ECSE 425: Pipelined Processor

Ahmad El Nakib
260610647

Mohamed El Sabagh
260603261

Omar El Lakany
260603639

I. Introduction

The purpose of this project was to create a pipelined processor using VHDL. The processor implemented a basic 5-stage pipeline consisting of the following stages: IF (Instruction Fetch), ID (Instruction Decode), EX (Execute), MEM (Memory Access) and WB (Write Back). Once the processor was functional, it was optimized by integrating both an instruction and data cache. The performance of the optimized processor was then compared to that of the un-optimized one by running both on benchmark programs. Throughout this paper, the design methodology, optimization and results will be discussed.

II. Program Design

Before discussing the implementation and functionality of the various pipeline stages, we shall discuss some implemented components that aided in creating a safe, efficient pipeline. These components were namely the pipeline registers and the forwarding unit, these components heavily relied on each other. A pipeline register was added for every phase, the pipeline register would then store the output value of that stage, allowing it to be accessible by a future instruction rather than have the instruction wait until the write back phase. A forwarding unit was implemented to deal with potential data hazards. The method in which the forwarding unit functioned was as follows: If a later instruction had a dependency upon an earlier instruction, the forwarding unit would pass the value from the prior instruction (by retrieving the value from a pipeline register) and passing it to the later instruction, given that the value was available in the register, and the difference between producer and consumer was enough. If not, stalls would have to be inserted until the difference was enough and forwarding could then occur (such as when a LW was followed by an ADD that required the value from LW).

A- Pipeline Design

Now that the components that were implemented in the pipeline have been explained, descriptions of how each of the 5 basic pipeline stages operated are as follows:

IF:
Instruction fetch, the goal of this stage was to simply fetch the instruction from memory based on the current value of the program counter. Once this instruction was fetched, it was sent to the instruction decode stage.

ID:
Instruction decode, in this stage, the instruction that was fetched in the IF stage was passed in. Based on the instruction that was received, various signals would be sent. If the instruction was a branch for example, a signal was sent so that the program counter was updated and would point to the instruction that we want to branch to. This new instruction would then be fetched and passed to the instruction decode stage where once again, signals based on which instruction would be sent. If the instruction was an ALU instruction (such as ADD), signals would be sent so that values from the registers were fetched and passed into the ALU to be operated on in the execution stage.

Signals were also sent from this stage to the forwarding unit in order to prevent data hazards. One of the signals sent to the forwarding unit would keep track of previous instructions' destination registers. This allowed the forwarding unit to detect any dependencies. If there were dependencies, the forwarding unit could determine whether the value required by the later instruction would have been written to the register by the time it is needed or not. If not, forwarding would need to be occur, and another signal would be sent, determining whether the value would be retrieved from the EX or MEM stage of the previous instruction. This was made possible by the pipeline registers mentioned before.

EX: Execution stage, this stage would receive signals from the ID stage and the forwarding unit. These would define the register values that would be used in the operation and which operation it was. The operation was determined by the opcode that was passed in. The ALU would then perform the appropriate operation on these values.
MEM: The memory stage was for any instruction that required access to data memory. Based on the signal from the decoder, it would determine whether a memory access was needed or not, for example, if the instruction was LW.

WB: The write back stage would receive signals from the EX stage, which determine whether stalls were needed for memory or if the values from the EX stage could be taken and written to memory instantly. The destination register input would come from the decoder

B- Optimization

Throughout this section, we shall discuss the optimization that was implemented, namely including a cache, and why we chose the specifications we did. So far, throughout our implementation, we had a unified memory that contained both data and instructions. In order to optimize this processor, two caches, an instruction and data cache, were included. The purpose of the instruction cache was to speed up the IF phase while the purpose of the data cache was to speed up the fetching and storing of data. The caches that were implemented had the following characteristics:

- Direct mapped
- 4096 bits for storage
- 32-bit address
- 128-bit blocks
- 4 words/block
- 32-bit words
- Valid and dirty bits

- Write back

Cache design and caching strategies are crucial aspects of processor performance. Caches are the second fastest type of memory (after registers), so strategical and intelligent implementation could reduce execution times significantly. For our block placement strategy, we used direct-mapped as opposed to fully associative or set associative. The reason for this is that direct-mapped is faster and more simple, even though there is a higher miss rate. Since we are using a large cache, implementing block placement with higher associativity would result in more time wasted searching through the cache. When using direct-mapped, we search in one block, if we hit, we have exploited the full efficiency of direct-mapped and retrieve the block instantly, there is no time wasted searching. Currently, almost no caches use write through as they require too many accesses to memory and the goal is to minimize execution time. It is for that reason that we implemented write back for our caches. When a block needs to be replaced, we check the dirty bit, if it is set, we write it to the memory then replace it. If the block is clean, we simply replace it. Since the cache was quite large, the chances of replacing a modified block were not high and therefore not many writes would be needed, decreasing execution time. The main downside to using write back however is that the memory will not have the most up-to-date values as it would with write through, however, we found this sacrifice worth it as the execution time would be much less when using write back.

