Luke Martin
0926966

Current State of the mMIPS Processor:

A mMIPS processor is a simplified version of a typical MIPS processor. It functions in the same as a typical MIPS processor except has reduced amount number of available instructions. Just like any other MIPs processor, this implementation has five stages namely; Instruction Fetch (IF), Instruction decode and register read (ID), Execution (EX), Memory Access (MEM), and Write Back (WB). In the first stage, the instruction is fetch from memory, based on the current PC counter. This is then stored into registers which store the information between cycles. During the instruction decode, the fetched instruction is interpreted and the corresponding registers are accessed again storing the information that may be used by future stages in registers. The execution stage is simply where the processors performs the computation for example this may be calculating the next address, such as the case during a Jump instruction. The stage features an arithmetic logic unit or ALU, which performs these computations. In memory access, memory loads or stores will be performed if necessary. Write back is where the result is written to the correct register file. Other important component features to point out are the control, hazard and hazard control, and branch control. Each of these components act to support the processor by relaying information, changing and controlling the data path based on the instruction, deciding whether or not to branch, and detecting hazards in the pipeline. These are important components that ensure that the processor runs smoothly. This CPU is used to run a C file called “image.c” which runs a basic lumen filtering on 32x32 square of pixels on an image.

Two major sets of changes were done to the original implementation. The first was the addition of a custom instruction. This instruction performs in hardware a function called “clipping”. Clipping in this context refers to the necessity to “clip” luminance values of pixels. Since the luminance of a pixel is represented by a single byte, the values can only range between 0 and 255. Therefore lumen values above and below the range are ‘clipped’ to 255 and 0 respectively. This was done by adding some hardware to the ALU which performed this function, and changes to the ALU’s control to properly call the instruction. The second was a change to the critical path of the processor. The critical path of a processor is the longest path that an instruction can takes. This determines the frequency of the CPU since it is necessary to wait for the longest possible instruction, the CPU cannot move on unless all potential instructions have been performed. This change moved the branch control module from the MEM stage to the EX stage. By doing so it was possible to know whether or not a branch was needed one stage earlier. Meaning that if it was necessary to branch, it would cause one less NOP or no operation. Although it did increase the length of the critical path, and therefore decreasing the clock frequency, it made up for it by decreasing the number of wasted cycles.

Even with these improvements, the processor still takes a long time to complete the filtering. As shown in Table 1. It takes 2,106,060 cycles to complete, with an execution time of 42.2 milliseconds, this can be greatly increased through some improvements. The task at hand is to analyze the current state of the processor, determine specific areas for improvement, and then implement them whilst ensuring that the functionality of the CPU is unchanged.

Priority List of Improvements:

When looking for potential improvements it first useful to understand what kind of improvements can be made. The execution time of a certain from can be calculated using $T_{exec} = N * CPI * t_{clock}$ where T_{exec} is the execution time, N the number of cycles, CPI the number of cycles per instruction and t_{clock} the time between clock cycles or the inverse of the frequency. It is only possible to affect the total execution time by changing the CPI, the number of cycles and clock frequency. Changing the CPI is a difficult but not impossible task. The CPI is affected by the amount of incorrectly predicted branches, the amount of read/write misses, and the degree of penalty for getting it wrong. This was actually already changed earlier when the branch control module was moved once stage earlier, which decreased the penalty of branching. Other improvements to the CPI can be done, such as makes attempts to more accurately predict whether or not to branch, this can be done by for example adding a 2-bit predictor. But implementing such a system is complex. Given that no previous lab work was done on it, it would be a difficult task to undertake. The next way to decrease the execution time is to reduce the number of cycles. Again this has already been done before. When adding the custom instruction, it was then possible to do a task that would take multiple instructions and therefore cycles. Therefore adding new instructions that do specific tasks, or multiples tasks at once would decrease the amount of cycles necessary. Finally shortening the critical path, and therefore increasing the clock frequency, simply makes the processor run faster allowing it to perform instructions faster. Decreasing the critical path could be done by doing some computations in parallel and optimizing existing functionality.

Since the previous labs showed examples of adding custom instructions and how to move/change modules. It is logical to start with similar improvements. Therefore this list of improvements was devised based on their difficulty to implement and the expected performance gain. They are ordered from highest priority to lowest.

