EECS 470 Computer Architecture
13 December 2006
Term Project Final Report

Group 3 Members
Joseph L. Greathouse (jlgreath@umich.edu) – EECS (G)
Andreas Moustakas (andmoust@umich.edu – EECS (G)
Carolyn L. Phillips (phillicl@umich.edu) – Applied Physics (G)
Nikhil C. Rao (ncrao@umich.edu) – EECS (G)
Dan Zhang (danz@umich.edu) – EECS (U)

Final Project Report

This report documents the design of the DUCHESS microprocessor. DUCHESS is a single-issue out-of-order 6-stage processor optimized to a set of relatively small benchmark programs. It implements out-of-order algorithm using a reorder buffer, reservation station, register alias table and retirement RAT. It takes advantage of a gshare branch predictor, a return address stack, and a prefetch unit. This processor implements a reduced Alpha instruction set, and successfully synthesizes to a clock cycle of 14 nanoseconds. The following report will discuss each of the design features in greater detail and explain the testing methodology used to verify correctness in its implementation. Finally, this document will analyze the performance of the design features.

The conclusion of our group is that DUCHESS is correct in its operation (i.e. it passed all tests applied), synthesizes, and has a reasonably fast execution time for the provided benchmarks. We believe that the most benefit was derived from our branch predictor, branch target buffer, and return address stack. These reduced penalties branch/jump significantly. We also benefited from our load/store queue, caches, and pre-fetching schemes. These reduced the impact of memory latency. Some of our design choices, such as committing two instructions per cycle, did provide a small performance benefit, but also made our design more complicated to implement.

Given extra time to rework our pipeline, we would optimize some of our modules to improve clock cycle time. Although they themselves were reasonably fast, some modules were accessed serially, rather than in parallel, with respect to each other. For example, the RAS could have been implemented in parallel with the IF stage at the cost of decreasing RAS prediction accuracy by a small amount. There are some simple optimizations, such as a larger data cache, that we could have implemented with only a short amount of extra time. We would also possibly modify our prediction scheme to handle a broader range of programs, and we would reduce the complexity of a few of our pipeline stages.

The components of the pipeline, as illustrated in the attached Figure 1 are delineated in Table 1. The letters (A, N, D, C, J, ALL) next to each indicates the architecture engineers chiefly responsible for the design. The breakdown of work at the end of this report explains this in more detail.

Table 1: Modules of the DUCHESS Processor

Prefetch (D)
Return Address Stack (D)
D-Cache (J,A,D)
I-Cache (N)
Branch Predictor (A)
Branch Target Buffer (A)
Instruction Fetch (N)
Issue Queue (N,D)
Decode (C)
Rename (C)
Reservation Station (A,J)
RAT/RRAT (A,J)
ROB (ALL)
PRF (D)
Five Execution Units
An Adder Unit for Logical Operations (J)
An Adder Unit for Arithmetic Operations (J)
A 4-Stage pipelined Multiplication Unit (J)
A unit for Branch Logic and Addresses (J)
A unit for Memory Operation Addresses (C)
Load-Store Queue (N,C)
CDB Queue/CDB (D)
Commit (C)

Major Design Features

A. Out Of Order Implementation – 'Tomasulo's III'

The DUCHESS Processor implements the 'Tomasulo's III' algorithm. This includes the following components:

1. A Reorder buffer (ROB) capable of holding 32 entries.
2. 16 Reservation Stations (RS) - These reservations stations are interchangeable. On each clock cycle, a rotating priority selector chooses between each of the available reservation stations. The code that handles the reservation stations is also where we handle CDB data arriving at the same time as a rename (i.e. "ships passing in the night").