As for how the caches themselves operated, the only consideration for the instruction cache was the read case, as opposed to the data cache where both read and write scenarios had to be taken into account (seeing as we never write instructions). Work flow for these caches was separated into 3 cycles: Idle and compute, memory access, and write-back/memory read. In the idle and compute stage, based on whether it was a hit or not, the instruction cache would either return the requested instruction while the data cache returned the requested block, if it was a miss however, the caches would have to determine whether a write back would be needed on the basis of whether a dirty bit existed or not. The memory stage simply set and reset signals that determined whether memory access was required or not, this was determined by whether it was a hit or miss. The memory and write back stage was for interacting with memory, which would be needed if there was a miss (as the requested instruction or block was not in the cache). This stage included the write back procedure for blocks with dirty bits (As mentioned before) and reading from memory when a miss occurred, replacing the missing block. Once the write back stage was complete, the cache would return to the first stage and repeat the process, with the previously missing block in the cache this time. The rest of the process would continue as if it was a hit.

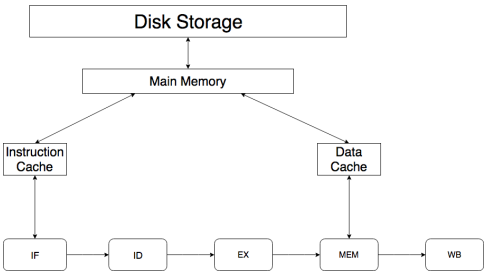

Figure 1: Diagram Showing Optimization

Please note that the above diagram is not the entire processor, it is simply to show how the caches were added.

C- Testing Design

Several programs were used to test the performance of the processor. These programs were run with both the un-optimized and optimized processor. The results were then compared to see to what extent the optimization had improved performance. The tests that were run were the following:

1) Factorial program: Calculate the factorial of an integer and store the result into a register. This program contains basic ALU operations, add and multiply.
2) Fibonacci program: This program is used to generate the Fibonacci sequence. The result is stored in a register then stored in another register. The result is 32 numbers, all stored in adjacent registers (5^{th} value in register adjacent to register containing 4^{th} value).
3) Addition program: Sum up the last n integers
4) Square root: Find the square root of a perfect square
5) GCD: Find the greatest common divider of 2 numbers
6) Primes: Find all prime numbers until n

The un-optimized and optimized processor were evaluated by comparing the total amount of clock cycles that were required for the processor to run the program. It is worth mentioning that the un-optimized processor was not fully functional and therefore failed some tests.

III. Testing

In this section, we shall discuss how we tested and evaluated the system both before and after adding the caches to optimize it. The following sections, system testing and performance evaluation, discuss how we tested the system itself and how we evaluated its performance

A- Functional Testing

In order to make sure that our system was functional, we implemented both unit and integration testing while developing the code. The testing was conducted as follows: After developing the code for a stage such as IF, testing on that stage was conducted. Once this stage had no errors, the second stage, ID was developed. ID would then be tested individually. Once it was functioning perfectly, it would be

tested alongside IF, this was the integration testing component. The same procedure would occur for the following stages, EX would be tested separately then tested alongside ID and IF. This resumed until we tested the 5 stages together. This method allowed us to easily define where issues were arising and made debugging significantly easier and more time efficient.

IV. Results

Several programs were run on both the optimized and un-optimized processor. The tests were as follows:

A- Fibonacci Sequence

This is the Fibonacci program that was provided in P4. It stores the generated numbers into register [2] ($2), and then into memory starting at address 2000. The details of the program are provided.

```
#This program generates Fibonacci series.

    addi $10,  $0, 12        # 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)
```

Figure 2: MIPS Fibonacci Program

B- Factorial

This is the factorial program that was provided in P4. The number is stored in register [2] ($2), we then loop, incrementing register [4] each time and multiplying it by the previous result. This result is stored in register [3]. We stop when we register [4] s equal to the number we put in. (Although that multiplication occurs in the loop before)

```
#Example: 6! = 6*5*4*3*2*1
#Initialize the beginning of data

addi $11, $0, 2000
addi $2, $0, 6      #n=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

DONE: sw $3, 0($11) #Store the result
```

Figure 3: MIPS Factorial Program

C- GCD

The purpose of this program was to find the greatest common divider between 2 numbers. The numbers are set in the third and fourth line, in register [4] and register [5]

```
addi $11, $0, 2000
addi $29, $0, 32764

test:   addi $4, $0, 100
        addi $5, $0, 60
        sw   $4, 0($11)
        sw   $5, 4($11)
        addi $4, $0, 99
        addi $5, $0, 151
        lw   $4, 0($11)
        lw   $5, 4($11)

        jal gcd
        sw   $2, 8($11)

gcd:    slt $8, $5, $4
        bne $8, $0, swap

        addi $29, $29, -12
        sw $5, 8($29)
        sw $4, 4($29)
        sw $31, 0($29)
        or $2, $0, $5
        beq $4, $0, gcd_rt
        or $8, $0, $5
        or $5, $0, $4
        div $8, $4
        mfhi $4
        jal gcd
        lw $5, 8($29)
        lw $4, 4($29)
        j gcd_rt

swap:   or $8, $0, $4
        or $4, $0, $5
        or $5, $0, $8
        j gcd

gcd_rt: lw $31, 0($29)
        addi $29, $29, 12
        jr $31

EoP:    beq  $11, $11, EoP
```

