Pipelined MIPS16 Processor Design

Mazen Amria* and Mayar Abuzahra†
Department of Electrical and Computer Engineeering,
Birzeit University
Ramallah, Palestine
Email: {*t.mazenamria,†mayarabuzahra}@gmail.com

***Abstract*—This project aims to design, implement and verify a simple 16-bit MIPS architecture based 'Pipelined RISC Processor'. The datapath of the processor was designed and decisions have been made in order to achieve better and more convenient functionality while not breaking the main standards of MIPS processors. Implementation of the components has been optimized according to multiple aspects: Performance, Cost, Simplicity and Portability. Each component has been verified using unit tests to verify the logical functionality of the component itself, and the whole processor has been verified by executing set of supported instructions and observing results.**

***Index Terms*—MIPS16, MIPS, Pipelined, Processor, Architecture.**

I. Introduction

MIPS architecture is a *Reduced Instruction Set Computer (RISC)* architecture that has been designed by *MIPS Technologies*, the main architecture of *MIPS* is *MIPS32* which is 32-bit architecture processor. *MIPS* processors are implementing the Pipelined Processing technique, which allows to process multiple instructions simultaneously on the same processor. In this project a 16-bit architecture is to be designed.
Designing an architecture starts by defining the functionality and requirements of that architecture, that is to define the *Instruction Set Architecture (ISA)*, in order to do that, the project starts by defining the set of operations needed to perform any type of functionality. Then defining the instruction formats, and finally defining the full *ISA*. After that the *Datapath* is implemented to support that *ISA*, taking in consideration the possible hazards in the design and solving them.

A. Set Of Operations

Needed operations can be classified by their importance, some of them are essentials, and some are less important. A full ISA need to consist of logical, arithmetic, branching and memory operations. In this project the main aim is to implement a full ISA, meaning that whatever the operation is, it can be performed on top of this processor, either directly from the supported instructions or by implementing software solutions on top of the supported instructions. After deciding the essential instructions, other instructions will be implemented in hardware if possible, depending on the complexity of the instruction and the percentage of use, meaning that complex instructions that need many instructions to perform in software and common instructions that are used widely are more likely to be implemented in hardware, in order to achieve the maximum performance.

1) Logical Operations: It can be proved that any logical operation can be inferred from either bitwise AND or bitwise OR and the bitwise NOT. Thus at least two of these should be implemented, and the rest can be implemented in the software algorithms if needed. In this project `AND`, `OR` and `NOT` operations are all implemented, this will achieve more performance than implementing two, since inferring logical operations in software require more instructions.
Another type of logical operations is Shift operations, although it might be inferred from some arithmetic operations but this requires huge amount of instructions unlike implementing them in hardware, which is considered to be one of the fastest operations in hardware. In this project both `SLL` (shift left logical) and `SRL` (shift right logical) are implemented.

2) Arithmetic Operations: For the arithmetic operations, the following set of operations is needed: `ADD`, `SUB`, `MUL`, `DIV`, `MOD` and the FPR version of each operation. For `MUL`, `DIV` and `MOD` these can be implemented in software using `ADD`, `SUB`, `SLL` and `SRL`. Also all the FPR operations can be implemented in software using the same mentioned operations and some branching operations.
Check Booth's Algorithm [1] as an example of implementing such operations in software on top of essential arithmetic and logical operations.
Also a special arithmetic operation is added instead of implementing it in software, that is `CAS` which returns the maximum number between two numbers, implementing it in hardware will increase the performance, since it's a widely used operation and this will reduce the number of needed instruction for this operation.

3) Branch Operations: In this class, two types of branches are needed: Unconditional Branch and Conditional Branch. For the Unconditional Branch, jump operations are needed `J` and `JR` for jumping to a location defined in the run-time, and for the Conditional Branch, a branch if equal operation `BEQ` is enough to cover both branch if equal and branch if not equal. Branching on less or greater results is achieved by making use of `SLT` which sets a register to 1 when first operand is less than second operand, and we can then check the result register with

`BEQ`.