3. A Register Alias Table and Retirement Register Alias Table (RAT/RRAT) module that issues Physical Register to Architected Register renaming. This module also tracks which physical registers currently contains valid data. The RAT maintains a list of the PRs free for renaming.
4. A physical register file that updates the values associated with each PR from the Common Data Bus (CDB).
5. A Rename Module that connects the outputs of the Decode stage to each of the above modules. This module also communicates instructions to the Load-Store Queue. Rename sends write signals to RS, ROB, RAT, Load-Store Queue (LSQ), or stalls Decode and the Instruction Queue (IQ) if there are no reservation stations, ROB entries, or LSQ entries available.
6. A Branch Or Store (BORST) Queue that sends a signal that it is acceptable to issue a store to memory when there are no halts, unresolved/mispredicted branches, illegal instructions, or other stores that precede the specific store in program order.
7. A Commit Logic Module - This module listens to the CDB and can send commit signals each cycle to the ROB and RAT/RRAT, based on up to two instructions from the top of the ROB. Also, in the case of a branch mispredict, it sends the squash signal to the entire pipeline and the new PC to the Instruction Fetch (IF) unit. Commit is able to retire two instructions every cycle.

The ROB, RS, and RAT are all examples of multiple write-port memory. The ROB uses non-conflicting write signals to write to register fields. The RS and RAT use arrays of modules.

One of the design requirements for DUCHESS was a reasonable method for handling exceptions, illegal instructions, and interrupts even with our out-of-order implementation. Our design will detect an illegal instruction entering the system, will halt when the instruction reaches the top of the ROB, and will assert an "illegal" output signal. In the case of an asynchronous interrupt, the ROB will be permitted to empty itself before control of the processor is handed over to the operating system. Currently mispredicted/unresolved branches, halts, and illegal instructions will prevent a store from being issued to memory, thus maintaining architectural state. If our set of operating instruction were to include instructions that could throw exceptions (special load types, divides), these instruction would also block store instructions from being issued to memory, by also being written into the BORST Queue. This results in somewhat unrealistically high performance statistics, since exceptions would increase the delay of memory operations before being issuing a store to memory.

B. Out of Order Load Store Queue

The major motivation for an out-of-order pipeline is to hide the latency associated with loading data from memory. As a result, it is desirable to be able to load and store data from memory as out of order as possible without compromising correct architectural state. In the DUCHESS, load and store operations are near optimal "out-of-order".

Stores set architectural state. As such, they may not occur out-of-order and may not occur when the instruction path is still speculative. Loads may occur when the instruction path is speculative, as they do not change architectural state, but if loads are sent out-of-order, they may return incorrect data. These problems are handled in the following way:

1. The BORST queue holds unresolved branches, halts, or stores not sent to memory. The top instruction of the queue is removed if it is a branch that was correctly predicted, or if it is a store that was sent to memory. A store is only sent to memory when it is the top of the BORST queue and the top of the LSQ.
2. Loads are sent to memory when their addresses are calculated. (The exception to this is if there is a store that may be sent to memory in the same cycle it is sent instead of a load. In this case, the "skipped" load will not be sent to memory until it reaches the top of the Load-Store Queue.)
3. When a load or store reaches the top of the LSQ and it has either received data from memory, in the case of a load, or been accepted by memory, in the case of a store, it is ready for the CDB Queue. Before a store is sent to the CDB Queue and removed from the LSQ, its data is put in a content-addressable memory (CAM) module. Before a load is sent, it checks the CAM module, and if a memory address matches, it replaces its data. In this way, wrongly loaded data is repaired.

The LSQ is an example of multiple write port memory and uses a set of non-conflicting signals on the positive edge to change the values of registers. It was implemented with eight entries. The size of the load-store queue was limited by the size of the CAM. If the LSQ was filled with store operations, the CAM must contain the address and data of each store operation to repair a hypothetical load issued to memory before all the store operations. This means the CAM must have at least as many address fields as LSQ entries. Since the CAM must be small enough to be searched every cycle, it must be relatively small.

C. Instruction Fetch System

To prevent bottlenecks in the front end of the processor, we implemented a next-line instruction fetch system. Instead of fetching a single instruction every clock cycle, our instruction fetch (IF) stage fetches an entire line of 2 instructions. These instructions are stored in a 16-entry issue queue (IQ). Therefore, should the instruction cache miss or if the data cache has control of the memory bus, only IF needs to stall instead of the entire pipeline. The rest of the pipeline continues to process instructions by removing them from the IQ. The rest of the pipeline may eventually be forced to stall if the IQ empties.

