ECSE 425: Computer Organization and Architecture

PD6 Final Report

Zhong Guan (260758587), Kristin Lee (260509976), Runze Zhang (260760723), and Yining Zhou (260760795), *Group 6, McGill University*

***Abstract*—This report presents a potential implementation of a pipelined processor. The processor was optimized to include an instruction and data cache. The processor was implemented using VHDL and simulation was completed using ModelSim and EDA Playground. The report contains information regarding the design process, performance evaluation, and results obtained.**

***Index Terms*—Cache, MIPS, processor.**

I. Introduction

THIS project required the group to implement a pipelined processor. The processor is a standard, five-stage, 32-bit pipelined MIPS processor with a forwarding unit and a hazard detection unit. The optimization is based on the cache implementation, which includes an instruction cache and a data cache.

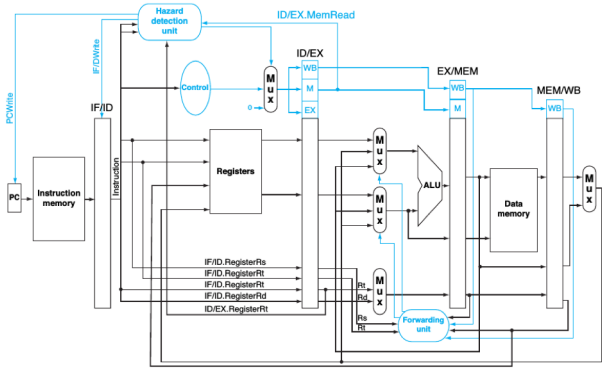

Fig. 1. Pipelined processor, with a forwarding unit and a hazard detection unit.

II. Design Approach

For the instruction cache, only the read case needs to be considered. We separate working flow into three stages: idle and compute stage, memory access stage, and write-back and memory read stage. In the first stage, if a hit exists, the instruction cache directly outputs the result. Otherwise, it will determine whether it needs to conduct write-back, based on the dirty bit. The second stage is more like a control stage, in which several intermediate signals are set and reset. The third stage is for interacting with memory, including write-back (if dirty) and reading from memory. Both require four times to complete, that is, one word at a time. After write-back is conducted, it will go back to the first stage and follow the hit pattern.

For the data cache, both read and write cases must be considered. However, the only change is in the first stage. If a hit occurs, data will be directly written into the block. The other steps are the same as the instruction cache.

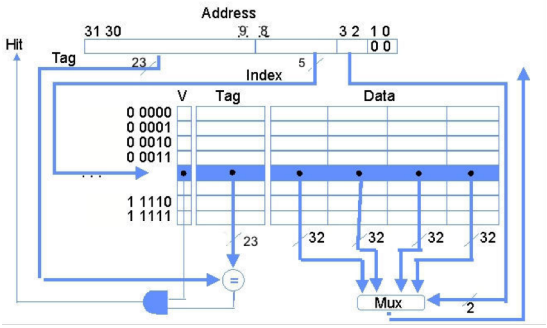

Fig. 2. Direct-mapped cache, 4 word blocks.

A. Testing Procedure

Several MIPS programs were used to test the performance of the processor.

Factorial program: it is used to calculate the factorial of an integer. The result is stored into register 2. The program contains basic ALU execution: add and multiply and uses branch instruction to conduct looping.

Fibonacci program: it is used to generate Fibonacci series. The generated numbers are first stored in register 2, and then to adjacent locations in memory. The program generates in total 32 numbers. Store word instructions show the advantage of cache for interacting with memory.

Bitwise Program: it contains no loop and a few branch commands. Therefore, caching hardly have any effect on the program performance.

GCD Program: it finds the greatest common divisor of 2 number using the Euclid's Algorithm.

Addition Program: it calculates the sum of the first n integers.

Array Store Program: it stores an array into memory

B. Evaluation Procedure

The run time of the programs were compared for each processor (with and without the cache).

III. Details

A. Component Descriptions

The pipeline divides a typical instruction into five stages,

Instruction Fetch (IF), *Instruction Decode* (ID), *Execution* (EX), *Memory* (MEM), and *Write Back* (WB).

