: MEM/WB Register

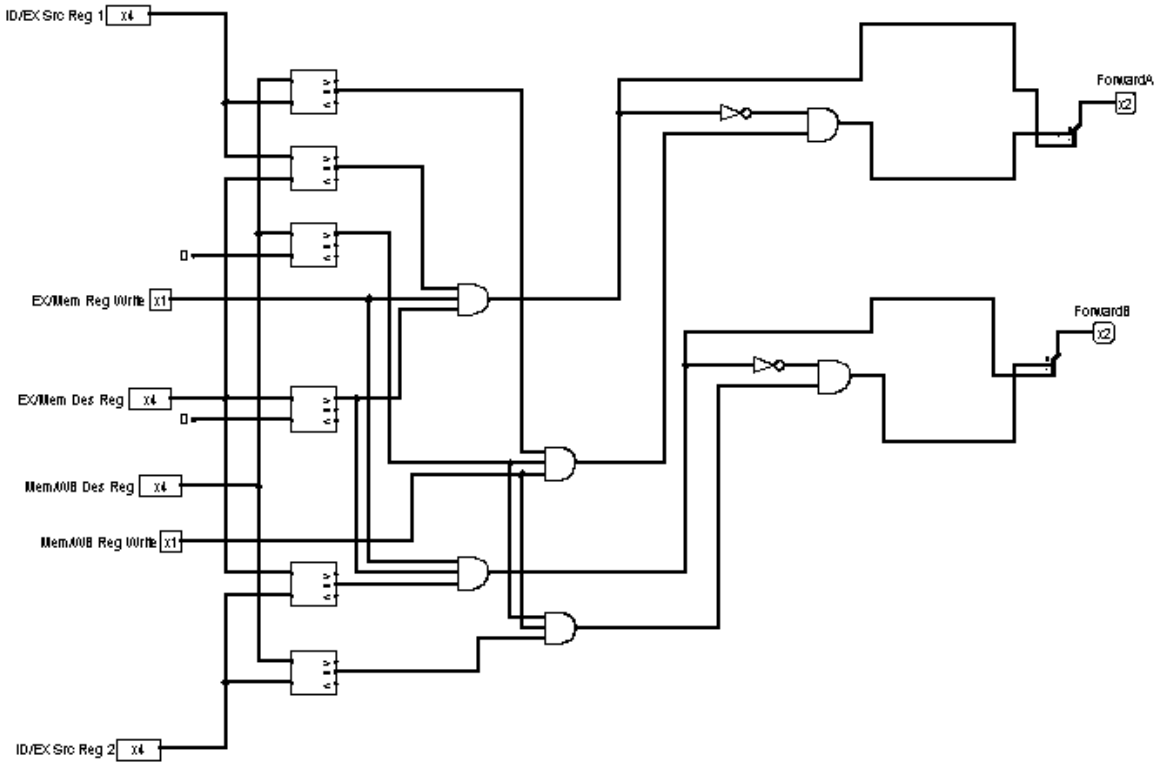


Figure 11: Forwarding Unit