D. Pre-Fetch

Given the small size of our target benchmark suite, we chose to implement an aggressive prefetch system to minimize instruction cache stalls. Our design goal was to maximize our prefetch accuracy and aggressiveness while keeping the module simple. Instead of a simple next line prefetcher, we chose to implement a hybrid prefetcher utilizing both next-line and target-line prefetching techniques. Our prefetcher consists of two phases: the next-line warm-up phase and the target-line main phase.

At program startup, the processor misses the first cache request and stalls. The prefetcher takes advantage of this period of inactivity through use of the warm-up phase. During this phase, thepPrefetcher requests the next line from memory every cycle for 4 cycles. Therefore, after the initial request returns, the I-cache will have already been warmed up with the first 14 possible instructions.

After the warm-up phase, the prefetcher enters the main phase, utilizing a fully-associative 16 entry line target prediction table. The table is updated on a cache miss or a branch mispredict. On a cache miss, the table records the last block requested and the current block being requested. Since the cache miss is most likely due to a jump in the program, the table essentially contains a relatively accurate prediction of the next line requested by IF. In the common case of a program looping, the line target prediction table prevents the prefetcher from requesting useless data from memory and evicting useful data from the cache.

During the main phase, our goal is to implement the most agressive prefetcher possible without evicting useful data. Therefore, the prefetcher is only active upon an instruction cache miss. On a miss, the prefetcher predicts the next three blocks using the line target prediction table, predicting one block per cycle. Since the data cache and instruction cache are also competing for the memory bus and have priority over the prefetcher, should the memory bus be unavailable for the prefetcher's request, we store the prefetcher's request in a small queue. When the cache miss returns, we clear the queue, so the prefetcher will not request stale data.

We also chose to implement a similar, but smaller, prefetch unit for the data cache. Rather than use a larger block size, we chose to keep a block size of 64bits and implement a prefetch unit. This decision not only kept the cache fast - doubling cache line size added more than 1ns on synthesis - but also made the cache much more flexible. For example, if our code sample looped over a block of code that loads every other line into the PC, a data cache with a 2 entry line size would delay every other memory access, while a data cache with a single entry line size and a prefetcher would not delay after the initial warm-up phase. The prefetch method also prevents the data cache from requesting useless data in the case of an unaligned cache request.

Just like the instruction prefetch unit's main phase, on a cache miss, the data prefetch unit will request the next block based off a prediction from another 16 entry, fully associative line target prediction table.

E. Return Address Stack

A branch target buffer is not designed to predict the next address of a return operation. Not knowing what the next address will be until the register value has been read, a processor is usually forced to squash, and flush the pipeline when a return operation reaches the head of the ROB. We implemented a return address stack (RAS) to improve our processor performance in these cases.

It is possible to exit the main body of a program using instructions such as jump to subroutine (jsr) or branch to subroutine (bsr). Return (ret) is generally used to return to the main program. When the commands jsr and bsr enter the instruction fetch stage, the RAS pushes the PC address onto a stack. When the ret operation enters IF, the RAS module pops the top address off the stack. This address then becomes the next address fetched.

However, standard branches - branches/jumps that are not jsr, bsr, and ret - can cause the RAS to become misaligned when a jsr, bsr, or ret enters the pipeline and then gets squashed. Misalignment is a large problem for the RAS, since the address is predicted by popping the RAS. A single misalignment essentially causes the entire RAS to become invalidated. To correctly implement a RAS, we must be able to undo speculated pops and pushes when a mispredict occurs.

To solve this problem, we implemented a check-pointing scheme as suggested in [1]. Whenever a standard branch passes through the IF stage, we save a copy of the Top of Stack (TOS) pointer and the TOS contents in the RAS buffer. At commit, if the branch was incorrectly predicted, then we restore the TOS pointer and top entry of the RAS. Through use of this check-pointing scheme, Skadron et al. suggests that we can achieve a significant increase in RAS prediction accuracy. For example, in the SPECint benchmark 'go', an increase of RAS prediction accuracy from 43.2% to 99.4% was observed. The worst-case scenario was with the SPECint benchmark 'perl', in which the RAS prediction accuracy increased from 66.4% to 96.3%.