Figure 4: MIPS GCD Program

D- Primes until N

This program generates a certain number of prime numbers. It starts off with 2 (as 2 is an exception), then checks if odd numbers are divisible by any prime already found, if not, then the number is a prime number.

```
        addi $16, $0, 2000
        addi $23, $16, 0
        addi $17, $0, 16
        addi $18, $0, 0

        addi $19, $0, 2
        sw   $19, 0($23)
        addi $18, $18, 1
        addi $23, $23, 4

        addi $19, $0, 3

L0:     beq  $17, $18, EoP
        addi $4, $19, 0
        addi $5, $16, 0
        addi $6, $23, 0
        jal  prime
        beq  $2, $0, L1
        sw   $19, 0($23)
        addi $18, $18, 1
        addi $23, $23, 4
L1:     addi $19, $19, 2
        j    L0


prime:  addi $2, $0, 1
L2:     lw   $8, 0($5)
        addi $5, $5, 4
        div  $4, $8
        mfhi $8
        beq  $8, $0, L3
        beq  $5, $6, return
        j    L2
L3:     addi $2, $0, 0
return: jr   $31

EoP:    beq  $0, $0, EoP
```

Figure 5: MIPS Prime number program

E- Square Root

This program calculates the square root of a perfect square. In the following we have input 36, we keep incrementing (starting from 0) and checking if the increment^2 is less than or equal to 36. If so, we increment again. If it is equal however (will never be greater than), we have found the square root of this perfect square. In this case, we will break when we reach 6, and store that in memory.

```
addi $21, $0, 2000
addi $6, $0, 1

main:    addi $2, $0, 0
         addi $1, $0, 36

compare: mult $2, $2
         mflo $3
         slt $5, $3, $1
         sub $4, $1, $3
         beq $4, $0, loop
         beq $5, $6, loop
         j return

loop:    addi $2, $2, 1
         j compare

return:  addi $7, $2, -1
         sw   $7, 0($21)

EoP:     beq $2, $2, EoP
```

Figure 6: MIPS Square Root

F- Sum

This program calculates the sum of integers 1 to N. As can be seen in the below assembly code, we generated the sum from 1 to n=15.

```
addi $11, $0, 2000
addi $1, $0, 15
addi $2, $0, 1
addi $3, $0, 0

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

DONE:   sw $3, 0($11)
EoP:    bew $11, $11, EoP
```

Figure 7: MIPS Sum Program

G- Final Results

	Un-optimized Processor	Optimized Processor	Performance Improvement
Fibonacci	873 CC	311 CC	65%
Factorial	488 CC	232 CC	52%
GCD	313 CC	189 CC	40%
Primes	793 CC	464 CC	42%
Square Root	532 CC	268 CC	50%
Sum	469 CC	208 CC	66%

Figure 8: Results Table

As can be seen from the above table, including a cache significantly improves performance. The improvement depends on the program being run. The only scenario in which a processor without a cache would perform better than one with a cache would be one in which there is a high miss rate.

V. Conclusion

Throughout this project, we gained a deeper understanding on how pipelined processors function, gaining in depth knowledge of not only the functionality, but also all the various issues and hazards that must be dealt with as well as the various solutions to the hazards. In addition to learning how a pipelined processor operates, we were also able to see what effect adding a cache has on the operation of the processor. By adding the cache, CPU times decreased significantly. However, while conducting tests, we did notice that there may be some downsides to using a cache. If there is a cache miss, the memory must be accessed and blocks replaced. So in addition to searching through the cache, there is also the time of accessing memory, replacing the block, then accessing the cache once more. All these extra times that occur as the result of a miss could lead to worse CPU times, especially if the miss rate is high. In conclusion, caches are effective when there is a large amount of instructions that require memory access such as load and store, given that the miss rate is not too high.

Many challenges were faced in this project. In part 4, implementing the various stages and connecting them was difficult, a lot of time was spent on integrating the stages together. Even when the whole pipeline was connected, it was still not completely functional. The program could run the Fibonacci program but not the factorial. Furthermore, when optimizing, new issues arose when integrating the cache into the prior model. The partially functioning part 4 had to be fixed and then the caches had to be integrated. New signals and many revisions were required before it was functional. Future improvements could be implementing branch prediction (2-bit branch predictors or correlated branches), static or dynamic scheduling or even arithmetic operations that operate on multiple words in parallel (SIMD).