The instruction fetch stage fetches the instruction from memory, according to the current value of the program counter register, and sends it to the decode stage.

The instruction decode stage receives the instruction from the instruction fetch unit and sends the appropriate signals to the other units in the pipeline. It will control the ALU to perform the appropriate operation. It reads values from the register file and sends them to the input of the ALU, if needed. It also handles the signals for branching and forwarding. If there is a branch, a signal will be sent to the IF stage to change the PC value. A signal for the forwarding unit keeps track of the destination register of the previous instruction. Thus, the forwarding unit can decide whether to operate on the current register value or the ALU output. If forwarding needs to be conducted, another signal informs the forwarding unit whether the ALU should receive the result from the last instruction or from the memory stage.

The execution stage receives signals from the decode stage and the forwarding unit. It is a simple unit to operate on the passed operands based on the opcode.

The memory stage acts on all data memory accesses. It receives a signal from the decoder and determines whether the current instruction needs to access memory.

The write back stage receives signals from the EX stage to determine whether it needs to stall for memory or it can take the value from the EX stage directly and perform write back. The destination register is sent from the decoder to it.

Between these stages, pipeline registers are inserted to ensure that there is no conflicting data due to multiple instructions being executed simultaneously.

The forwarding unit is a hardware solution to deal with data hazards, which is used to properly pass values early from the pipeline registers to the input of the ALU rather than waiting for WB to write to the register file. When an instruction depends on a non-memory instruction prior to it, the forwarding unit will feedback the output of the EX stage back to it. When an instruction depends on a memory instruction, the forwarding unit will pass the output of the memory stage to the EX stage.

The hazard detection unit is used in the ID stage to insert a stall between load and its use.

B. Optimization

In the deliverable 4, we developed two memories: instruction memory and data memory, and put them into two stages: IF and MEM. It was also assumed that it has no delay. In this project, memory is implemented as a unified one, for both code and data, and is isolated from stages as a separate entity. The memory delay is set as 10 cycles.

Two types of caches are included in this project. An **instruction cache** is to speed up executable instruction fetch. A **data cache** is to speed up data fetch and store.

The caches implemented in our project have the following characteristics and parameters:

- Write-back policy
- Direct-mapped
- 32-bit words
- 128-bit blocks (4 words per block) word addressable
- 4096 bits of data storage (32 blocks)
- 32-bit addresses
- data tags flags (valid bit, dirty bit)

Memory cache is the fastest type of memory after the CPU registers, therefore, the cache design and caching strategies directly impact processor performance.

The write method used is write-back policy. Since the cache storage is large (4KB), there are not many write-backs and the method is clearly better than the write-through method.

The structure used is direct-mapped. When the data required is in the cache (hit), the efficiency of direct-mapped is fully exploited. This is because it needs less time to search for the data in the cache compared to higher associativity.

IV. Testing and Performance Evaluation

In this section, we will discuss how we tested our system and evaluated its performance after improving it through instruction and data caching. We will first describe how we tested our system from a functional perspective. After that, we will give a thorough description of our performance evaluation.

A. Functional Testing

We tested our pipeline processor starts from the very first stage IF after implementation. After the IF stage is validated, we then add the second stage ID and then add each stage one by one. Every time we only add one unit of pipeline to easily verify which stage has occurred problems so that we can specifically focus on one stage to perform debugging instead of working on the whole pipeline.

Once we added all stages together, we performed integration testing of the whole pipeline using the test programs. Several instructions were fed into the CPU and the outputs of registers 0 through 31 were monitored. The instructions that were tested included register based and immediate adds, subtracts (both signed and unsigned), reading and writing memory, and a loop that would force the CPU to jump back to the start of instruction memory and execute those same instructions again. We output the contents of memory inside the MEM stage and output the register file inside the ID stage instead of doing them in the test bench. We can easily make sure the contents of both registers and memory content are correct by checking out these files.

B. Cache Performance Testing

First, we need to verify the execution times of both instruction cache and data cache with the same memory delay when a miss happened. We will discuss the simulation result of our pipeline processor based on the Fibonacci series program and verify how the instruction and cache work in our processor.