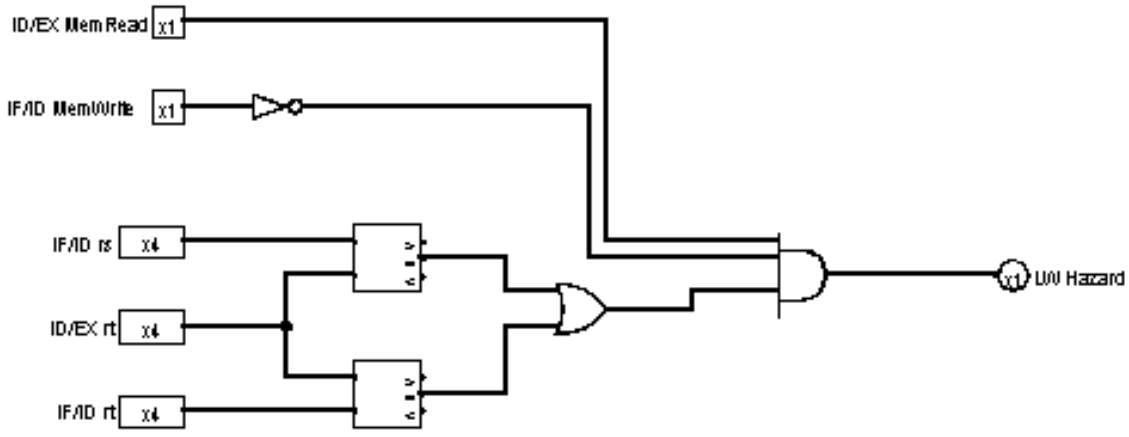


Figure 12: Load Use Hazard Detector

