Jordan Fox
Jacob Gosse
CPRE 381
5/2/19

MIPS Processor Report

Introduction

Throughout this semester, we have learned a great deal about MIPS processors. We learned about the single cycle MIPS processor, which would run one instruction per cycle. To a pipelined MIPS processor, that could implement multiple instructions at a time. And to a final pipelined MIPS processor with hazard detection and forwarding. These processor designs all have characteristics that make it perform differently as seen in the results of the benchmark tests.

Results

MIPS Processor 2b	# Instructions	# Cycles	CPI	Max cycle time	Execution time
Synthetic benchmark	31	31	1	42.5 ns	1317.5 ns
Bubblesort	222	222	1	42.5 ns	9435 ns

MIPS Processor 3a	# Instructions	# Cycles	CPI	Max cycle time	Execution time
Synthetic benchmark	50	59	1.18	21.4 ns	1262.6 ns
Bubblesort	551	583	1.05	21.4 ns	12476.2 ns

MIPS Processor 3b	# Instructions	# Cycles	CPI	Max cycle time	Execution time
Synthetic benchmark	31	41	1.32	23.8 ns	975.8 ns
Bubblesort	222	305	1.37	23.8 ns	7259 ns

Discussion

Based on the tables above it can be seen how the different types of processors demonstrate different characteristics that define how it handles different instructions. The most basic type of processor is the 2b, the single cycle processor. As the name implies, it uses a single cycle for every instruction. This has the unique effect of having a CPI of 1 regardless of what instructions are fed through it. Despite this consistent performance, it comes with the downside of having a long cycle time in comparison to the other processors.

An improvement on this cycle time is done by introducing the pipeline in processor 2a. This greatly reduces cycle time at the expense of making the processor vulnerable to many data hazards. Theoretically, the only time a cycle is done which does not execute an instruction is when a jump or branch is done and that instruction must be ignored. Other than these instances where the CPI is 2, all other instructions have a CPI of 1 which is only true in a pipeline sense as a single instruction actually takes 5 cycles to complete. This means that the only instances of a CPI greater than 1 are the 4 cycle startup, as well as branches and jumps. This is why the CPI for processor 3a executions is very close to being 1.

It is critical to point out that this impressive CPI and short cycle time do not tell the whole story. Because of the possibility of data hazards. The software scheduled pipeline solves data hazards by rearranging instructions when it can and by inserting NOP instructions when that isn't possible. If all data hazards can be resolved by rearranging instructions then this processor achieves its maximum performance by being fast and efficient. This is rare to accomplish absolutely, however. Thus in most applications, NOPs are inserted which execute in 1 CPI but they are a complete waste of time. That is why despite its impressive specifications, the software scheduled pipeline ends up executing far more instructions than intended which cause it to have the worst execution time compared to processors 2b and 3a.

An improvement on the software scheduled pipeline is the processor 3b the hardware scheduled pipeline. This processor, compared to processor 3a, tries to overcome data hazards by forwarding and automatically stall and artificially create a bubble of NOP instructions when data hazards are present which cannot be solved by forwarding. This has the result of a relatively large CPI compared to the other processors but only because of this characteristic which allows it to achieve a very quick cycle time while still using unmodified software to avoid data hazards. This allows it to have the best performances in most cases because of this "best of both worlds" approach to executing the given software.

As far as their comparable performance goes it is apparent that in terms of execution time, processor 3b is the best in both cases. It is worth noting that it is the difference in performance is not always definite since processor 3b is a 5 stage pipeline but the cycle time is only half the time roughly. It can be deduced that any instruction that takes greater than 2 CPI would be quicker to run on the single cycle processor 2b. In practice, however, the hardware

schedule pipeline processor tends to be faster for most software. In addition to the lost cycle for branching a jumping, the hardware scheduled pipeline tends to encounter more data hazards when doing conditional branches and loading from memory.