The other main limitation of the RAS is its finite size. If the number of jsr and bsr operations entering the system exceeds the size of the RAS, some predictive capability will be lost. We chose to continue pushing addresses onto the stack by overwriting the oldest RAS entry. As a result, the correctness of most of the RAS is preserved. To further prevent RAS incorrectness, we chose a large RAS size of 16 entries. The RAS buffer uses a similar overwrite technique and contains up to eight entries.

As can be seen in Figure 2, the RAS had a substantial impact on the performance of objsort.s and fib_rec.s, the only two programs to implement bsr/jsr/ret operations. Combined with the branch predictor and branch target buffer, branch prediction accuracy of the direction and destination of conditional and unconditional branches jumped to 96% for objsort.

F. Branch Predictor and Branch Target Buffer

The penalty associated with squashing the pipeline due to a mispredicted branch is very high. Therefore, our goal was to implement a branch predictor with a high level of accuracy to reduce that penalty. We chose to implement a branch predictor using a hybrid bimodal-gshare algorithm that per [3] is reputed to be highly accurate.

The predictor consists of two components: a bimodal predictor and a gshare predictor. The bimodal predictor has a table of 2-bit saturating predictors which is indexed by the PC of the incoming instruction. The gshare predictor XORs an array of global branch histories with the PC (minus the bottom two bits) of the branch and uses the value to index a table of 2-bit saturating predictors.

If there exists a conflict between the prediction of the bimodal and gshare Predictors, the conflict is resolved by adding the two predictions, each of value 0, 1, 2, or 3, and predicting a branch is taken if the sum exceeds a threshold value of 2. In this way, if one predictor is predicting strongly taken, the branch will be predicted taken.

If a branch is predicted taken, but destination address is not known, it must effectively be treated as not taken by the pipeline. Thus, the BTB is indexed by the PC and keeps the destination of every branch, barring aliasing, that has entered the pipeline and had its address executed, whether it is taken or not or squashed.

There is a substantial delay between the prediction of a branch and the resolution of the branch. In the meantime, other branches may have entered the pipeline and must be predicted. We chose a scheme by which the predictor is speculatively updated at the time the prediction. In our scheme, each branch carries its indices and 2-bit predictor values for the gshare predictor and bimodal predictor and its branch history register pointer. In the special execution unit just for branches, both the value of its register, if any, and the predicted PC destination are checked. Being wrong in either generates a mispredict (not a squash) signal. After the branch has been processed by the execution unit, this information is sent back to the branch predictor so that the branch predictor can update itself relative to the speculation, and the address is sent to update the BTB.

In our final implementation, we chose to turn off the Bimodal predictor and use only the gshare 1024, which uses 10 bits of global branch history to make branch predictions. As is discussed in the analysis section, our implementation of the gshare/bimodal scheme performed aggressively for some programs but was not an ideal implementation for others.   Whether speculating or not speculating, many branches flowing through the pipeline at a time compounds the error in the gshare or bimodal scheme. And when a branch is mispredicted, the incorrect branch history paths that have updated the global history and bimodal predictor are not undone, as we have not implemented a check-pointing scheme.

G. Cache Description

Both the instruction cache and the data cache are designed as two-way set-associative caches with line sizes of 4 bytes. The final sizes of the caches are 128 bytes each. We wanted to determine the size of the caches by the maximum size that fit in the runtime of DUCHESS, but ran out of time to implement this optimization. Our data cache is write-through. It is also no-write-allocate, due to each line being the same size as a piece of written data.

The out-of-order LSQ and data cache together act as a non-blocking cache. The LSQ tracks which load in its queue has its address ready and data ready and sends them to the CDB in order. It does not need to receive their data back in any particular order from the data cache for this scheme to work. As a result, the OoO LSQ can handle as many load misses at a time as it has LSQ entries.