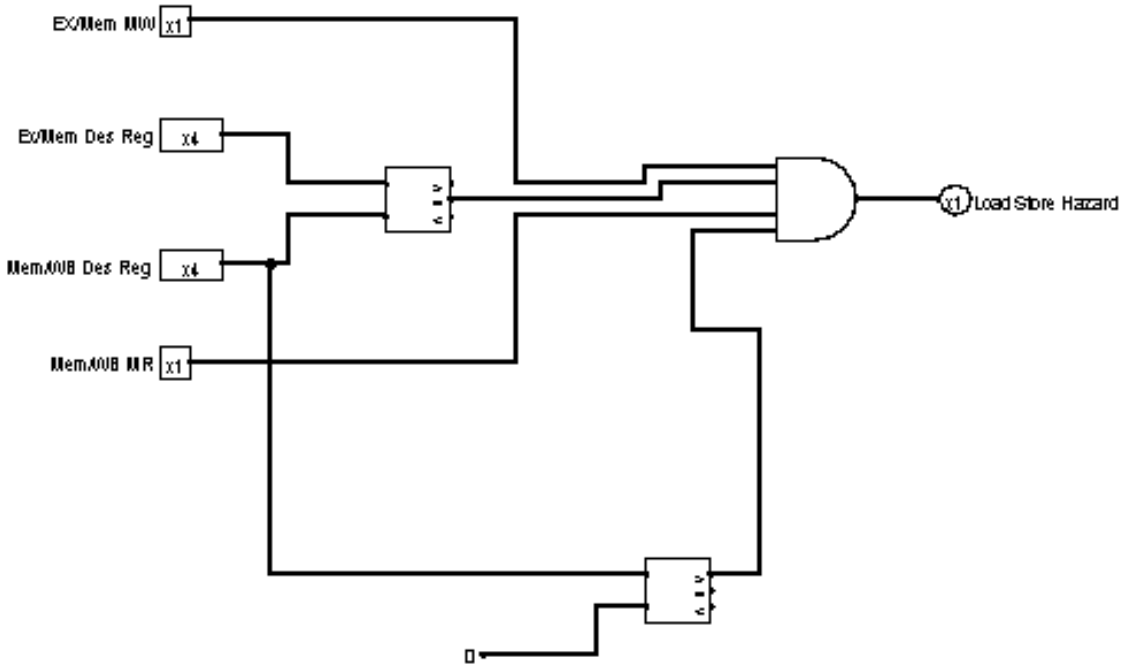


Figure 13: Load Store Hazard Detector