This case of the hardware scheduled pipeline being most likely to encounter data hazards when performing conditionals for branches and when loading from memory. This very thing happens frequently in the first part of the bubblesort code which starts by scanning the size of the number array by loading a word and then immediately comparing it to 0. This data hazard which was made with no intention of being a hazard is a great loss of processor time and helps to explain why the hardware scheduled pipeline processor is only 23% faster than the single cycle processor. In the synthetic benchmark code, this happens much less often and the result is that the hardware scheduled pipeline is 26% faster than the single cycle processor. This again comes from the disparity in characteristics between the hardware scheduled pipeline which has a performance that depends on the type of instructions it receives. The single cycle processor performance is directly proportional to the number of instructions executed in runtime.

Optimization

To optimize software for each processor, the characteristics of that processor define what methods can be done to improve performance time. For processor 2b, the single cycle processor, the most important characteristic is the namesake of the processor that is the fact that every instruction is done in a single cycle. This means that the cycles per instruction (CPI) will always be 1 and for all executions. Given this, it is implied that for a given cycle time C, and software containing N assembly level instructions, the execution time will always be C*N for any software. Given this direct relationship between the number of cycles and the execution time, it can be said that the only way to optimize a software would be to reduce the number of instructions to be as few as possible. This can be straight forward for a linear code, however, for software containing conditional branching where the conditions depend on given arguments, knowledge of software applications must be employed to make the average execution of the code use as few instructions as possible in order to optimize it to the single cycle processor.

For processor 3a, the software scheduled pipeline processor, it is again the namesake of the processor that reveals the most effective way to optimize software to run on it. The instructions for this processor must be scheduled in such a way, generally done by a compiler, to reduce data hazards, which are alleviated by the insertion of NOP instructions. This kind of optimization can be done by rearranging instructions, whenever possible, to minimize the number of NOP instructions needed. Having NOP instructions in software adds numerous cycles to the overall execution time so performance can be improved by having as few as possible. The strength of the software scheduled processor is it's extremely short cycle time, the quickest of all

3 processors. To capitalize on this strength is to not let it be smothered by a large quantity of NOP instructions.

For processor 3b, the hardware scheduled processor, it has to ability to overcome some data hazards by forwarding data ahead in the pipeline. It does this when possible but an hazard still arises when the data it needs isn't yet available. The processor solves this by stalling until the needed data is available and then forwarding it. This is, along with the one extra cycle which is lost with every jump and branch instruction, is the only thing that prevents this processor from having a perfect 1 CPI. As a software optimization, preventing the code from activating this comparably much narrower set of cases which produce true data hazards is the best way to maximize the performance of software with this processor. The structures of these processors are all far from perfect and can definitely be improved. All processors are built so that they minimize the number of lost cycles that occur when a branch or jump occurs. That said, the critical path points to ways that the cycle time can be shortened. For the 2b processor, the critical path was a strange implementation of sltu in the ALU which involved 32 consecutive multiplexers in a single path. This was thought to be clever when it was first written but the synthesis revealed how false that was. A better implementation would be to set sltu to make a decision based on the most significant bit of the two comparands when they are different and otherwise use the same result of slt. This optimization implemented for later versions of the processor 3a and 3b.

For processors 3a, after employing a better implementation of the sltu logic, the critical path remained to involve the ALU. This time it was the regular adder logic which was the critical path. It became apparent at this point that the critical path tended to be any logic path that needed to propagate horizontally across the 32 bits of the data width. Given this pattern of the critical paths in processors 2b and 3a, it seems the best way optimize this section of the processor is remove the ripple carry adder in favor of a carry look ahead adder which does not contain a horizontal path across the data width and instead declares every bit of the output as a direct combinatorial expression of all the bits from both inputs plus the carry in. Unlike the sltu, this optimization was not implemented for later versions of the processor.

Processor 3b has its own quirk which allows for optimization. This does not come from the critical path as it remains to advocate a better implementation of a 32-bit adder. Instead, this comes from the process of adding different components over time. In this case, there is a strange communication that occurs between the control unit and the hazards detection unit. The control unit, of course, reads the instruction and generates several control signals. Among those signals is the branch and jump signals which are read by the hazard detection, which is jointly implemented with the forwarding unit since they read a lot of the same information. The hazard detection uses the jump and branch signals to set a context towards which defines what and when hazards are present. The issue is that when a hazard is detected in the process of either a conditional branch instruction or a jump register instruction, the processor must prevent the jumping or branching until it has the information it needs to decide what to do. When this happens the hazards detection unit, of course, sends out a stall signal which halts the necessary

registers. This signal also tells the control unit to mask all meaningful control signals like the write enables for the register file and memory. This signal cannot, however, mask the jump and branch signals since they are needed at the hazards detection unit (there is a separate signal used to block all control signals including branch and jump which is used when branching, jumping, and after resets) so the hazard detection must filter and output the jump and branch signals to be used by the processor. This means there is a path from the control unit, to the hazard detection unit and then back to the control unit. One possible optimization of this processor would be to combine the control unit into the hazards detection and forwarding unit to simplify this overly complex process.

