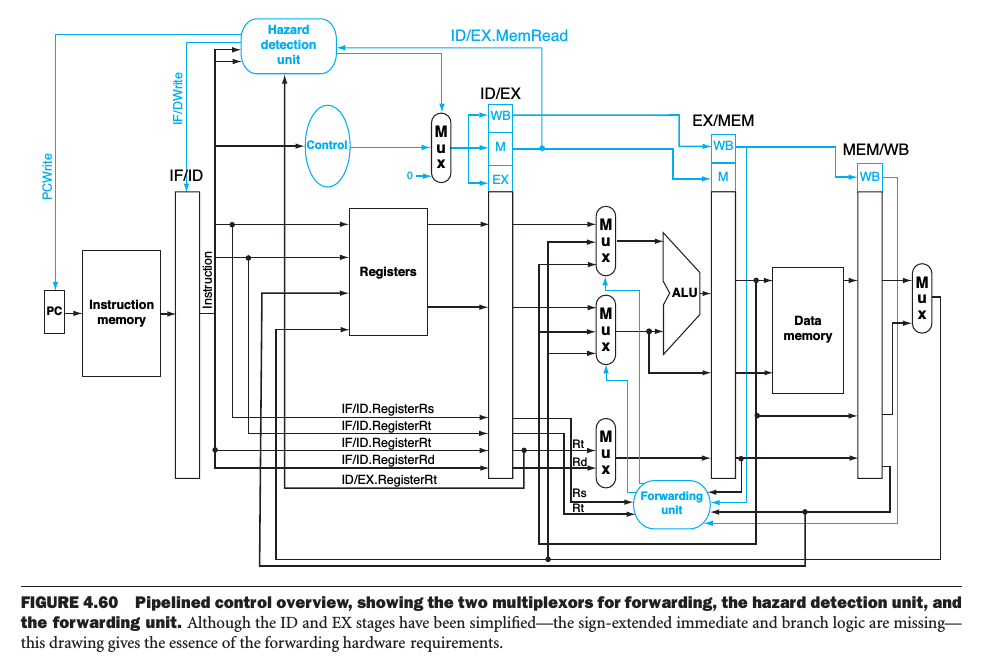
1. An overview of your processor optimization, including figures (e.g., block diagram, cache or branch predictor structure).
2. A description of your design approach, including testing and evaluation procedure.
3. Detail any optimizations you implemented, evaluation methodology, and results.
4. Conclusions and, if appropriate, post mortem detailing things you would do differently given the opportunity.

the approach taken to evaluate the optimization

1. Introduce the design problem.
2. Introduce your solution, in general terms, with a block diagram (if applicable).
3. Describe in some detail the different components of your system, including why you implemented particular elements.
4. Describe how you chose particular parameters and how you evaluated the system.
5. Describe your results and explain the behavior you observe.
6. Make recommendations based on your experience; would you do anything different if you were able to do the project over?

*Overview*

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

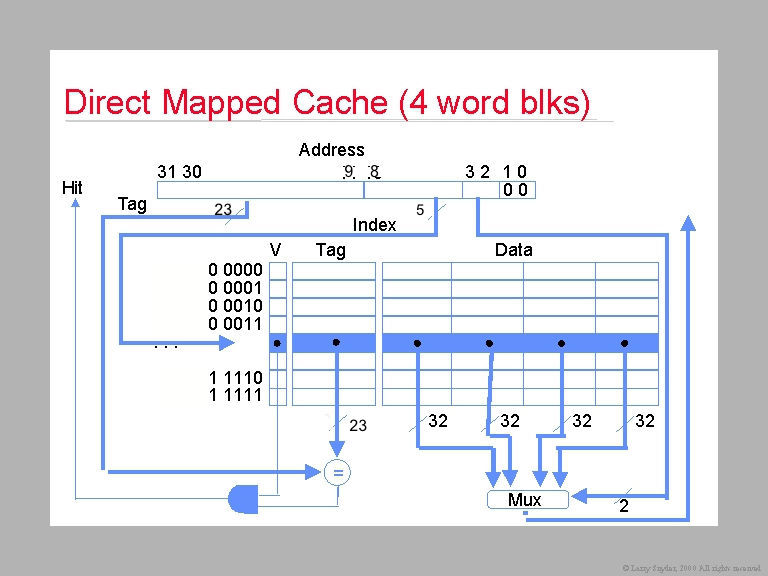


*Description*

*design approach*

For instruction cache, there is only read case. 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 hit exists, instruction cache directly outputs result. Otherwise, it will determine whether needs to conduct write-back based on dirty bit. The second stage is more like a control stage, in which several intermediate signals are set and reset. The third stage is actually for interacting with memory, including write-back (if dirty) and reading from memory. Both requires 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 data cache, there are both read and write cases. However, the only change is in the first stage. If hit happens, data will be directly written into the block. Other steps are the same to instruction cache.