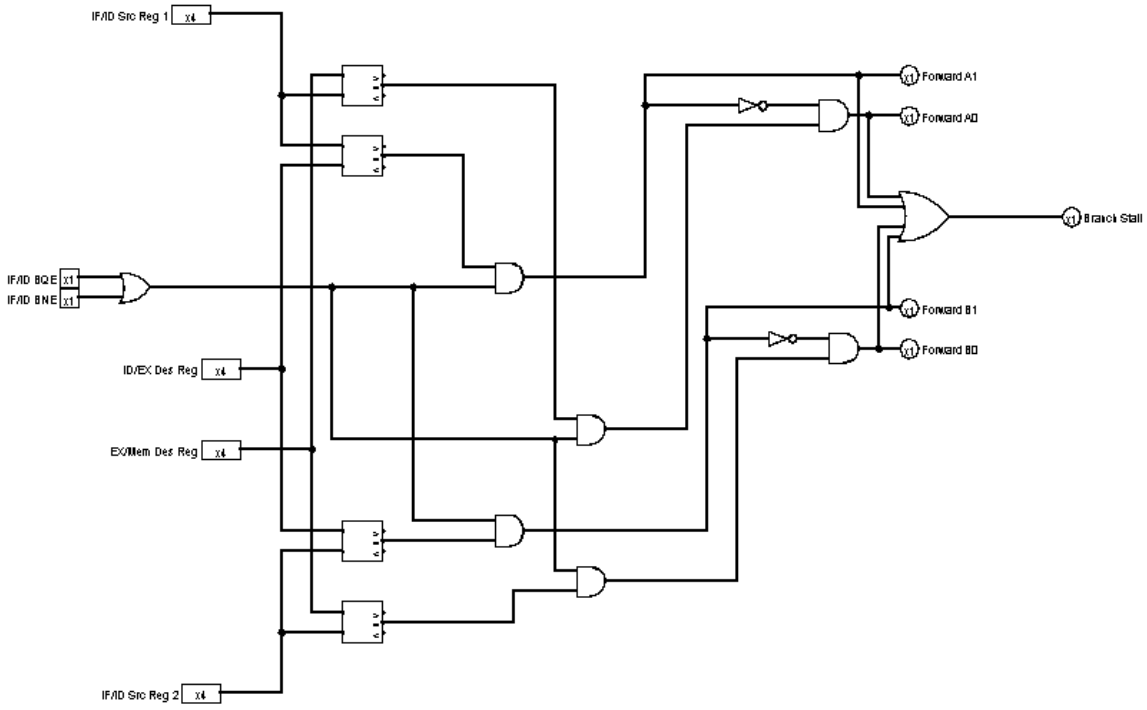
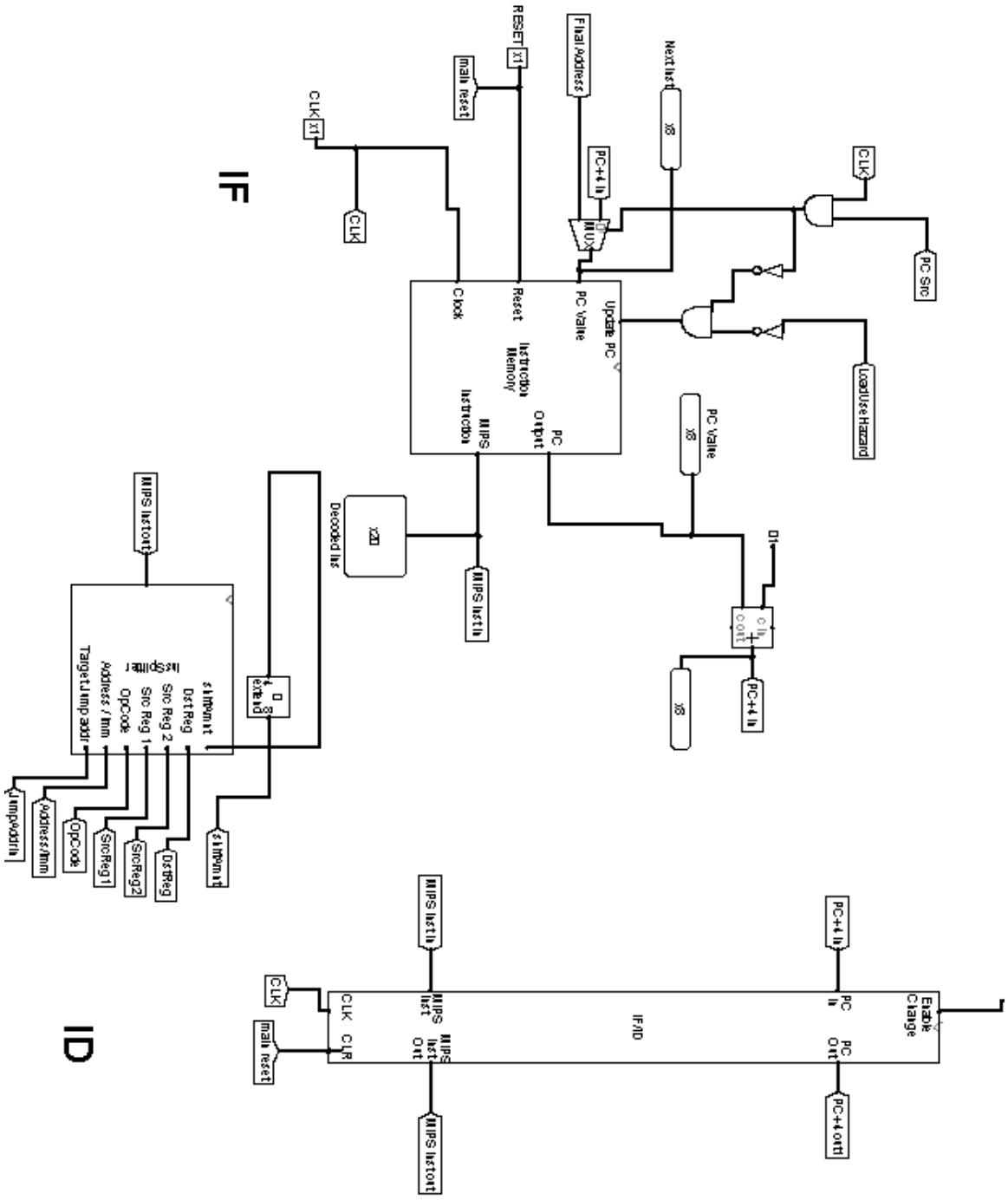