Since the size of block for both instruction cache and data cache is 4 words and the size of each memory address is one word, once a cache miss happened, cache would read four times data from memory to fill in its block after 40 clock cycles memory delay. The corresponding simulation waveforms are shown below.

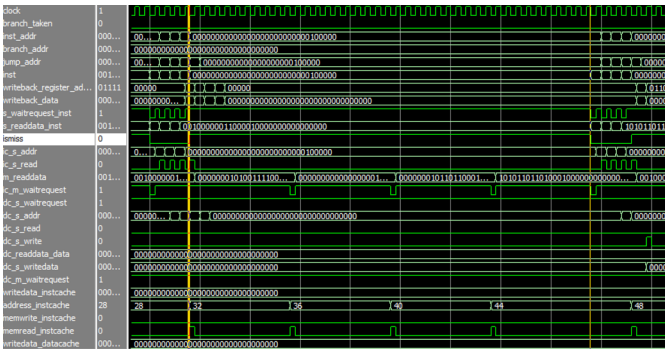

Fig. 3. Simulation Waveform.

In Figure 3, the period between the two yellow lines represents the time of instruction cache read miss and it takes 40 clock cycles memory delay to read a block from memory into cache. Once the block is filled into cache, cache hit happened four times afterwards and each time takes one clock cycle, which conforms with our previous discussion. Hence, an instruction miss would totally cost 44 clock cycles.

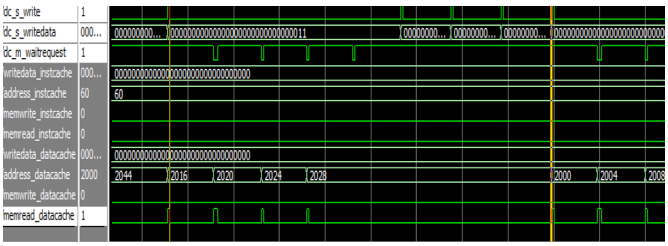

Fig. 4. Simulation Waveform.

In Figure 4, the period between the two yellow lines represents the execution time of store instruction. Data cache write miss happened at the first yellow line and then data cache read a block from memory which takes 40 clock cycles. After block is filled into cache, write hit happened four times which also takes 40 clock cycles totally. Hence, a store instruction would take 80 clock cycles with a write miss happen.

Next, the instruction and data caches were tested by running the pipeline on 4 different programs. However, due to the limited presentation space, we only present analysis result on 3. For each program, we measured the runtime in clock cycles without cache and with cache separately. Hence, we have performed 8 experiments in total. We will look at the runtime of each program and the performance of processor with and without cache.

```
    # Example: 6! = 1 * 2 * 3 * 4 * 5 * 6
    # initializing the beginning of Data
    addi $11, $0, 2000
     # n = 6
    addi $2, $0, 6
    addi $3, $0, 1
    addi $4, $0, 1

loop:   beq $2, $4, DONE
    addi $4, $4, 1
    mult $3, $4
    mflo $3
    j loop
    # store result
DONE: sw $3, 0($11)
```

Fig. 5. MIPS factorial program.

1) Factorial of an integer

This program generates the factorial of a positive integer. It stores the result first into Reg[2], and then into memory. The details of the program are shown in Figure 5.

Runtime with cache: 205 CC
Runtime without cache: 392 CC

2) Fibonacci

This program generates the Fibonacci series and stores the generated numbers first into Reg[2], and then into memory, starting from 2000. The details of the program are shown in Figure 6, which is used to generate 8 numbers of the Fibonacci series.

```
    addi $10,  $0, 8    # number of generating Fibonacci-numbers
    addi $1,   $0, 1    # initializing Fib(-1) = 0
    addi $2,   $0, 1    # initializing Fib(0) = 1
    addi $11,  $0, 2000     # initializing the beginning of Data
    Section address in memory
    addi $15,  $0, 4    # word size in byte

loop:   addi $3, $2, 0      # temp = Fib(n-1)
    add  $2, $2, $1     # Fib(n)=Fib(n-1)+Fib(n-2)
    addi $1, $3, 0      # Fib(n-2)=temp=Fib(n-1)
    mult $10, $15       # $lo=4*$10, for word alignment
    mflo $12        # assume small numbers
    add  $13, $11, $12  # Make data pointer [2000+($10)*4]
    sw   $2, 0($13) # Mem[$10+2000] <-- Fib(n)
    addi $10, $10, -1   # loop index
    bne  $10, $0, loop

EoP:    beq  $11, $11, EoP  #end of program (infinite loop)
```