*testing procedure*

We applied several MIPS program to test the performance of 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 command. 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

*evaluation procedure*

compare the clock cycle consumed to run program

justify the accuracy of final result

*Details*

*components description*

The pipeline basically divides a typical instruction into five stages, IF (*Instruction Fetch*), ID (*Instruction Decode*), EX (*Execution*), MEM(*Memory Access*), and WB (*Write Back*).

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

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

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

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

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 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.

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

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

*optimization*

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

In the present work there are two types of caches included. **Instruction cache** is to speed up executable instruction fetch. **Data cache** is to speed up data fetch and store.

The caches implemented in our project has 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 cache(hit), the efficiency of direct-mapped is fully exploited. Because it needs less time to search for data in cache compared to higher associativity.

*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.

1. 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. A number of 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.

1. Cache Performance Testing

First, we need to verify the execution time 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 according simulation waveform is shown below.

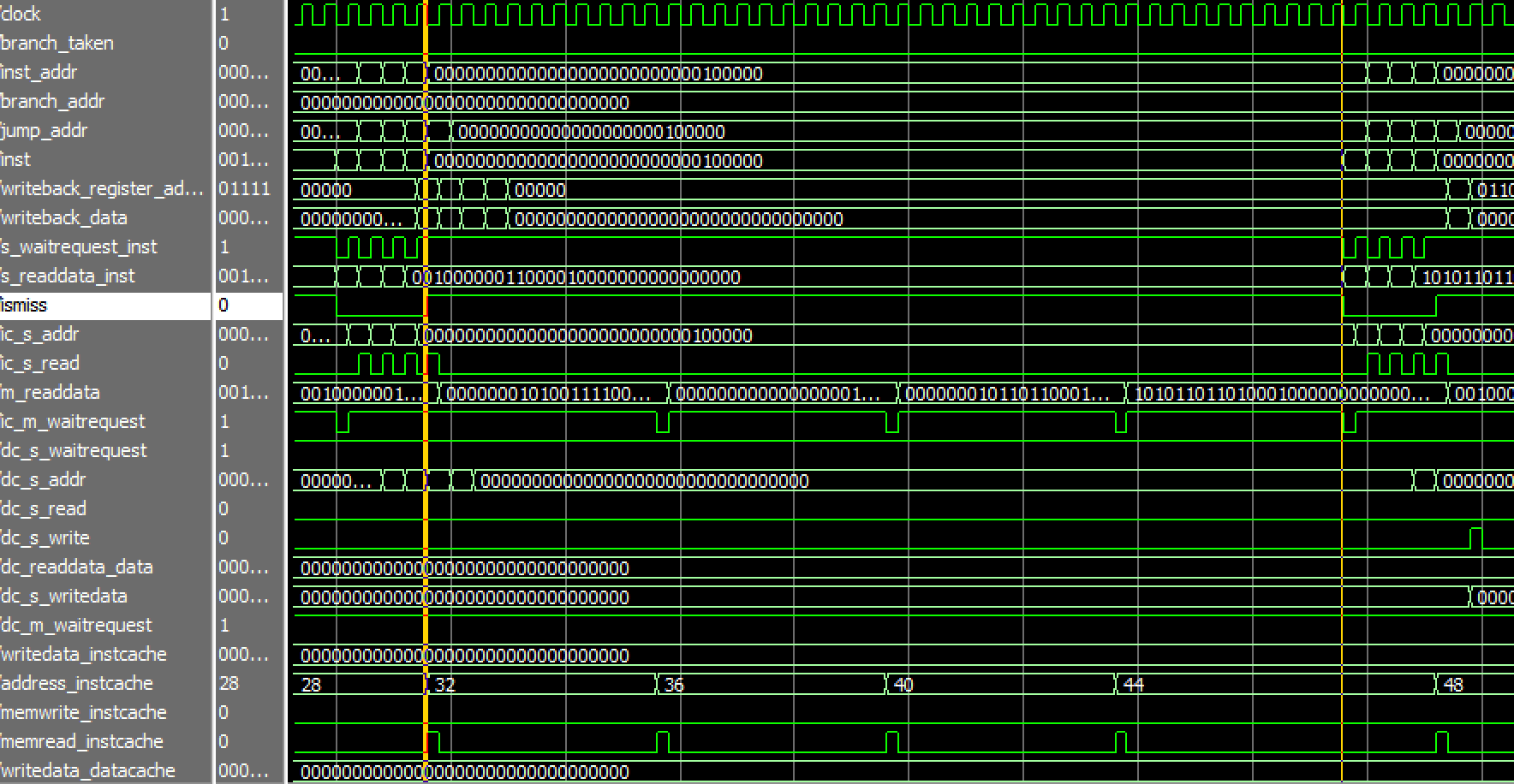


Figure 1. Simulation Waveform

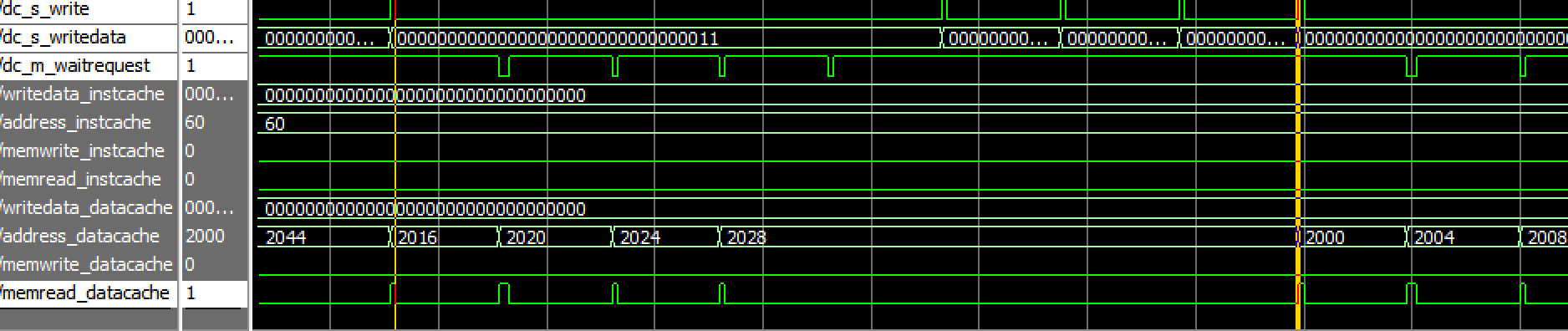


Figure 2. Simulation Waveform

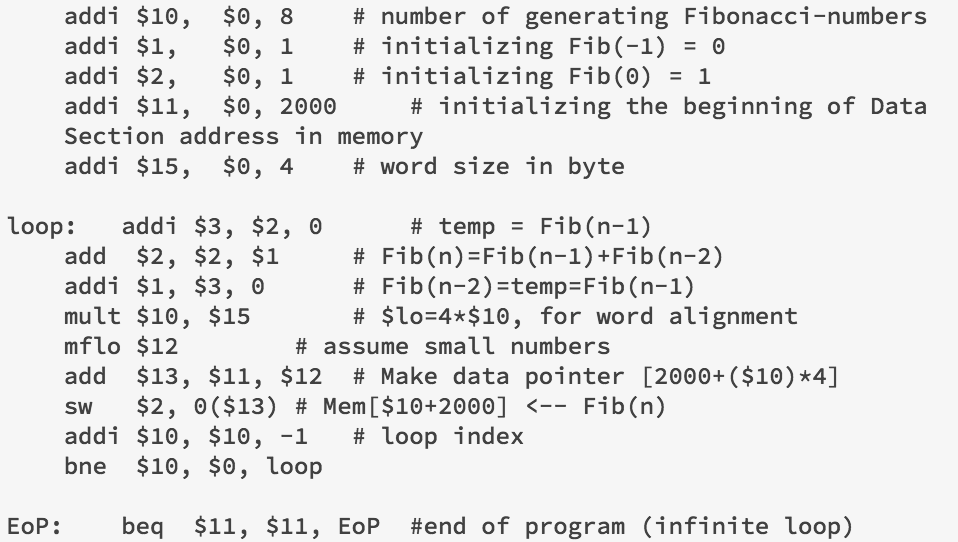
For the first figure, the period between the two yellow lines represents the time of instruction cache read miss and it takes 40 clock cycles memory delay to read a block from memory into cache. once the block is filled into cache, cache hit happened four times afterwards and each time takes one clock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

For the second figure, 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, we test the instruction and data cache by running our pipeline on 6 different programs. For each program, we measured the runtime in clock cycles without cache and with cache separately. Hence, we have performed 12 experiments in total. We will look at the runtime of each program and the performance of processor with and without cache.

1. Fibonacci Series

This program generates Fibonacci series and stores the generated Fibonacci numbers first into Reg [2], and then into memory starts from 2000. The detail of program is shown below, which is used to generate 8 of Fibonacci series.

