ECE401 - COMPUTER ARCHITECTURE

PROJECT -1

PROJECT PARTNERS –

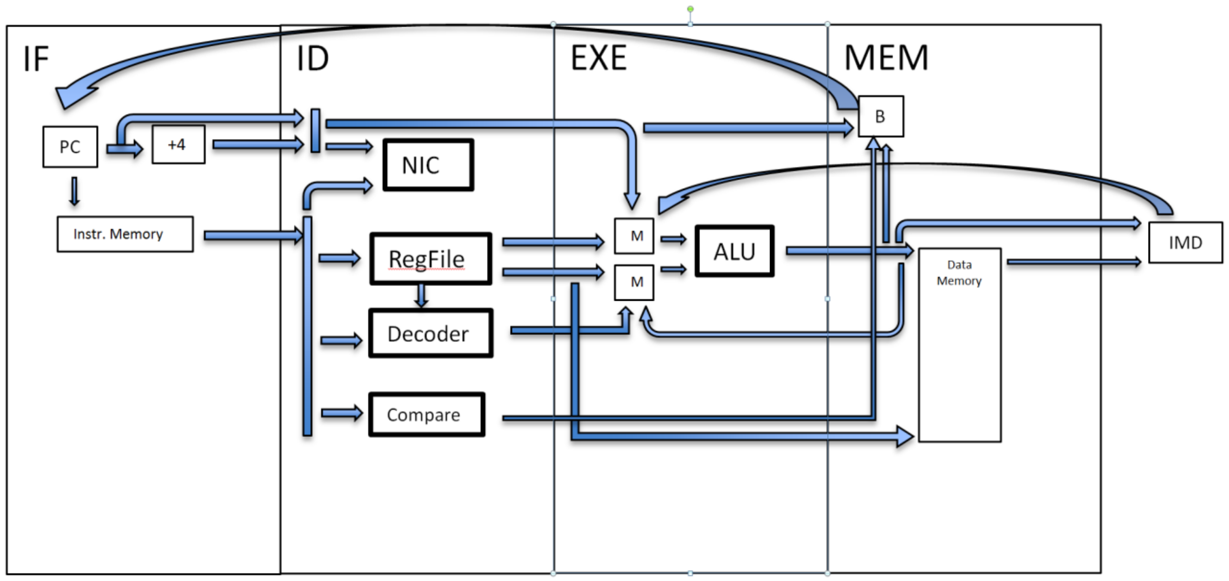
Chinmai

Clay Kardas

I. Introduction

A high level MIPS 5 stage pipelining was provided, which consists of Fetch, Decode, Execute, Memory and Write-back stage. However few of the code components weren't working well and hence involved us to scrutinizing the code in order to extract the component structure of the existing architecture, and adding the necessary components to enable it to run correctly. Flow Diagram below does not include bypassing logic.

MIPS Flow Diagram with Bypass and Forward Logic



II. Understanding MIPS stage and Implementation of necessary components

IF (Instruction Fetch):

This stage of the MIPS pipeline updates the Program Counter (PC) as per the instruction i.e if it is branch or jump PC is updated to the target address specified by the instruction otherwise it PC is incremented by 4 to fetch the next instruction.

Malfunction – In this stage PC was not getting incremented by 4. PC received alternate address for jump and branch instructions; however did not update PC accordingly *Implementation* – Checks if the instruction has to make a jump and loads the alternate address else loads PC + 4.

ID (Instruction Decode):

Data from instruction memory goes to NextInstructionCalculator (NIC) which determines which instruction address should be jumped/branched to next. RegFile processes writebacks from MEM and the instruction memory to determine which registers to address, use, and operands to send to the execute block.Decoder processes instruction memory and decodes opcode and determines the format, rt, rs rd, funct, memory operation, jump instruction etc. This stage also consists of a Compare and branch module which performs a comparison between operands A and B and determines if the branch has to be taken or not taken. The flag from this is sent to NIC to calculate the Alternate address.