4) Memory Operations: Load and Store operations need to be implemented in order to deal with the memory. As the memory is byte-addressable and the size of the processor is 16-bits, hence two access modes are needed: accessing one byte or accessing two bytes. Hence the following operations are needed: `LW` and `SW` to load and store a word (i.e. two bytes). `LB` and `SB` to load and store a byte. Also `LBU` can be added to load the byte and write it to the most significant 8 bits in the destination. No `SBU` is needed since the memory—destination in this case—is byte addressable.

B. Instruction Format

MIPS32 consists of three instruction formats: R-type, I-type and J-Type.

1) R-type Format: R-type Format is identified by `OPCODE = 000000`, and the rest of the format is interpreted according to the following [2]:

31 26	25 21	20 16	15 11	10 6	5 0
000000	R_S	R_T	R_D	SA	FUNC
6	5	5	5	5	6

R-type format is used to implement operations that uses only registers, in the project, the size of the instruction is limited to 16-bits, hence the fields need to be resized, and some fields may be removed. By listing the operations that only deals with registers, some decisions may be made for the instruction format.

- `AND`
- `OR`
- `ADD`
- `SUB`
- `CAS`
- `JR`
- `JALR`
- `SLT`
- `LWS`

As both `JR` and `JALR` works in similar flow and does not require more than one register, so the other register fields can be used to identify whether it's `JR` or `JALR`. So a `FUNC` with size of 3 bits can be enough for the mentioned instructions. No `SA` field is needed as the shift operations wouldn't be implemented using registers—immediate shift amount could be enough for performing shift on a register and storing the result in another register, hence no need to implement it in R-type—. For `OPCODE` field, the size should be enough to allow all supported opcodes, but it shouldn't be much larger in order to leave more space for the operands, in this case, three operands are identified by three register fields, all these fields need to be the same size. Having 4-bit register field will leave only 1-bit `OPCODE` which isn't enough, and enlarging the `OPCODE` to exceed 4-bits will force the register fields to be 2-bits maximum (i.e. 4 unique registers only), thus 4-bit `OPCODE` and 3-bit register fields will allow for 15 different instructions—other than the 8 R-type instructions—and 8 unique registers. So the format of the R-type in 16-bit will be as following

15 12	11 9	8 6	5 3	2 0
0000	R_S	R_T	R_D	FUNC
4	3	3	3	3

2) I-type Format: This format is used for the operations that uses immediate value for one of the operands. And it takes the following format [2]:

31 26	25 21	20 16	15 0
OPCODE	R_S	R_T	Immediate
6	5	5	16

the following instructions can be implemented in I-type format and still have a full capable ISA:

- `ANDI`
- `ORI`
- `NOT`
- `SLL`
- `SRL`
- `ADDI`
- `LW`
- `LB`
- `LBU`
- `SW`
- `SB`
- `BEQ`

`NOT` does not require `Immediate` value because it's unary operation, but this means also that it only requires source and destination register, and no need for third register, so implementing it as I-type would be better since implementing it in R-type would require larger `FUNC` and hence less `OPCODE` or register fields.

15 12	11 9	8 6	5 0
OPCODE	R_S	R_T	Immediate
4	3	3	6

3) J-type Format: This format is used for the operations that uses immediate value only without any other operands, like `J`. And it takes the following format [2]:

31 26	25 0
OPCODE	Immediate
6	26

the following instructions can be implemented in J-type format:

- `J`
- `JAL`
- `LUI`

As only `OPCODE` and `Immediate` are needed, and `OPCODE` has the size 4 bits, the rest are all assigned as the `Immediate` field, and the J-type format becomes:

