PROJECT 4

Architecture Review of the Nintendo® GameCube^{TM} CPU

Ross P. Davis
CDA5155
Sec. 7233
08/06/2003

Introduction

In this report, statements or sets of consecutive statements will be backed up by stating the reference from which I found them. This will be done by either explicitly stating the reference, or ending a sentence with "(reference #)" where "#" indicates the reference as it is numbered in the **References** section at the end of this report. Reference 3, the 750CXe user manual, is the primary reference for this report. If a statement does not have a reference, it can be assumed that the information was drawn from the user manual. In general, information taken from the user manual is marking with "[#]" where "#" indicates the page number in the user manual on which the information was found.

For Project #4 I have chosen the **Architecture Review** option in conjunction with the **Original Report Reviewing Literature Research** option. The Nintendo GameCube (GCN) uses a custom IBM PowerPC "Gekko" processor (reference 2). Unfortunately, details on the Gekko, like a user manual, are quite scarce and seem to be limited to overviews and press releases. Reference 1 provided the most detailed information, though the author admits that details on the Gekko are "sketchy". Many other technical articles I found on the Gekko often state verbatim the text in reference 1, as if all the authors copied it from the same source. Fortunately, all sources and press releases agree that the Gekko is based largely upon IBM's PowerPC 750CXe processor. The highly technical and detailed user manual for the 750CXe can be easily found on IBM's website (reference 3). Thus, the title of this report is something of a misnomer because I focus on the architecture of the 750CXe processor.

In the first section, I will provide a detailed discussion of the processor pipeline and functional units. The next section will contain information about how the 750CXe handles branching. The third section will give a brief overview of the processor's memory system and the final section will be composed of concluding remarks.

Processor Pipeline

The above is a picture of the Gekko processor as seen in reference 1. The Gekko runs at a modest (compared to today's processors) 485 Mhz and uses a RISC style instruction set. Reference 1 makes a point of mentioning that the RISC instruction set allows the Gekko to spend minimal time in the instruction decode phase, namely, one clock cycle. This is apparently not true of the Microsoft's Xbox which must spend several cycles decoding its more complicated instructions. The processor is equipped with thirty-two general purpose registers, and thirty-two 64-bit floating point registers. There are two 32-bit integer arithmetic-logic units (ALU) and a single 64-bit floating point ALU. According to the PowerPC user manual (reference 3), only one of the integer ALUs is capable of performing multiplies and divides, though I would not be surprised if the Gekko was made so that both ALUs could perform these operations. Each integer ALU is a 4-stage pipeline while the floating point ALU is seven stages (reference 1). In addition to these ALUs are a load-store unit and a branch prediction unit. It should be noted that information on how the Gekko differs from the 750CXe is scare. But reference 1 and reference 4 confidently state that about 40 instructions have been added. These instructions are special floating-point instructions designed to specifically aid in game performance.

The 750CXe pipeline is basically composed for four stages: fetch, decode/dispatch, execute, and complete. To draw parallels to what was taught in class, the fetch stage is obviously analogous to the instruction fetch (IF) stage. The decode/dispatch stage is a combination of instruction decode (ID) and issue (IS) stages. The execute stage (EX) is obvious, and the complete stage maps to the write back (WB) stage. Reference 3 contains an excellent diagram that I have duplicated below. This diagram shows the pipeline stages for the processor.