Potential Improvements:

1. Custom division instruction
2. Forwarding
3. Removing the branch control module dependence on the ALU
4. Changes to the multiplication instruction
5. 2-bit branch predictor

Custom Division Instruction

Division by an odd number is a difficult task for a CPU. It has to do many computations to correctly compute the outcome. The existing method in the mMIPS is slow. The compiler has to break the problem down into multiple smaller problems. Currently the compiler will sequence a series of subtractions wherein the numerator is subtracted by the dividend. The amount of iterations is recorded, once the numerator is less than the denominator, the answer is then the amount of iterations and the remainder. So for example: 7/3 is computer as: 7 – 3 = 4, 4 – 3 = 1, 1 – 3 < 3. Therefore the answer is 2 remainder 1. This method of division works but is slow. Taking many cycles to compute large divisions of large numerators and small denominators.

Knowing this led to the conclusion that creating a different division method would save an immense amount of cycles, at least when running the image code. An entirely new method that account for any odd divisor, is possible but very difficult. A similar result can be achieved by creating a specific instruction for dividing by 13. Which is what occurs in the image filtering. To divide by an odd number it is possible to multiple by another fraction that estimates very closely the odd fraction. This fraction is constructed with an even denominator, making the division easier, as dividing by an even number is done by bit shifting and adding, which is far faster.

Implementing this method would be relatively simple. First, having already implemented a custom instruction it would be possible to follow the same method for this instruction. Second, it doesn't include adding any fancy hardware, but rather only necessitates hard coding in the multiplication and division values. The decrease in execution time would be immense. The divisions in the image file account for a large portion of the cycles. It is not easy to accurately estimate the increase in performance, but based on the assumption that division accounts for about 70% to 80 % of all cycles. It is so high because in the C code, the surrounding pixels are added together with some weights, and then divided. The range of the outcomes should fall somewhere close to 0 and 255. Meaning that as a worst-case scenario, the CPU would have to perform over 255 subtractions and even more supportive instructions to keep track of the information. With this large amount of work necessary, the expected decrease in cycles would be around 70% with a custom single-cycle division by 13 instruction.

With the relative ease and large performance boost, this improvement was placed first on the priority list.

Forwarding

As the CPU performs its normal functions it may encounter data hazards. One computation relies on the outcome of a previous computation that is currently in the pipeline. Due to the nature of the pipeline, the information will not be accessible in time and therefore cause a halting of the pipeline. Issues like this cause a rather dramatic slowdown of the processor, as it has to stop and wait for an instruction to pass through the rest of the pipeline before it can continue. These types of issues can be prevented by implementing forwarding. Forwarding is a system that will detect potential data hazards, and if possible relay the necessary information back to the correct stage.

This can be implemented by adding two additional muxes and some specific detection hardware into the hazard unit to control them, and the correct wires to relay the information. The hazard module will detect whether or not a hazard is going to occur and then control the muxes as necessary.

The implementation of this is not simple. It requires adding in very specific checks that will account for possible hazards. The muxes themselves may seem simple but correctly wiring them can be challenge. The upside is that it is a very useful improvement. It stops the occurrence of many data hazards, and provides a general purpose speed-up can that can applied to any type of program.

When looking at the deconstructed assembly code, it is obvious that this type of hazard occurs a large number of times. Meaning that the CPU stops many times during the course of running the image code. So therefore implementing forwarding would stop these from occurring, this would most likely decrease the amount of cycles necessary by as much as 45% depending on the occurrence of these hazards.

Based on the large decrease in cycles, but the potential difficulty in implementation this improvement was placed second on the priority list.

Removing the branch control module dependence on the ALU

When analyzing the mMIPS circuitry, it was noted that the branch control module was reliant on a computation that runs through the ALU. This makes it reliant on running a computation through the ALU. Which is unnecessary, as the computation done is a very specific comparison, which does not have to occur through the ALU. The ALU is used to check whether or not the data in the two registers are different or the same and then act accordingly. This would be an attempt to do more computations in parallel, by removing its reliance on the ALU. It can instead be done by connecting the input signals directly to the branch control module and allowing it to do the comparison, this could potentially shorten the critical path. This would allow for an increase in the clock frequency and therefore an overall improvement of the CPU.

This improvement would probably only decease the critical path by a 1.0 to 1.5 ns, as there are other long paths that data could take elsewhere. Most likely some other path through the ALU. The implementation would be rather simple though. It would only involve the movement of wires, addition or removal or ports and then implementing the comparison into the branch control module. Although this change would could only have affect if the critical path runs through the branch control module. Therefore it was placed third on the list due to its relative ease of implementation.

2-bit branch predictor

Branching is always an issue when it comes to pipelined CPUs. Effective pipelining is reliant on the CPU's ability to correctly fetch the next instruction. This means that being able to correctly predict whether or not to branch, each time it gets the prediction wrong it will suffer a number of wasted cycles as penalty, which can greatly affect a CPU's performance. Therefore it would be useful to implement a system that would attempt to predict whether or not to branch. This can be done with a 2-bit predictor. A 2-bit predictor attempts to predict branches by storing two bits to keep track of how often a branch is taken. Fig 1. Illustrates how the 2-bit predictor attempts to predict the branching.