Fig. 6. MIPS Fibonacci program.

Runtime with cache: 370 CC
Runtime without cache: 934 CC

3) GCD of 2 numbers

This program is used to find the greatest common divisor of 2 numbers using Euclid's Algorithm. The details of the program are shown in Figure 7, which finds the GCD of 25 and 35.

```
    addi $13, $0, 2000
    addi $10, $0, 35
    addi $11, $0, 25
gcd:    beq $10, $11, DONE
    slt $2, $11, $10
    bne $2, $0, loop1
    sub $11, $11, $10
    j gcd
loop1:  sub $10, $10, $11
    j gcd
DONE:   sw $10, 0($13)
```

Fig. 7. MIPS GCD program.

Runtime with cache: 201 CC
Runtime without cache: 343 CC

4) Array Store

This program is used to store an array into memory. The values being stored are integer from 1 to 20. The details of the program are shown in Figure 8.

Runtime with cache: 385 CC
Runtime without cache: 985 CC

```
    addi $10, $0, 2000
    add $1, $10, $0
    addi $2, $0, 1
    addi $3, $0, 2020
    addi $11, $0, 2021

loop1:  sw $2, 0($1)
    addi $1, $1, 1
    addi $2, $2, 1
    bne $1, $3, loop1

    addi $1, $10, 0

loop2:  lw $5, 0($1)
    addi $1, $1, 1
    bne $1, $3, loop2

    sw $5, 0($11)
```

Fig. 8. MIPS Array Store program.

5) Sum from 1 to n

This program calculates the sum of the first n integers. The test was carried out with n = 8. The details of the program are shown in Figure 9.

```
    addi $11, $0, 2000  # initializing the beginning of Data
    addi $1, $0, 8    # n = 8
    addi $2, $0, 0      # init i = 1
    addi $3, $0, 1      # init j = 1

loop:   beq $1, $3, DONE
        add $2, $2, $3
        addi $3, $3, 1
        bne $11, $0, loop

DONE:   sw $2, 0($11) # store result
EoP:    beq  $11, $11, EoP  #end of program (infinite loop)
```

Fig. 9. MIPS Addition program.

Runtime with cache: 211 CC
Runtime without cache: 442 CC

6) Bitwise Program

This program is used to compare the differences between bitwise and logical operators. It contains no loops and a few branch instructions. We choose to perform the bitwise operators in this program, which would only run a few instructions and end.

Runtime with cache: 408 CC
Runtime without cache: 253 CC

Unsurprisingly, the runtime without cache is smaller than the runtime with cache.

Since there are quite few instructions in this program and even without loop, it is possible for the processor without cache run faster than the processor with cache. It is because there would be 40 cycles memory delay if a miss happened in cache but only 10 cycles memory delay for a processor without cache, if the instructions is so few that cache can only read a few blocks from memory, the processor without cache would be possibly faster.

V. Conclusion

This project helped us to understand how a pipelined processor works and how it can be optimized by implementing separate caches for instructions and data. Both instruction and data caches can save CPU time by reducing time delay when accessing memory with a great number of instructions specifically with several loop or branch instructions. However, in certain cases such as the program contains only few instructions with single loop or even no loop, the cache may decrease the performance of the pipeline processor.

This was a difficult project, mainly because we need to combine the pipelined processor structure developed in deliverable 4 and cache. Separate memories needed to be merged into a unified one. Multiple revisions of the design were required when wiring the components together.

Other future improvements that could be implemented on the processor are better branch prediction and register renaming to eliminate name dependencies.