Malfunction – NIC module does not check if the jump has to be performed Jump with register value or the immediate target address. Branch instructions were not updating Alternate_PC with the branch address.
Implementation-Checks if it is Jump instruction. If yes, then checks if this is a jump instruction has to load target address from register or immediate address and updates Alternate_PC accordingly. If this is not a Jump instruction then Alternate_PC updates it with a branch address.
Bypassing logic – Register and register data and write flags are received from MEM stage and EXE stage in order to select the new register data for the next instruction that will decoded.

a) Jump_reg is compared with the write registers of ID, MEM and EXE registers. If this register was used in the previous instruction (.i.e) if the latest value is in MEM or EXE stage the the rsval_jump1(target address) is updated with the value from MEM or the EXE stage.
b) Similar operation are performed for Operand A. Latest value is then sent to the EXE stage.
c) Similar operation are performed for Operand B.
d) Similar operation are performed for Write Registers instructions.
This comparison was achieved via following MUX implementation:

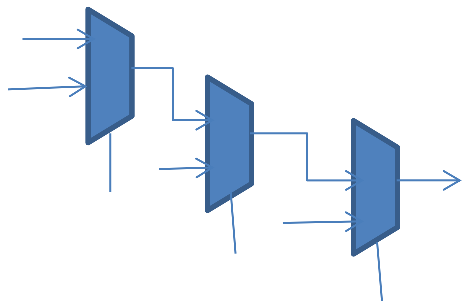


Fig 1 MUX3REG

EXE (Execute):

This stage of the MIPS pipeline receives ALU_control, operand A, Operand B, Instruction, PC, and send the operands to ALU and performs respective ALU operations depending upon the ALU_control. ALU_Result and Registers to be written and write_data is then passed to MEM stage.

Implementation:

Bypassing Logic-

The RegA and RegB was forwarded from ID stage to implement bypassing logic. Before performing ALU operation we ensure the operands values are latest values (.i.e) If any memory operations was performed before an ALU instruction. Hence we bypass the write register, write_data and a write flag from MEM stage to EXE stage.

a) RegA is compared with register from MEM stage and write register. We update the Data of Operand A with the latest value either from MEM stage or write_data depending on the comparison.

b) Similar operation are performed for Operand B.

c) Similar operation are performed for Write Registers instructions.

This comparison is achieved by the following MUX implementation.

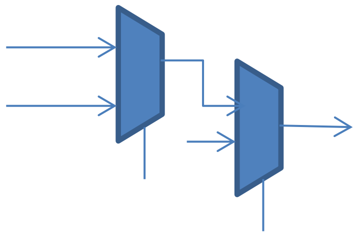


Fig 2 MUX2REG

MEM (Memory):

Store and Load operations are performed in this stage. ALU_result from the EXE stage provides the address from which the data has to be loaded. This stage write data from register or immediate data to memory for a store operation. For a load operation is loads data from memory into a register.

Malfunction – None of the load operations or the store operations were loading value from the memory or storing to the memory.

Implementation- MIPS memory is stacked as a 8bit data, however register size is 32 Bits. Hence to load data from memory and Store data into registers we need to check for the address and perform the load word, load half word , load byte depending on the address (last two bits of ALU_result received from EXE). MIPS-ISA.pdf was referred to implement the Load and Store Operations.

4	..			
0	8 Bits	8bits	8bits	8bits

$\Longrightarrow$

3	..
2	..
1	8 Bits
0	8 Bits

We have MUX to ensure we load the new value to the register.

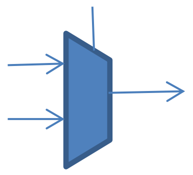


Fig 3 MUXREG

WB (Write-Back):
Sends the Register and Data to RegFile and Writes to the register.
Malfunction- RegFile was not writing into the registers.
Implementation – Checks if the data has to be written and writes the data to the respective register.

III. Experiments and Tests

We performed the tests present in tests folder. The processor was successful for pipetest, airth, branchtest. Hence tests were followed to perform noio tests. However code stopped working after 144 cycles. Bugs were discovered in the memory stage. On fixing the bugs we were able to match Instruction count and Cycle count given by the report, however fact12 instruction count does not match.

Application	Instruction Count	Cycle Count
noio	2081	2112
file	95215	95282
hello	95705	95760
class	97177	97232
sort	104924	104987
fact12	110820	110923
matrix	137286	137401
hanoi	201667	201842
ical	216479	216534
fib18	305749	305804