15 12	11 0
OPCODE	Immediate
4	12

C. *Full ISA*

TABLE I
THE DESIGNED INSTRUCTION SET ARCHITECTURE

Mnemonic	**Description**	**Encoding**				
AND	REG(R_D) := REG(R_S) & REG(R_T)	OP = 0000	R_S	R_T	R_D	F = 111
OR	REG(R_D) := REG(R_S) | REG(R_T)	OP = 0000	R_S	R_T	R_D	F = 110
CAS	REG(R_D) := MAX[REG(R_S), REG(R_T)]	OP = 0000	R_S	R_T	R_D	F = 101
LWS	REG(R_D) := MEM[REG(R_S) + REG(R_T)]	OP = 0000	R_S	R_T	R_D	F = 100
ADD	REG(R_D) := REG(R_S) + REG(R_T)	OP = 0000	R_S	R_T	R_D	F = 011
SUB	REG(R_D) := REG(R_S) - REG(R_T)	OP = 0000	R_S	R_T	R_D	F = 010
SLT	REG(R_D) := [REG(R_S) < REG(R_T)] ? 1 : 0	OP = 0000	R_S	R_T	R_D	F = 001
JR	PC := REG(R_S)	OP = 0000	R_S	000	000	F = 000
JALR	REG(7) := PC, PC := REG(R_S)	OP = 0000	R_S	001	000	F = 000
ANDI	REG(R_T) := REG(R_S) & Immediate	OP = 0001	R_S	R_T	Immediate	
ORI	REG(R_T) := REG(R_S) | Immediate	OP = 0010	R_S	R_T	Immediate	
ADDI	REG(R_T) := REG(R_S) + Immediate	OP = 0011	R_S	R_T	Immediate	
LW	REG(R_T) := MEM[REG(R_S) + Immediate]	OP = 0100	R_S	R_T	Immediate	
LBU	REG(R_T)[15:8] := MEM[REG(R_S) + Immediate]	OP = 0101	R_S	R_T	Immediate	
LB	REG(R_T)[7:0] := MEM[REG(R_S) + Immediate]	OP = 0110	R_S	R_T	Immediate	
SW	MEM[REG(R_S) + Immediate] := REG(R_T)	OP = 0111	R_S	R_T	Immediate	
SB	MEM[REG(R_S) + Immediate] := REG(R_T)[7:0]	OP = 1000	R_S	R_T	Immediate	
BEQ	PC += (REG(R_T) == REG(R_S)) ? Immediate : 0	OP = 1001	R_S	R_T	Immediate	
NOT	REG(R_T) := ∼REG(R_S)	OP = 1101	R_S	R_T		
SLL	REG(R_T) := REG(R_S) << Immediate	OP = 1110	R_S	R_T	Immediate	
SRL	REG(R_T) := REG(R_S) >> Immediate	OP = 1111	R_S	R_T	Immediate	
J	PC := {PC[15:12], Immediate}	OP = 1010	Immediate			
JAL	REG(7) := PC, PC := {PC[15:12], Immediate}	OP = 1011	Immediate			
LUI	REG(1) := Immediate << 4	OP = 1100	Immediate			

The full ISA is shown in Table I, showing the description of each instruction and the encoding in binary.

II. Pipelining

The main Idea of Pipelining is to divide the datapath of all instructions into isolated stages, meaning that each stage will use its own hardware and signals and no other stage will use them. In MIPS architecture, the instruction stages are:

- Instruction Fetch (`IF`): At this stage the instruction stored at the address stored in `PC` register is fetched from memory and brought into the CPU.
- Instruction Decode (`ID`): At this stage the instruction is decoded, and the control signals are generated according to the decode of the instruction to control the flow of that instruction. Register values are read at that stage also.
- Execution (`EX`): At this stage any arithmetic or logical operations are executed in the ALU.
- Memory (`MEM`): At this stage any read/write operation from/to memory is performed.
- Write Back (`WB`): At this stage the result of the instruction might be stored in the destination register.

Also some stages may require the same hardware, but that is solved and described in Section III.