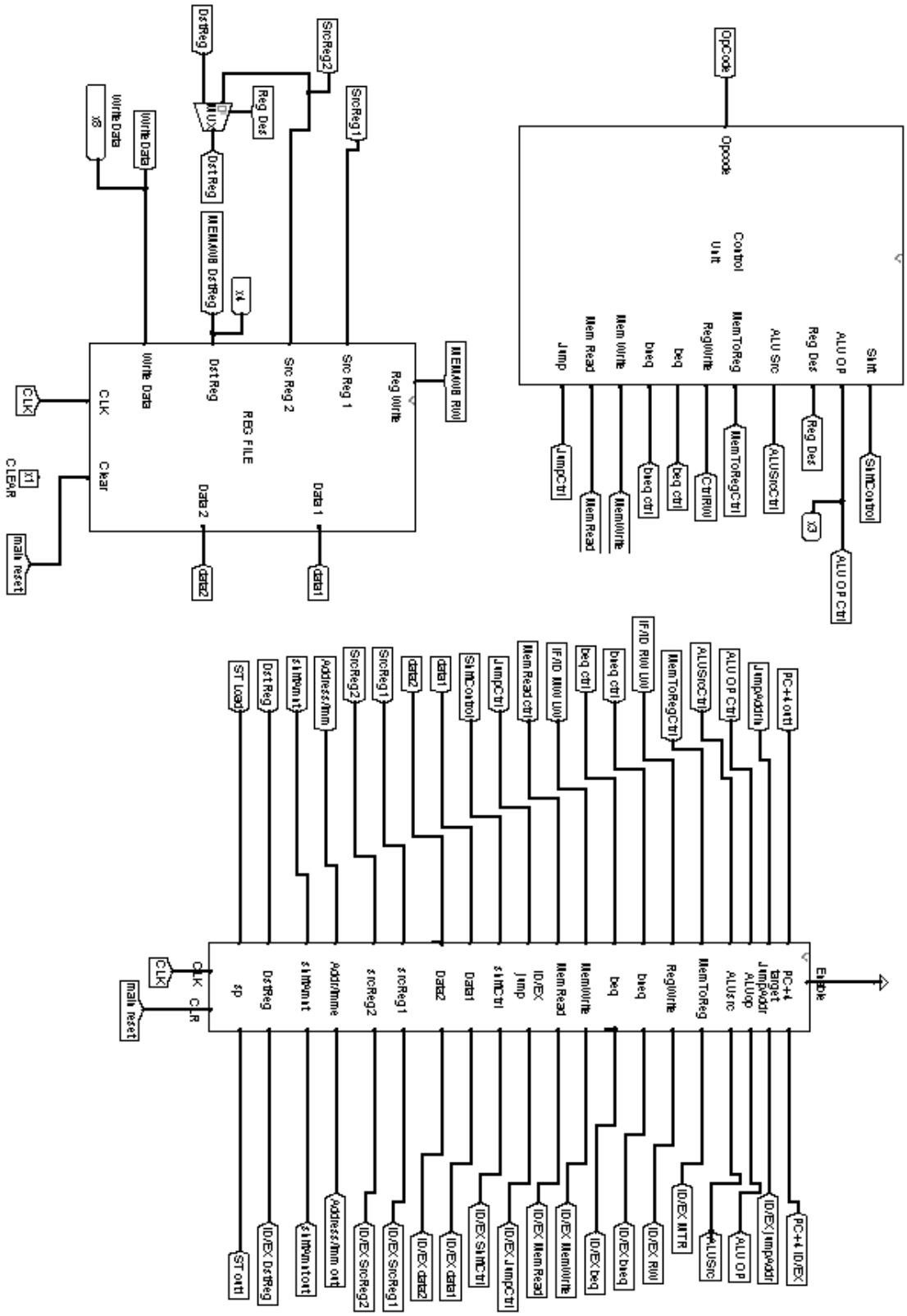
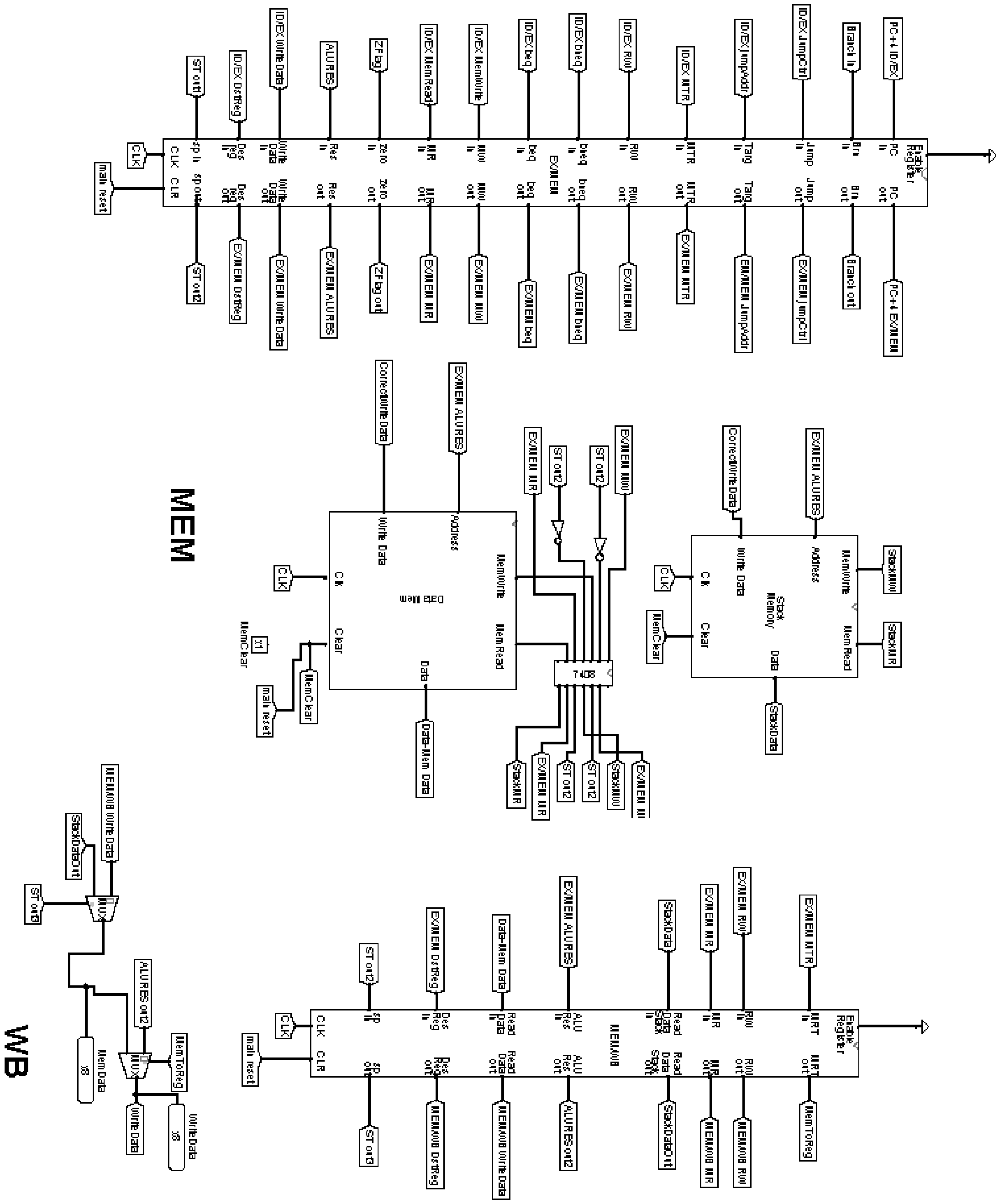