H. Improved LRU Algorithm

A few of the structures in DUCHESS, such as the BTB, use a fully associative cache structure. Therefore, to achieve high performance and utilize the cache efficiently, an effective cache eviction algorithm is required. Instead of implementing the pseudo-LRU replacement algorithm (PLRU) described in lecture, we chose to implement the modified pseudo LRU replacement algorithm (MPLRU) described in [2]. The MPLRU uses a tree structure identical to that of PLRU. However, when a block is required for replacement, instead of using the current history bit at the top of the tree, MPLRU uses the previous history bit to make the decision. Ghasemzadeh et al. show that MPLRU shows an average improvement of 8.52% in cache miss rate over PLRU. The additional cost is small - only a single additional register per cache set. Therefore, due to the large improvement in performance versus the negligible increase in complexity, we chose MPLRU over PLRU.

Testing Methodology

The following methodology was used to build up and test the DUCHESS Processor.

1. Testbenches were written for each module that tested standard behavior and a variety of corner cases (Although, for most modules, it was later discovered that there were corner cases that were not considered).
2. The back-end of the pipeline was constructed and was fed sets of instructions decoded by hand. As modules were added, new type of instructions were sent.
3. The front end of the pipeline was constructed and tested independently.
4. A "dumb" front end that assumed not-taken for each branch or jump and zero latency memory access was attached to the backend with a simple load-store queue that communicated with a separate zero latency copy of memory. First simple codes were run. When these were implemented successfully, the provided 18 test programs were run. The instruction path order and memory outputs of the un-pipelined Project 3 in-order processor was used for determining correctness.
5. The out-of-order LSQ was attached and the 18 test programs were run through again, while changing the latency of the data memory.
6. The branch predictor and target buffer was then attached to the front end and each of the 18 test programs were run through again. With the reduction of squashes, the registers within the system started to fill and significantly new dynamic states within the processor were generated.
7. The latency of the memory unit attached to the LSQ was changed from 0, 1, 2, ..., 10 to induce new dynamics states.

8. As new modules were added (Icache, Dcache, prefetch, RAS etc.) the suite of test programs were run and outputs checked.
9. Analyses of statistics regarding pipeline behavior were used to see if the pipeline was performing to expectation. In some cases where it was not, bugs were found that did not affect correctness but did affect performance.
10. The aggressiveness of various modules were increased and decreased to introduce new dynamic states (aggressiveness of prefetch, prediction scheme) and the suite of test programs were run and the outputs checked.

Analysis

A. Sizing of ROB, RS, LSQ

Figure 3 graphs the utilization of the Reorder Buffer for the 17 test programs provided. During the progress of our project, it was observed that as our predictor performance improved, the ROB became more filled with respect to individual programs. When our prediction performance was poor, most of the programs could have been executed with a ROB of eight entries without resulting in a stall. Or in other words, when the frequency of squashes was high, no structure in our pipeline was highly utilized. Even as our predictive capability increased, our ROB, sized at 32 entries, was large enough to never stall the pipeline for most of the programs. From Table 2 below, we see that while the ROB size did cause the pipeline to stall for fib_rec, objsort, and parsort, it was not the major cause of pipeline stalls. We conclude that our ROB is well sized for this application.

The number of reservations stations and the number of LSQ entries, on the other hand, appear more limiting to program performance, Table 2. Again, statistics gathered while the pipeline was being put together indicated that these two modules did not cause stalls until a minimal level of functionality had been reached. For example, without the RAS, the LSQ did not become filled while running the program fib_rec.s or objsort.s.

Table 2: Number of Cycles Module Was Full

FULL	ROB	RS	LSQ
fib.s			12
fib_rec	33	10	304
insertion			12
mult		425	
objsort	2	16503	19321
parsort	26	6135	
saxpy			134
sort			23

B. Out of Order LSQ Handler

The impact of the out of order LSQ was evaluated by attaching the LSQ directly to a memory of fixed latency. In the attached figure 4, the ratio of cycles required for out-of-order versus in-order LSQ operation as memory latency was increased was measured for the test programs (except for halt.s). Note that as there is no data cache between memory and the LSQ, the trends noted do not indicate directly what speedup would be expected. The trends do indicate the relationship between latency and out-of-order load issue, and also indicate which programs are significantly improved and which are not.

Not surprisingly, many of the programs are not affected by out-order load issue at all. For example, since evens.s and evens_long.s only store numbers, OoO LSQ does not speed up these programs. The same is true for btest1.s, btest2.s, parallel.s, parallel_long.s.