III. Components

A. Register File

Consists of 8 registers (`R0` to `R7`) each of 16-bit size, the first register `R0` is hardwired to the zero value. In order to obey the isolation principle mentioned at Section II the register file consists of 3 different buses each with it's own register selection, two of them for reading operations and used at `ID` stage, the third bus is for writing operations and used only by the `WB` stage. The following figure shows the implementation of the Register file.

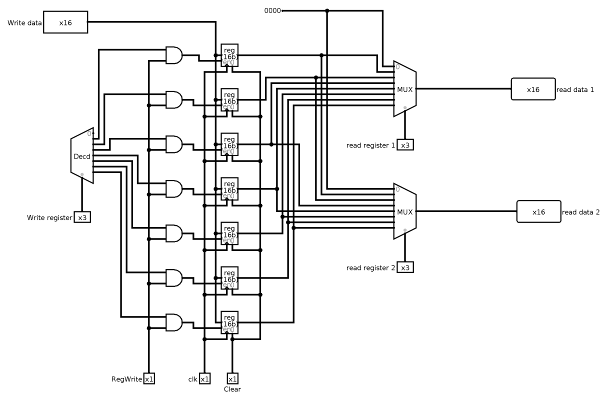

Fig. 1. Register File

B. Memory Components

In the Pipelined architecture, the instruction memory and the data memory—later will be referred to as `IMEM` and `DMEM`—are separated, since memory can be accessed once at the cycle and the Pipeline contains interleaving instructions at different stages, so at the same cycle one instruction might be fetching the instruction from the memory and the other instruction is loading/storing some data from/to the memory, so in order to obey the isolation principle mentioned in Section II they need to be separated into different memory elements.

`IMEM` is used for reading only, and some references consider it to be Read Only Memory (`ROM`) but in this project it's considered to be Random Access Memory (`RAM`) with read and write capabilities, however it's used in the final datapath as reading only memory, but still have the capability for writing, this makes the CPU easily configurable, allows for using test benches circuits, and allow for extending the project in the future to include an instruction writing datapath (as in any general purpose CPU). Also since the instructions are all with a fixed size, the memory can only read/write words (2 Bytes) but it still byte-addressable memory, it's meant to be byte-addressable to match the addressing format of the `DMEM`, and hence, make it easier for compilers and operating systems to deal with addresses.

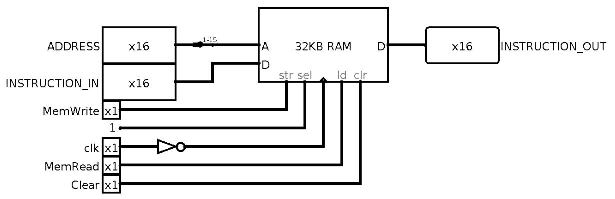

Fig. 2. Instruction Memory

`DMEM` was built using $2 \times 32KB$ `RAM` each of them is byte-addressable. A `BYTE MIRROR` component is used to reflect the bytes in both store or load operation depending on the endainness of the datapath, also a `BYTE SELECTOR` is used when the load operation requires only one byte, so it clears out the other byte. All these components are controlled according to `DMEM CONTROL`.