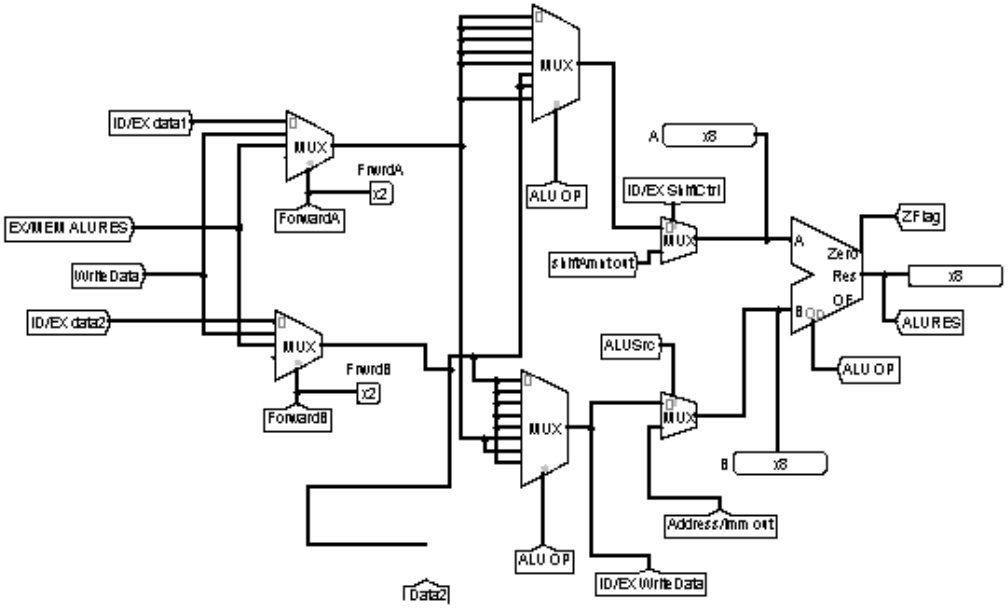
Figure 14: Branch Hazard Detector

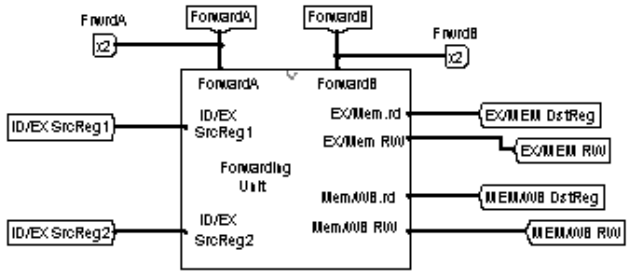
Main Circuit

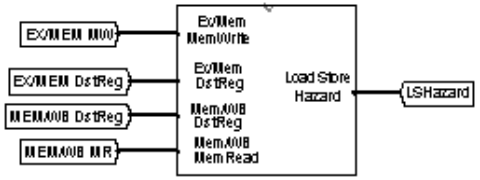


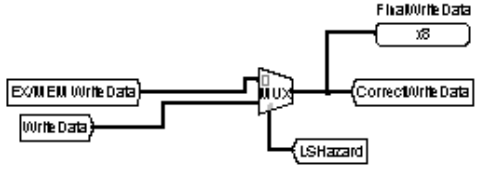

OpCode
Opcode
Control Unit
Shft
ALU OP
Reg Des
ALU Src
MemToReg
RegWrite
beq
bneq
Mem Write
Mem Read
Jump
Shft Control
Reg Des
x3
ALU OP Ctrl
ALUSrcCtrl
MemToRegCtrl
CtrlRW
beq ctrl
bneq ctrl
MemWrite
MemRead
JumpCtrl
DstReg
SrcReg2
SrcReg1
Reg Des
MUX
DstReg
MEM/WB DstReg
x4
Write Data
x8
REG FILE
Write Data
DstReg
Src Reg 2
Src Reg 1
Reg Write
MEM/WB RW
CLK
Clear
CLK
x1
CLEAR
main reset
data2
data1
EX
Enable
PC++ ctrl
Jumpaddrs
ALU OP Ctrl
ALUSrcCtrl
MemToRegCtrl
IF/ID RW Ctrl
beq ctrl
IF/ID MW Ctrl
MemRead ctrl
JumpCtrl
Shft Control
data1
data2
SrcReg1
SrcReg2
Address/Imm
shftImmt
DstReg
ST Load
PC++
Target
JumpAddr
ALUop
ALUsrc
MemToReg
RegWrite
beq
bneq
MemWrite
ID/EX
Jump
shftCtrl
Data1
Data2
srcReg1
srcReg2