On the other hand, if we compare fib.s with fib_long.s and copy.s with copy_long.s, we can see the effect of inserting streams of NOPS into the instruction stream. Latency in memory is hidden by the cycles associated with issuing the stream of NOP instructions.

The programs sort.s, objsort.s, parsort.s, which sort lists of numbers, repeatedly read data from and write data to memory. Thus these programs can be significantly impacted by the time required to load data, and benefit from being able to issue loads out-of-order. The benefit to objsort.s appears to saturate at about a latency of 2. This is probably because the RAS was not installed in the pipeline when this test was run and the squashes that therefore would follow the many "ret" instructions would have created a delay in the execution of the program between the loading of data and the use of data.

Copy.s, which loads data and then immediately stores it elsewhere, benefits the most from out-of-order load issue, since anything that helps separate those two operations for high latency memory will allow the program to finish faster.

Saxpy.s, one notes, actually managed to operate slightly faster with the in-order load issue. We conclude that the latency of memory managed to induce a slightly better dynamical state within the architecture for this relatively short program, and that this is probably not a significant result.

In figure 5, the actual performance of out-of-order vs. in-order LSQ modes for the final processor is shown. Although, it was understood that there were a number of simplifications made in figure 4, the performance of the OoO LSQ in the final design was disappointing. We believe this was caused by the small data cache that was part of the final implementation. By issuing loads out-of-order and using a small data cache, we believe that some of the spatial and temporal locality benefits of the data cache might have been reduced. This theory is supported by the observation that the cycle time of sort.s, which repeatedly load and stores the same data in close program order proximity, with the OoO LSQ was made worse.

C. Branch Predictors

In the attached figure 6, the predictive capability of a variety of branch predictor configurations is shown. This chart is not a pure assessment of the branch predictors, as the measure of predictive worth is measured as 1-s/(u+c), where s is the number of squashes in a program run, u is the number of total committed conditional branches, and c is the total number of committed unconditional branches. Correctly predicting either a conditional or an unconditional branch required determining both its direction and destination correctly. As a result, jumps and returns, both hurt and help the assessment of the predictive power.

We were very satisfied with the performance of our gshare branch predictor in general. However, there were certain programs where our predictor did not perform as well as we would have liked. Of special note was the performance of the pipeline relative to Insertion.s. In the attached figures, we considered the pattern of instruction paths to see if we could explain this behavior. These figures show the program path behavior of a sampling of our test programs. For example, from figure 7, it can be observed that btest1.s, although always taken, never repeats a program instruction, and therefore the BTB never has destination PC stored for each new branch. In figure 8, copy.s, on the other hand, repeats the same behavior over and over again and so is easily predicted. A sample of the program path of objsort.s, (figure 9) which has many internal loops and utilizes returns, is noticeably less regular, although given sufficient warm-up time, a good predictor and a return address stack together should be able to perform well on this program. In fact, objsort.s was one of the best predicted programs for our predictor/RAS combination.

Based on the way we expected our predictor to work, and given that most of the branches in insertion.s, as seen in Figure 10, are almost always taken or almost always not taken, we expected our predictor to do considerably better than it did. So we closely examined our branch predictor's methodology. The insertion.s program has many branches flowing through the pipeline at one time. We realized that our choice to speculatively update the predictor was biasing its histories significantly with branches that were never subsequently committed. By experimenting with not speculatively updating the Branch Predictor, we got a predictive rate of, at best, 40% for Insertion.s. We realize now, that without implementing a scheme by which the state of the branch predictor is returned to its original state after a branch mispredicts, that the more branches in the pipeline, the farther from ideal behavior the predictor will provide.

D. Committing Two Things at a Time

In order to speed up our processor in the case that many instructions are waiting on one instruction to complete, we implemented the ability to commit up to two instructions at a time from the top of the ROB. This extra logic created quite a number of corner cases in the RAT in the allocating and deallocating of PRs. Does the capacity to commit up to two things at a time for a single issue pipeline have a performance benefit? If there is a benefit, you would expect it to scale with the latency of memory. The longer an instruction sits at the top of the ROB, the larger the backup of instructions done executing behind it, assuming they are independent of the top instruction.