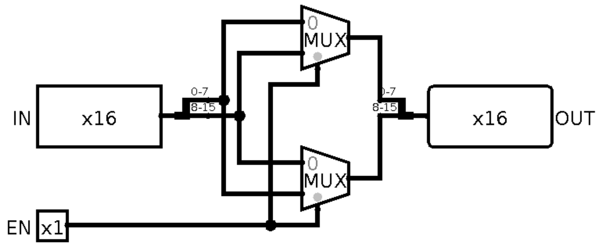

Fig. 3. Bytes Mirror

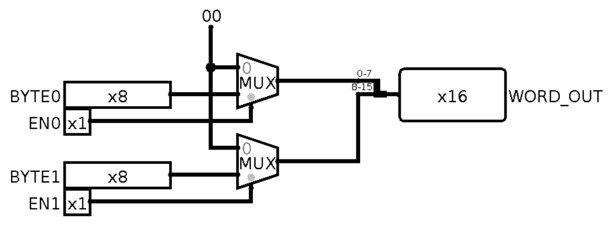

Fig. 4. Byte Selector

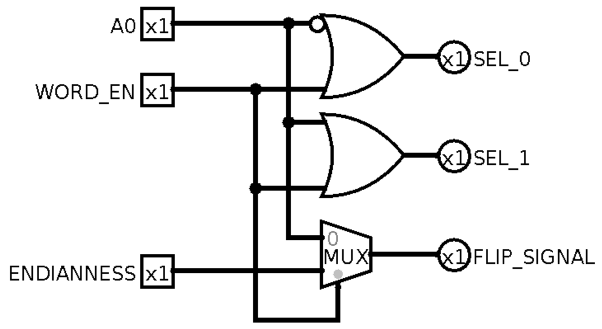

Fig. 5. DMEM Control

DMEM deals with the endianness by itself, it expects the desired word to be in DATA_IN, and returns the word at DATA_OUT, in case of single byte mode it uses the lower byte of both terminals.

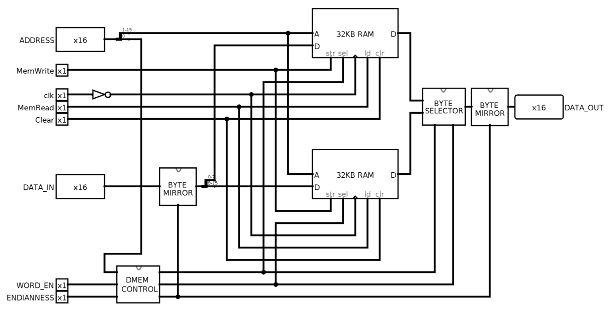

Fig. 6. DMEM Control

C. Arithmetic and Logical Unit

ALU contains 8 functional units, and can perform 8 types of operations listed below and sorted by codes from 000 to 111:

- AND: Bitwise logical AND between the operands.
- OR: Bitwise logical OR between the operands.
- ADD: Adding two operands without input carry.
- SUB: Subtracting two operands without input borrow.
- CMP: Comparing two operands and set the flags accordingly.
- NOT: Bitwise logical NOT operation for the first operand.
- SLL: Logical Left Shift of the first operand with shift amount stored at the second operand.
- SLR: Logical Right Shift of the first operand with shift amount stored at the second operand.

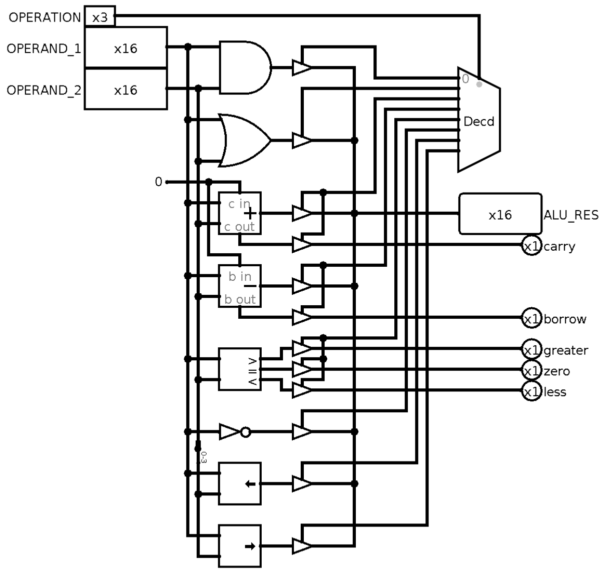

Fig. 7. Arithmetic and Logical Unit

D. Control Unit

Control Unit is used to generate the signals that will control the functional units in the datapath, each functional unit can be used by multiple instructions in the ISA, so all these instruction must be decoded to generate the signal that controls the behaviour if that functional unit. The following list contains the needed control signals:

- RegWrite: used by the instructions which will write the result to the register, and it will enable the writing functionlaity in the register file.
- RegDest: is used to select the destination register, some instructions are using RD field to be the destination register, and other instructions are using RT as the destination register, where some instructions have pre-defined destination register, like R1 for LUI and R7 for LINK.
- MemRead: used by the load operations to enable the read functionality in DMEM.
- MemWrite: used by the store operations to enable the write functionality in DMEM.
- ALUOP: defines the operation to be performed in the ALU.
- JUMP: indicates that the current instruction is a jump instruction.
- LINK: indicates that the current jump instruction would store the return address at R7.
- OP1_IMM12: indicates that the value of the first operand is taken from 12-bit immediate.
- OP2_IMM6: indicates that the value of the second operand is taken from 6-bit immediate.

- `PCSrc`: selects the source of the next `PC` whether it's incremented as usual of it's the exception handling address or it's the jump target or the branch target.
- `WORD_EN`: indicates that `DMEM` will work in word mode or byte mode.

In case of stall cycles, the control unit will be zeros to disable all functionalities, and in case of flushing the pipeline all signal will be cleared and the instruction will be discarded. The following figure shows the implementation of the control unit circuit.

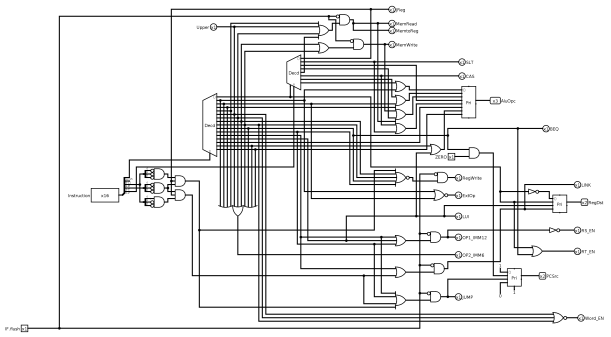

Fig. 8. Control Unit

E. Forwarding Unit

Hazards occurs in the Pipelined processors if dependency occurred between instructions, since instructions are performed concurrently, in case of hazards, the pipeline can be stalled (stopped temporarily until the independent instruction completes execution) and this is performed in the Hazard Detection Unit, or if the data is valid in the independent instruction it can be forwarded to the dependent instruction using the forwarding unit. The following list explains at each stage which instructions will get the result ready after that stage:

- `ID`: Only jump and branch instructions will finish execution (or get the data ready) after the `ID` stage, instructions with `LINK` functionality are the only instructions that would write to the register file. The value of `RD` can be forwarded from the `ID/EX` output to the `ID` if needed, but since it's a jump instruction it will always be followed by a stall cycle, so when the next instruction is at `ID` it's obvious that the `LINK` is at `MEM` stage, and hence no need for forwarding from `ID/EX` output to `ID`.
- `EX`: Arithmetic and Logical instructions will get the data ready after that stage. Some instructions may require that data at the `ID` stage or the `EX` stage, so the data need to be forwarded from `EX/MEM` output to `ID` and `EX` if it's valid at the forward source (`RegWrite` is high and `MemRead` is low) and it's required at the forward destination (`ID.Rs == EX/MEM.Rd`, `ID.Rt == EX/MEM.Rd`, `EX.Rs == EX/MEM.Rd` or `EX.Rt == EX/MEM.Rd`).
- `MEM`: Load operations will get the data ready after this stage, and it might be required at all stages (`ID`, `EX` or `MEM`) in most references, there's no forwarding from `MEM/WB` output to `MEM`, this forwarding handles only the load followed by store with stored data dependency, since store has two types of dependencies on load, it might require the result of the load for the address calculations and this cannot be forwarded, but also it might require it as the data to be stored, this case has been solved with forwarding instead of stalling the pipeline.