Addr/Imme
shftImmt
sp
DstReg
CLK
CLR
main reset
PC++ ID/EX
ID/EX JumpAddr
ALU OP
ALUSrc
ID/EX MTR
ID/EX RW
ID/EX beq
ID/EX bneq
ID/EX MemWrite
ID/EX MemRead
ID/EX JumpCtrl
ID/EX ShftCtrl
ID/EX data1
ID/EX data2
ID/EX SrcReg1
ID/EX SrcReg2
Address/Imm ext
shftImmtext
ST ctrl
ID/EX DstReg

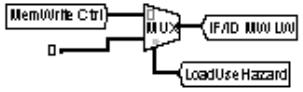

MEM
WB
EX/MEM
MEM/WB
Data Mem
Stack Memory
Enable Register

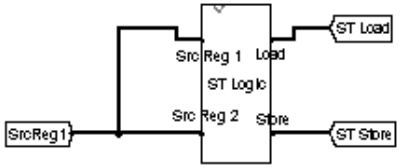

MUX
ID/EX data1
MUX
FrwrdA
x2
ForwardA
EX/MEM ALURES
Write Data
ID/EX data2
MUX
FrwrdB
x2
ForwardB
MUX
ALU OP
A
x8
ID/EX ShiftCtrl
MUX
shiftAmtout
A
Zero
Res
OF
B
ZFlag
x8
ALURES
ALU OP
ALUSrc
MUX
B
x8
Address/Imm out
ALU OP
ID/EX Write Data
Data2

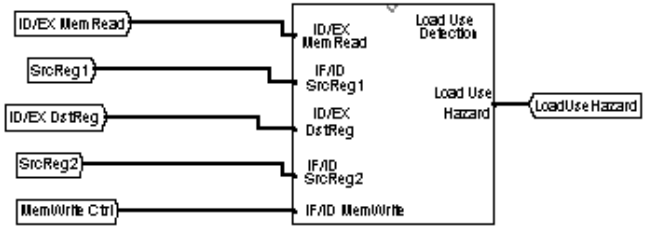

FrwrdA
x2
ForwardA
ForwardB
FrwrdB
x2
ForwardA
ForwardB
ID/EX SrcReg1
ID/EX SrcReg1
EX/Mem.rd
EX/Mem RW
Forwarding Unit
MemA/WB.rd
MemA/WB RW
ID/EX SrcReg2
ID/EX SrcReg2
EX/MEM DstReg
EX/MEM RW
MEMA/WB DstReg
MEMA/WB RW