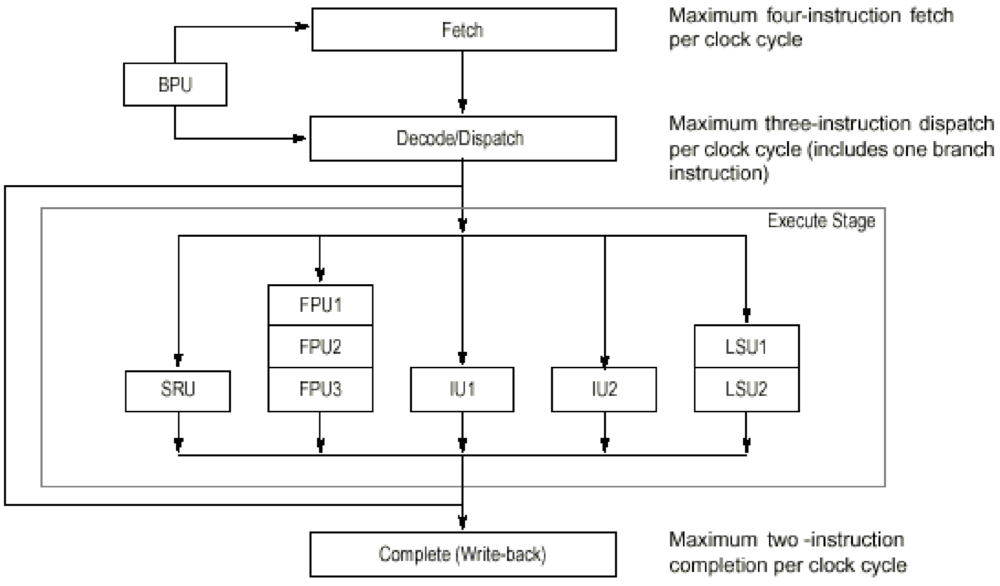

Fetch
Maximum four-instruction fetch per clock cycle
BPU
Decode/Dispatch
Maximum three-instruction dispatch per clock cycle (includes one branch instruction)
Execute Stage
FPU1
FPU2
FPU3
SRU
IU1
IU2
LSU1
LSU2
Complete (Write-back)
Maximum two -instruction completion per clock cycle

Figure 6-2. Superscalar/Pipeline Diagram

In the fetch stage (IF), an instruction is fetched from memory (the memory details will be discussed later) and placed in an instruction queue. This queue holds a total of six entries. From what I have learned in class I can speculate on the reason for this. There are bound to be pipeline stalls during the course of execution, but there is no reason that the IF stage should have to wait. By having six entries in the queue, the IF stage can continue to work even if another part of the pipeline has stalled. Additionally, this has a memory pre-fetch effect in terms of instructions. Instruction memory is basically requested before it is needed, giving the memory system more time respond.

As it turns out, the 750CXe is designed to keep its instruction queue full at all times. The queue can be populated with up to four instructions per fetch. The decode/dispatch (issue) stage happens instantly, meaning that it takes zero clock cycles. Three instructions may be issued at once assuming that one is a branch instruction. It should be noted that the three-instruction issue is somewhat misleading. In general, two instructions are issued from the front of the instruction queue to execution units during a clock cycle (assuming available resources). The possible third instruction is a branch instruction that can be thought of as coming from the rear of the queue. It issues out-of-order to the branch prediction unit (BPU) – branching will be discussed in the next section.

Moving on to the execute stage, I will discuss the integer ALUs (IU1 and IU2 in the diagram). Although reference 1 claims that the integer ALUs are pipelined, the user manual (reference 3) says that the ALUs are NOT pipelined. Either reference 1 is incorrect, or this is one of the differences between the 750CXe and the Gekko. But since the vast majority of integer instructions take only one clock cycle, it does not seem like there is much to gain by pipelining the integer ALUs. I believe that reference 1 means to say that the overall pipeline for an integer instruction is four stages (1 fetch stage, 1 issue stage, 1 INT stage, 1 complete stage). The instructions that take the most number of clock cycles at 19 are divides: divwu and divu. As previously mentioned, only IU1 is capable of performing multiplies and divides.

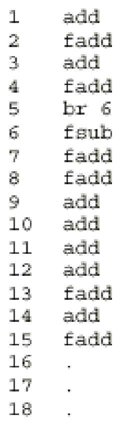
The floating point ALU is another unit in the execution stage. While reference 1 claims that the FPU has a seven-stage pipeline, reference 3 says that it is three stages. As with the integer pipeline, I believe reference 1 is referring to the overall pipeline, not just the FP ALU. If we look at the overall pipeline, we see that a floating point instruction does indeed have seven stages (1 fetch stage, 1 issue stage, 3 FP stages, 1 complete stage). It is interesting to note that several instructions (fdivs, fdiv, mtfsb0, mtfsb1, mtfsfi, mffs, and mtfsf) will completely block the FP pipeline. In other words, if one of these instructions enters the pipeline, no other FP instruction may enter the pipeline until it is clear. Most floating point instructions spend only three or four clock cycles in the pipeline. However, the fdivs instuction takes 17 clock cycles and fdiv requires 31 cycles.

There is a single, pipelined load-store unit (LSU) that has two stages. The first stage takes care of effective address calculation while the second stage read/writes to the cache. Obviously, cache misses and other related problems will stall the LSU pipeline [231].

The last unit in the execute stage was initially a mystery to me, but I believe I now understand what it does. The system register unit (SRU) appears to perform actions that modify the normal operation of the processor. For example, the whimsically-named eieio instruction (Enforce In-Order Execution of I/O) can be used to control the order in which loads and stores access memory [390]. To make an educated guess, I would say that these are instructions that a compiler would use to optimize execution.