In the attached figure 11, it can be seen that the performance benefit of committing two instructions at a time was at a maximum of 9%, for a single-issue pipeline. It is not surprising that there was no performance benefit for programs that do not load from memory (btest1, btest2, evens, evens_long). If we more carefully examine the case of copy.s, we can get some insight into why committing two at a time does not benefit us more. Figure 12 shows what fractions of a program's instructions were committed in two-at-a-time mode versus one-at-a-time mode. In figure 12, it can be observed that 83% of copy.s's instructions were committed in two at a time mode. Figure 13 shows what fraction of the time zero, one, or two instructions were committing during the operation of the program. A general conclusion is that most of the time, nothing is committing at all. This implies that there was time to commit one-at-a-time in between the bursts of two-at-a-time. And since only one instruction can be written into the ROB per cycle, removing one instruction per cycle is also sufficient to prevent stalls. The exception is if the burst of two-at-a-time occurs at the very end of the program execution, in which case, the program ends slightly sooner. However, the major benefit of being able to commit two at a time is resolving mispredicted branches faster by letting them get faster to the top of the ROB. Therefore, the benefits of committing two at a time would be nullified by early branch resolution. We conclude that for a single issue pipeline, committing two instructions at a time is worth the design complexity required.

It is also worth noting that until the branch predictor was predicting well, the maximum benefit was a 3% reduction in cycle time. This is one small example of the following more general conclusion. Attempting to analyze the performance of various modules of the pipeline in isolation is a low-value effort. For example, the performance of the load-store queue was impacted by the frequency of squashes, since when they were frequent, it never stalled the pipeline, and by interactions with the small data cache. The optimal sizing of various components is a direct function of the particular configuration of the pipeline. Thus we repeat our conclusion from the beginning of the report, that had we more time, we would have better optimized our pipeline. We believe that the pipeline we built has more performance potential than what it has in its current configuration.

E. References

1. Skadron, K., Ahuja, P. S., Martonosi, M., and Clark, D. W. 1998. Improving prediction for procedure returns with return-address-stack repair mechanisms. In Proceedings of the 31st Annual ACM/IEEE international Symposium on Microarchitecture (Dallas, Texas, United States). International Symposium on Microarchitecture. IEEE Computer Society Press, Los Alamitos, CA, 259-271.

2. Ghasemzadeh, H.; Mazrouee, S.; Kakoee, M.R., "Modified pseudo LRU replacement algorithm," Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on , vol., no.pp. 6 pp.-, 27-30 March 2006

3. Combining Branch Predictors, S. McFarling, WRL Technical Note TN-36, June 1993.

Work Break Down: Who Did What

Modules:

Prefetch (Dan)
Return Address Stack (Dan)
D-Cache (Joe, Andreas, Dan)
I-Cache (Nikhil)
Branch Predictor (Andreas)
Branch Target Buffer (Andreas)
Instruction Fetch (Nikhil)
Issue Queue (Nikhil, Dan)
Decode (Carolyn)
Rename (Carolyn)
Reservation Station (Andreas- Primary, Joe)
RAT/RRAT (Andreas- Primary, Joe)
ROB (ALL)
PRF (Dan)
Five Execution Units
An Adder Unit for Logical Operations (Joe)
An Adder Unit for Arithmetic Operations (Joe)
A 4-Stage pipelined Multiplication Unit (Joe)
A unit for Branch Logic and Addresses (Joe)
A unit for Memory Operation Addresses (Carolyn)
Load-Store Queue (Carolyn – Primary, Nikhil)
CDB Queue/CDB (Dan)
Commit (Carolyn)

Testing:

Issue Queue: Dan/Nikhil
Reservation Station/Rat Module Testing: Joe
Testing the front end (Icache - Decode): Nikhil
Testing the back end (Rename - Commit): Joe

Pipeline Integration: Joe, Andreas
Testing the Integrated Pipeline: Joe (50%), Carolyn (20%), Andreas (10%), Nikhil (10%), Dan (10%)
Synthesis - Joe

Report Writing and Analyses - Carolyn

It is the consensus of the group that work was evenly divided between all group members.