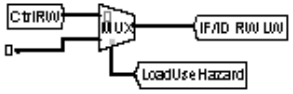

EX/MEM MW
Ex/Mem MemWrite
EX/MEM DstReg
Ex/Mem DstReg
Load Store Hazard
LSHazard
MEMA/WB DstReg
MemA/WB DstReg
MEMA/WB MR
MemA/WB MemRead


FinalWrite Data
x8
EX/MEM Write Data
MUX
CorrectWrite Data
Write Data
LSHazard


MemWrite Ctrl
MUX
IF/ID MW LW
0
LoadUse Hazard


SrcReg 1
Load
ST Logic
Src Reg 2
Store
SrcReg1
ST Load
ST Store


ID/EX Mem Read
Load Use Detection
ID/EX MemRead
SrcReg1
IF/ID SrcReg1
Load Use Hazard
ID/EX DstReg
ID/EX DstReg
SrcReg2
IF/ID SrcReg2
MemWrite Ctrl
IF/ID MemWrite
LoadUse Hazard


CtrlRW
MUX
IF/ID RW LW
0
LoadUse Hazard

4 How to Write and Execute A Program

Write the assembly code without any empty lines in a text file named `in.txt`. Then, run the `assembler.cpp` code to generate `instructions.hex`. This file will be loaded into the instruction memory. Afterward, providing a clock signal will execute the program.

5 ICs used with their count

Gate	Gate Count	IC	IC Count
MUX(2-to-1)	8	74157	2
MUX(4-to-1)	3	74153	2
MUX(8-to-1)	2	74151	1
MUX(16-to-1)	2	74150	2
Adder(8-bit)	3	7483	6
Decoder(4-to-16)	1	74154	1
AND(2-bit)	10	7408	3
AND(8-bit)	1	7430	1
OR(2-bit)	4	7432	1
OR(8-bit)	1	7432	1
NOT(1-bit)	1	7404	1
Priority Encoder(2-bit)	1	Priority Encoder(2-bit)	1
Register(8-bit)	33	Register(8-bit)	33
ROM(256X20)	1	ROM(256X20)	1
ROM(256X8)	1	ROM(256X8)	1
ROM(16X13)	1	ROM(16X13)	1
Subtractor(8-bit)	1	Subtractor(8-bit)	1
Comparator(8-bit)	8	Comparator(8-bit)	8
RAM(256X8)	1	RAM(256X8)	1
Clock	1	Clock	1

6 Contribution

Instruction Memory: 2105009

Register File: 2105007, 2105009

Control Unit: 2105008, 2105009

Data Memory: 2105007, 2105008

IF/ID, ID/EX: 2105007, 2105009

EX/MEM, MEM/WB: 2105007, 2105008

Load-use, Load-store, Branch-hazard detector: 2105007

Main circuit: 2105007, 2105008, 2105009

Report: 2105007

7 Discussion

In the process of designing and implementing the system, we took several careful measures to ensure correctness, efficiency, and ease of debugging. One critical aspect was creating the table for control bits with precision, as any inaccuracies in this table could lead to unexpected outputs. To streamline the design process and minimize wiring errors, we utilized tunnels instead of physical wiring, which not only reduced clutter but also made debugging significantly easier.

Before implementing the circuit, we prepared a hand-drawn diagram to avoid potential faulty connections. This step helped us identify and resolve design flaws early. Additionally, we ensured that the instruction hex file was formatted correctly to prevent errors when loading MIPS instructions into the ROM.

To manage complexity and enhance implementation, we modularized each circuit component into separate `.circ` files. This modular approach facilitated independent testing and debugging of individual components. Moreover, we optimized the circuit by removing unnecessary components and reducing register usage, resulting in a more efficient design.

Finally, we invested considerable time in carefully cross-checking corner cases for each sample test case. This thorough verification process ensured the system's reliability and robustness under various scenarios.