Following the execute stage is the complete (write back) stage. Like the fetch stage, the complete stage also has a six-element queue associated with it. On any given clock cycle, a maximum of two instructions may be “retired” (to use the terminology in reference 1) from the front of the queue. This retirement takes place at the end of the complete cycle. Although execution can occur out-of-order, the completion unit will complete instructions in-order to guarantee a consistent program state. As an aside, the out-of-order execution is aided by reservation stations and register renaming, much like Tomasulo's algorithm. When an instruction is retired, the real register is updated with the rename register results. The description given in the user manual sounds very similar to Tomasulo's algorithm, though the name “Tomasulo” is not mentioned [223].

The user manual contains much more detail and information with regards to the pipeline and it would be impossible to describe it all in a 10-page paper. I will conclude this section on processor pipeline by going through a few examples given in the user manual. The following is the sequence of instructions that will be examined [217]:



```
1    add
2    fadd
3    add
4    fadd
5    br 6
6    fsub
7    fadd
8    fadd
9    add
10   add
11   add
12   add
13   fadd
14   add
15   fadd
16   .
17   .
18   .
```

The point of this simple example is to show the basic operation of the pipeline with integer and floating point adds. Additionally, there is a single branch instruction. In this example, it is assumed that there is always a cache hit. I disagree with user manual's numbering of instructions in this example. In the above instruction sequence, the instructions are 1-based, but in the diagram that will follow, they are 0-based. This can be a bit confusing when examining the branch instruction. To be clear, the “br 6” instruction will cause the processor to effectively skip the fsub instruction and continue execution with the seventh instruction. The following diagram shows the timing and pipeline stages for these instructions [218]. The manual also contains an excellent, cycle-by-cycle textual account to accompany the diagram [219-220]. I will paraphrase and summarize this account.

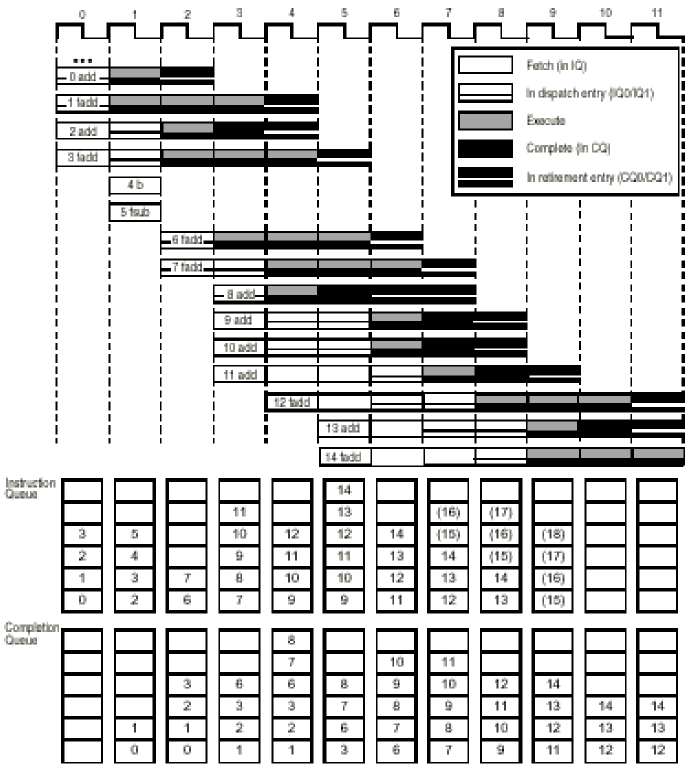


Figure 6-5. Instruction Timing—Cache Hit

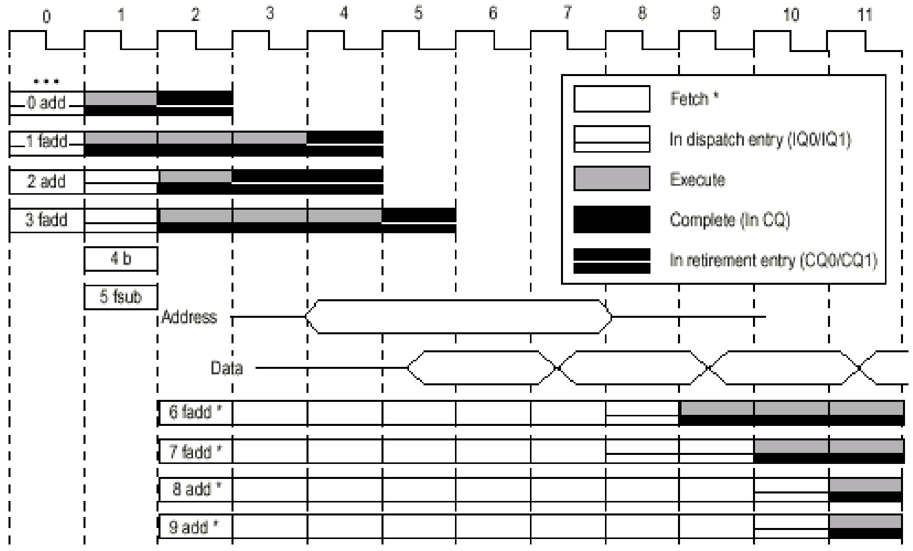
In clock cycle zero, we see that the first four instructions are fetched into the instruction queue. Since the first two adds are at the front of the queue, they are dispatched at the end of the clock cycle. In clock cycle one, we see that the first two instructions have begun executing and therefore get entries in the completion queue. While the maximum number of instructions that can be fetched is four, the processor will only fetch a number of instructions equal to the number of free queue slots in the previous cycle. Therefore, we see that two instructions are fetched (br and fsub). The second two instructions move forward in the queue.