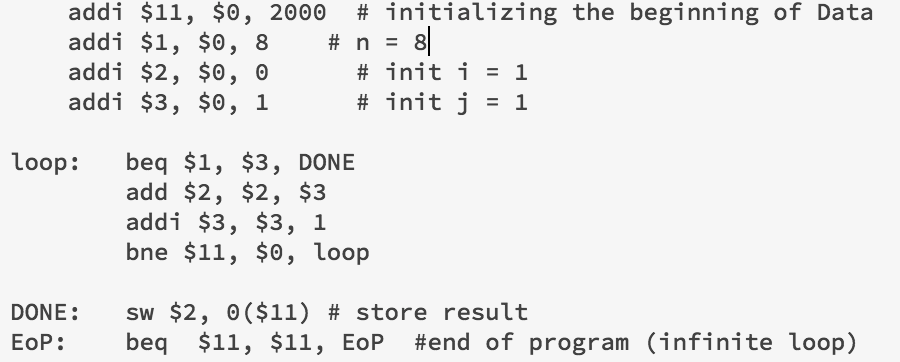


**The runtime with cache: 370 Clock cycles.**

**The runtime without cache: 934 Clock Cycles.**

1. Sum from 1 to n:

This program calculates the sum of the first n integers. The test was carried out with n = 8. The detail of program is shown below.

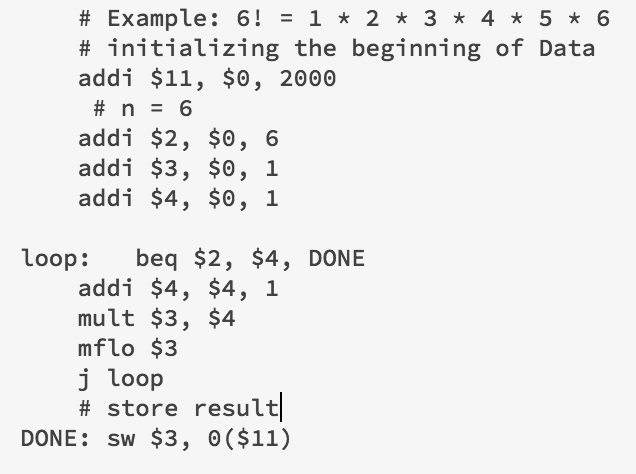


**The runtime with cache: 211 Clock Cycles.**

**The runtime without cache: 442 Clock Cycles.**

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 detail of program is shown below.



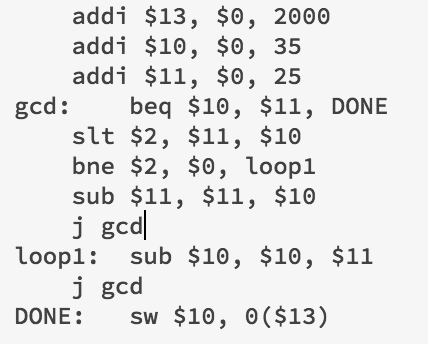
**The runtime with cache: 205 Clock Cycles**

**The runtime without cache: 392 Clock Cycles.**

This program contains only one store instruction and a few branch commands among the small number of instructions. Consequently, both instruction and data caching does not have any effects on the program performance. This is reflected as almost the same runtime between the processor with and without cache.

1. GCD of 2 numbers:

This program is used to finds the greatest common divisor of 2 number using the Euclid’s Algorithm. The detail of this program is shown below, which finds the GCD of 35 and 25.

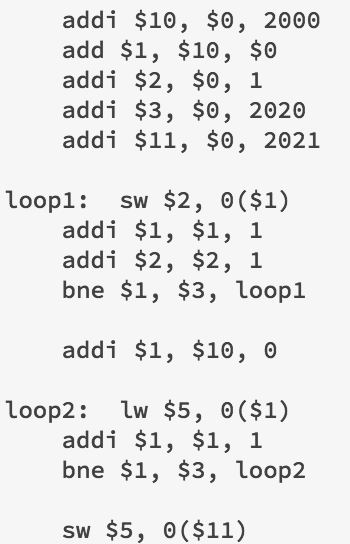


**The runtime with cache: 201 Clock Cycles**

**The runtime without cache: 343 Clock Cycles**

1. Array store program:

This program is used to stores an array into memory. The values being stored are 1~20.



The runtime with cache: 385 Clock Cycles

The runtime without cache: 985 Clock Cycles

1. Bitwise Program

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

**The runtime with cache: 408 Clock Cycles**

**The runtime without cache: 253 Clock Cycles**

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.

*Conclusion*

This project helps us to understand how a pipeline processor works and how it can be optimized by implementing separate caches for instruction and data. Both instruction and data caches can save CPU time by reducing time delay when accessing memory with a great number of instructions specially with a number of 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.

The was a difficult project, mostly because we need to combine developed pipelined processor structure in project 4 and cache together. We need to merge separate memory into a unified one.

And we had to revise our design multiple times when wired the components together.

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