The following circuit shows the implementation of the forwarding unit.

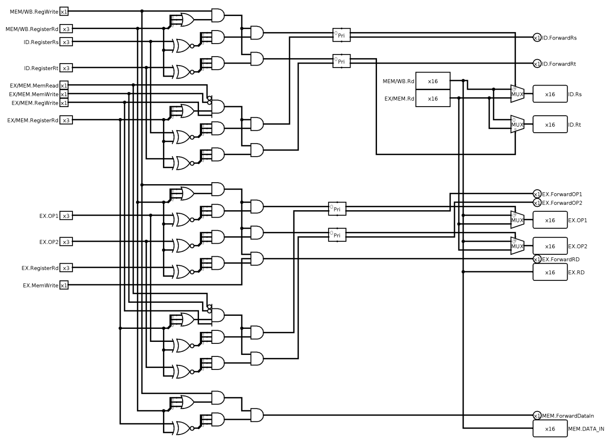

Fig. 9. Forwarding Unit

F. Hazard Detection Unit

Data and Control hazards might occur in the pipelined processor, hence, these hazards need to be solved. Some cases of the data hazards have been solved in the Forwarding Unit, but the rest cannot be solved without stalling the pipeline. When the register value is being modified by a load operation and this operation is followed (directly) by an instruction that requires the same register in `EX`, then the forwarding cannot solve the problem unless the next instruction is being delayed by single cycle, one exception is the load operation followed by a store operation that requires the register value for the data to be stored and not for the address calculation, this case it has been solved in the Forwarding Unit. Also if the load instruction is followed (directly) by a jump or branch instruction, it has to be delayed by two cycles in order for the data to be forwarded, if the dependent instruction isn't directly after the load, then it might be delayed for one cycle.

Control Hazards occurs when the jump or branch operations are performed (jump or branch is taken), in this case the fetched instruction would be invalid, hence it must be flushed from the pipeline.

The following figure shows the implementation of the Hazard Detection Unit.

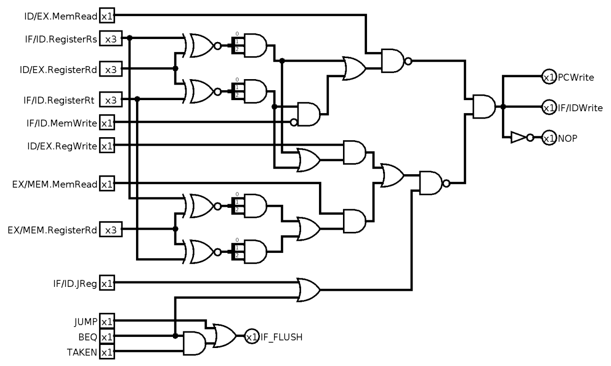

Fig. 10. Hazard Detection Unit

IV. Datapath

After building each component and testing its functionality, the final datapath has been built using the mentioned components, multiplexers has been used beside the control signals to control the flow of the data in the datapath. Stages has been isolated using registers to propagate the status of the instruction from one stage to another.

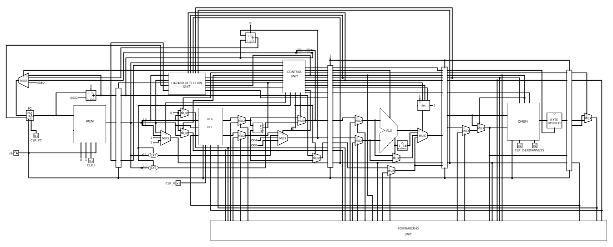

Fig. 11. Datapath

V. Results and Discussion

Testing the CPU includes two different stages:

A. Logic Verification

In this stage the logic and behaviour of the CPU is verified by applying a sequence of instructions and observe the accumulative output of that sequence.

```
addi $1, $0, 25
addi $2, $0, -19
addi $3, $0, 7
addi $4, $0, -22
addi $5, $0, 14
addi $6, $0, 2
addi $7, $0, -1
sw   $4, 0($0)
add  $4, $1, $2
lw   $3, 0($0)
addi $2, $3, 7
srl  $5, $5, 1
sw   $2, 2($0)
sw   $5, 4($0)
cas  $7, $5, $7
```

Starting with DMEM and Register file full of zeros, the expected Register file after this sequence is:

```
R1: fff0
R2: fff1
R3: ffea
R4: 0006
R5: 0007
R6: 0002
R7: 0007
```

And the expected DMEM after applying the sequence is:

```
0000: ffea
0002: fff1
0004: 0007
```

Another sequence is applied to verify more instructions:

```
addi $2, $0, 2
addi $3, $0, 2
addi $4, $0, -3
sw   $4, 4($0)
sb   $2, 1($0)
lb   $4, 1($0)
lbu  $5, 1($0)
lws  $6, $2($3)
not  $7, $6
```

Starting with DMEM and Register file full of zeros, the expected Register file after this sequence is:

```
R2: 0002
R3: 0002
R4: 0002
R5: 0200
R6: fffd
R7: 0002
```

And the expected DMEM after applying the sequence is:

```
0000: xx02
0004: fffd
```

After applying the instruction sequences, the results matched the expectations. The HEX files containing the instruction sequences are attached with the project.

B. Flow Verification

Another sequence is applied to test the jump and branching functionalities.

```
addi $2, $0, 2
addi $6, $0, 4
LOOP:
beq  $2, $6, END_LOOP
addi $6, $6, -1
j LOOP
END_LOOP:
jalr $0
```

and the expected register file after applying it is:

```
R2 = 0002
R6 = 0002
R7 = 000e
```

And another sequence is applied to test backward branching:

```
addi $1, $0, 0
addi $2, $0, 5
addi $4, $0, 1
DO_WHILE:
addi $1, $1, 1
slt  $3, $1, $2
beq  $3, $4, DO_WHILE
```

and the expected register file after applying it is:

```
R1 = 0005
R2 = 0005
R3 = 0000
R4 = 0001
```

The results matched the expectations, and the HEX files are attached as well.
The previous results proof three main results:

- No logical or arithmetic errors occurred in the results, hence, the components are logically correct.
- No data hazards occurred, since data hazards will cause reading inappropriate values. But in this case none of that occurred, this means that the forwarding logic and the stalling functionality is working well.
- No control hazards occurred, and that has been proofed by the jump and branch instruction sequence.

VI. Conclusion and Future Work

The project has succeeded in designing a usable, convenient, portable, and high performance MIPS CPU with 16-bit architecture. **Performance** has been achieved by reducing the stall cycles as possible, and that has been observed in optimizing 'Store after Load' cases. **Portability and Simplicity** have been achieved by making single modules portable, isolated and abstracted from other modules, and making them more general and not specified for the project, like `IMEM` which can be extended for instruction writing. Also the CPU is designed to match most of OS and Compilers standards, that is by building a complete ISA, that can be extended by software solutions such as compiler to cover most operations in the high level languages. **Cost** has been optimized as possible by not using redundant hardware, or overusing hardware to optimize other aspects.

The CPU can be extended to include status registers that will help adding more modes to the CPU execution, for example, User Mode and Kernel Mode, that will help in building Kernels on top of that CPU and may allow for adding instructions writing datapath. Also the Exceptions and FPR co-processors can be added to the CPU.

References

[1] Booth's Algorithm. [Online]. Available: https://en.wikipedia.org/wiki/Booth's_multiplication_algorithm

[2] MIPS Technologies, Inc., *MIPS32™ Architecture For Programmers Volume II: The MIPS32™ Instruction Set*, MIPS Technologies, Inc., 1225 Charleston Road, Mountain View, CA 94043-1353, 2001.