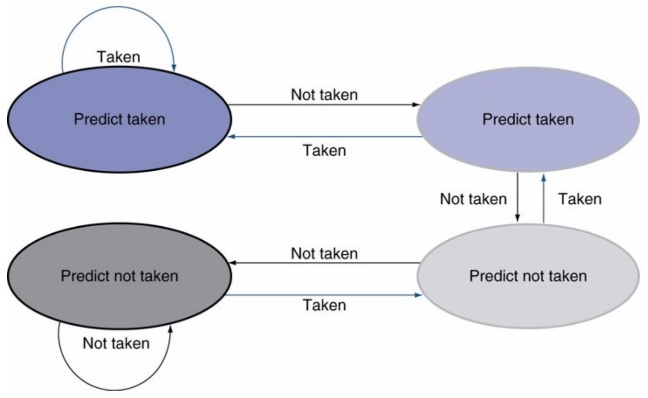

Figure 1: The states of a 2-bit branch predictor and the transitions between them

Although this system would a great system to add as it would be useful when any type of program, it requires some significant changes. Specifically a way to identify one branch from another. Which would necessity some further research as to how to implement such a feature, and a thorough overhaul of the existing branch control module.

Due to its difficulty to implement, this was placed last in the priority list

Implementation of Improvements and Testing:

Custom Division Instruction

To implement this instruction changes had to be made to the alu.v and aluctrl.v files, and the C code. It was first necessary to calculate the correct fraction that estimated the division by 13. This was done using the following equation: $\frac{1}{x} * 2^y \gg y$ where x is 13 and y is number that that was changed to find the correct degree of precision. This was done using a simple c program that looped through y until it got to a suitable precision. It was found that a y of 17 had a suitable level of precision. This means that if a number is multiplied by 10,083 or $\frac{1}{13} * 2^{17}$ and then bit shifted right by 17, which is the same as a division by two to the power of the amount of bits shifted. The result would be a suitable estimated division by 13 computation.

Now that the correct value have been attained it now must be implemented. First off the aluctrl.v file was edited to include an option function code 31 in hexadecimal or 49 in decimal. This option set aluctrl's output, ALUctrl to 15 in hexadecimal. The corresponding case option for 15 was then added to ALU as follows:

```
'h15: // custom instruction: division by 13

begin

        sign = s_int[31:31];        //stores the sign of the input

        c = 10083 * s_int;          //multiplies and the divides as a

        result = (c >> 17);

        result[31:15] = {sign, sign, sign, sign, sign, sign, sign, sign, sign, sign, sign, sign, sign, sign, sign,
        sign, sign};

end
```

Sign is an added register that stores the first bit of the inputted integer. This is done because the int are stores using 2's complement which uses the first bit to determine the number sign. This becomes an issue when bit shifting, as bit shifting will fill in those missing bits with 0's, which could change the sign of the integer. Therefore the first bit is then stored and later added back in during the bit assignment of the thirty first to fifteenth bits as equal to the stored sign. This maintains the sign of the inputted bit. The other two steps are simple, c = 10083 * s_int, simply multiplies the input by the x value that was determined earlier, and result = (c >> 17) bit shifting by the corresponding y value that was determined earlier. It is necessary to point out that use the use of the register c is very necessary because the size of the multiplication that is occurring. C is a register that has 64 bits as compared to the regular 32 of most other registers. Allowing for it to properly store the potentially very large number that will result in the multiplication step. The result is of course then stored back in the result register.

The next step was to change the code in the image.c file. This is done to indicate to the compiler to call the correct opcode. The changes was to replace the "/ 13" portion at the end of the weighted adding of the neighboring pixels, with

```
result =  ((result) - ((zero) + *(int *) 0x12344321));
```

What this does is take advantage of some unused opcodes know to the compiler. The compiler is looking for this specific structure of assignment and when it sees it, it interprets it as call to a specific opcode.

Now the aluctrl know when to call the new function, the alu knows what to do when asked to perform that function and the compiler is able to correctly identify when to use the function.

Table 1: A comparison of the cycles, frequency and execution time between the standard mMIPS and the mMIPS version with a custom division function.

mMIPS version	Number of cycles	Frequency (MHz)	Execution Time (ms)
Standard mMIPS	2,106,060	49.911	42.196
With Division	465,955	49.956	9.327
Difference	1,640,105	0.045	732.869
% Improvement	77.9%	0.09%	77.9 %

The addition of this instruction performed better than expected. The estimation of a 70% reduction in overall cycles was proven to be an underestimation. An incredible decrease of 77.9 % in both the cycles needed and execution time. There was also an unexpected slight increase of 0.09% of the frequency.