Performance Disparity

As previously mentioned there is a disparity in the characteristics of processors 2b and 3b which creates a case where one has performance dependant on the type of instructions executed and the other has a direct linear relationship between execution time and instructions executed. In most cases, the hardware scheduled pipeline performs faster as seen in the two programs used for testing including bubblesort which processor 2b performed in 9435 ns while processor 3b performed that same code in 7259 ns making it the victor in that application since it ran 23% faster. That's not to say that the hardware scheduled pipeline is always better. One simple way to demonstrate this is to run short programs which take advantage of the startup lag of the 5 stage pipeline. For example the test code add_1.s contains 3 instructions but it executes in 7 cycles for processor 3b taking 167 ns, compared to 127.5 ns for processor 2b it is apparent that the single cycle can sometimes win out. This particular situation, however, only demonstrates the startup delay of the pipeline which is hardly an issue in long programs. Despite this, there are greater faults in the design of processor 3b which cause the single cycle processor to win out.

For this situation we wrote a longer code designed to break the pipeline by creating data hazards rapidly and severely, this code was able to make the hardware scheduled pipeline perform very poorly compared to the charismatic underdog that is the single cycle processor. This code is titled smackpipe.s and it rapidly creates the most severe data hazard possible in processor 3b. This is possible since the comparison for conditional branching is done in the ID stage of the pipeline, this means that data loaded from memory must reach the WB stage before it can be forwarded to this part of the pipeline. This is complemented by the cycle which is ignored immediately following branches and jumps. So this code simply creates 8 instances of data being loaded from memory and immediately used for a conditional branch. This code contains 20 instructions and thus the single cycle processor executes it in 20 cycles which takes 850 ns. The hardware scheduled pipeline, however, needs 48 cycles to run this software, that gives it an enormous 2.4 CPI and it takes it 1142 ns to complete execution. This means the single cycle processor is actually 25% faster than the hardware scheduled pipeline to process this code.

Reflection

The first critical challenge that we faced was learning about VHDL syntax. VHDL syntax was crucial in debugging some problems in the code for VHDL. The syntaxes would help to fix certain errors that came up in the pipeline. We had to learn about these syntaxes to help resolve errors or warnings in our code. To help resolve this in the future, we will do more research into VHDL syntaxes.

The second critical challenge was implementing pipelining. Pipelining wasn't that difficult but it was more about getting the proper signals to each part of the pipeline or not forgetting a certain signal. There were a few times that we forget to change a signal to be implemented into the pipeline, which affected one of the instructions in the pipeline. To help organize the signals that we needed for pipelining, we used a notepad file to organize each signal and the bits used for each part of the pipeline. Another issue we had with pipelining was the issue of the NOPs being outputted after resetting the processor or after branches and jumps. This issue was resolved by using a one bit single to control when to read the NOPs into the processor. These issues could be avoided in the future by creating a document to organize signals and how the signals are used for pipelining.

Data hazards/forwarding equations - many ideas came and went, difficult to debug

The final issue that was the most time consuming was the data hazards/forwarding equations. This was very time consuming on debugging if a data hazard was found, if forwarding was active, or the data hazard found sends out the right signals to implement it. We had multiple times where forwarding wasn't being implemented or a data hazard was found. This means that our equations were off for both. This was solved by implementing the forwarding and data hazards together. This would mean that the equations of both data hazards/forwarding were set up to have the data hazards override forwarding from happening. This means that if a data hazard is found, then forwarding wouldn't happen. In the future, this issue could be fixed by getting the proper equations for the data hazards and forwarding equations.