The difference between the estimated value and the measured value most likely comes from an underestimation of the number of cycles necessary to perform division. It obliviously took more cycles than previously thought.

To validate that the implementation didn't affect the functionality of the CPU, the outputted mips_ram.dump.hex file was compared to a reference file created before any changes were made to the processor. Using the provided memory.py python script it was determined that the two files were identical. Therefore indicating that the output is unchanged. Before any simulation occurred, the files were always checked by Xilinx for syntactical errors, therefore ensuring the code could be properly compiled and simulated.

Forwarding

Adding forwarding requires the addition of two extra muxes. These muxes control input to the ID_data_reg1 and ID_data_reg2. These muxes were created in the mmips.v file with four inputs each. By contolling the inputs of registers the muxes are then able to account for potential memory hazards that may occur. These are the controlled by the hazard module which was extended to have two extra output, one per mux. These muxes takes inputs from the ID, EX, MEM, and WB stages. The main decisions are made in the hazard module. The hazard module will compare the contents of the read registers with the address of the accessed data in later stages. For example this following piece of code checks for a memory hazard in the WB stage:

```
if (MEMWBRegWrite == 1'b1 && MEMWBWriteRegister == ifidreadregister1 && MEMWBWriteRegister != 0)

forwardA = 2'b11;

if (MEMWBRegWrite == 1'b1 && MEMWBWriteRegister == ifidreadregister2 && MEMWBWriteRegister != 0)

forwardB = 2'b11;
```

The first check (MEMWBRegWrite == 1'b1) checks if the data currently in the WB stage will be written. Next MEMWBWriteRegister == ifidreadregister2 checks for if the destinations are the same, if that is true that means that the address of data that will be accessed is the same as the data that will be written to at a later stage. The last check, MEMWBWriteRegister != 0 is to ensure it is an actual location and not a defaulted value. Based on the evaluation of these statements. The forwardA and forwadB outputs are changed appropriately to transfer the information back such that this hazard does not result in a stall.

For each stage comparsion such as the one shown are done. The important aspect to check for is to check for dependencies. If later instructions are dependent on the outcome of earlier instructions, then there is a hazard. That is where the forwarding muxes must correctly route the information if possible.

Table 2: A comparison of the cycles, frequency and execution time between the mMIPS version with a custom division function and the mMIPS with forwarding implemented.

mMIPS version	Number of cycles	Frequency (MHz)	Execution Time (ms)
With division	465,955	49.956	9.327
With forwarding	236,570	54.958	4.305
Difference	229,385	5.002	5.022
% Improvement	49.23%	10.01%	53.85%

The results of implementing forwarding were better than expected. The cycle reduction prediction of about 45 % was close, but again the implementation outperformed the prediction. This is most like again due to a higher occurrence of these hazards then estimated. The change of the critical path was unexpected change. If anything predictions would most likely estimate an increase in the critical path, since there is an addition of logic. A sizeable increase of 10% was measured, which is difficult to explain as there is no reasonable explanation as to why the addition of logic will lead to a decrease in the length of the critical path.

Just like the previous improvement, the output mips_ram.dump.hex file was compared to the reference file. Unless they were deemed an exact match by the memory.py script, they improvement was not considered valid.

Non-Implemented improvements

The rest of the improvements were not implemented because of the time requirements of the assignment. Given more time, perhaps more improvements could have been made.

Conclusions:

After each improvement it was clear that the estimations were underestimating the potential gains in performance. This discrepancy is most likely do to the inability to correctly count the amount of cycles need to perform a certain task, like in the case of division, and the difficulty to estimate the occurrence of hazard, and the impact they will have. Although the predictions were somewhat accurate, they also lacked the ability to predict the changes in critical path. It is clear now that the critical path is an unpredictable aspect.

It is important to point out the usability of the division instruction. It may have been a great improvement when running the image.c file but that is exactly its only purpose, to reduce the runtime of this specific program. In nearly all other programs this instruction would be completely useless. Although it does show the power creating highly specialized instructions for very specific functions.

Forwarding on the other hand proves that hazards have the ability to slow down a processor immensely, and subsequently the prevention of those hazards speed up the processor a by a great degree. The addition of these general purpose features will help to make the processor better at running all programs.

In conclusion, the two improvements implemented represent two different ways to improve the CPU's performance, specialized functions and general improvements. Whilst specialized functions are more powerful when it comes to reducing the execution time, they lack the flexibility to be applied elsewhere beyond a very specific range of programs. The general improvements require more effort but have a far greater range of applications.