In clock cycle two, the first add completes, the first fadd enters the second stage of the FPU pipeline, the second add begins execution, and the second fadd enters the first stage of the FPU pipeline. The processor sees that there is an unconditional branch at the head of the queue, clears the instruction queue, and uses its branch target instruction cache (BTIC) to fetch two new instructions starting at the branch target. Again, only two instructions are fetched because there were only two empty slots in the instruction queue on the previous cycle.

In clock cycle three, we see that the FPU pipeline is now completely full of instructions. The second add has completed execution, but it may not be retired until the preceding fadd is finished with execution. As previously mentioned, the 750CXe requires that instructions complete in-order, which is why the second add must wait to retire. Also of interest in this clock cycle is that the fourth fadd must stall in the dispatch stage because the FPU is full. On this cycle, the processor is able to fetch the full complement of four instructions.

In clock cycle four, the FPU is again full as one fadd completes and another enters the queue. Although there are two adds at the front of the instruction queue, they may not issue because the completion queue is full. In clock cycle five, the two adds at the head of the queue must wait again because there were no empty completion spots available in clock cycle four. The third add has completed, but it must wait until it move forward in the completion queue in order to retire.

The remaining cycles shed little additional light on the workings of the 750CXe and so they will not be discussed. Again, each clock cycle is explained in the user manual on pages 219-220. The last example I will show is interesting because it describes the timing that occurs on a cache miss. The diagram below shows this timing in the same format as the previous diagram [221]. The instruction sequence is the same as in the previous example, but in this case, the BTIC and both L1 and L2 caches miss when fetching instructions following the branch. The first four instructions proceed as before, but this time the branch target instructions must be fetched from main memory. In the diagram, we can see that the requested address is sent out on the bus in clock cycle two. Data comes back in 64-bit blocks (two instructions). The two target instructions make it to the processor at the beginning of clock cycle seven. In clock cycle eight, the instructions are placing in the instruction queue so that execution may being on clock cycle nine.



Branch Prediction

When the 750CXe fetches a branch instruction, it forwards it to the BPU instead of the instruction queue. In cases where it is known which way a branch will take, fetching will continue at the appropriate place. If the branch direction is not known, then the processor uses several techniques to make a prediction [214].

The first technique is not so much for branch prediction, but for speed. The branch target instruction cache (BTIC) is a 4-way associative cache with 64 entries. It is useful mainly for looping situations where a branch target will frequently reoccur. It simply maps a branch target address to sets of instructions, allowing the processor to fetch instructions quickly from the cache. The 750CXe offers another technique: dynamic branch prediction via a branch history table. This table contains 512 entries with two bits of prediction. The user manual describes the four possible states in a slightly different way than the course textbook: “not-taken, strongly not-taken, taken, and strongly taken”. The third technique is static branch prediction which will be discussed later [215].

The diagram below shows how the BTIC is used and what happens on a hit or miss in the BTIC when the branch is taken [225].

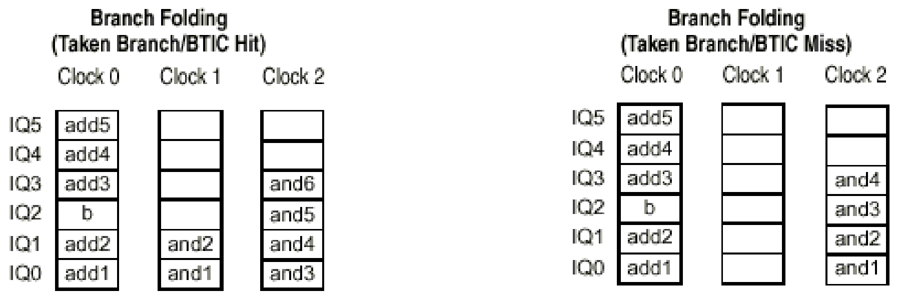


Figure 6-7. Branch Taken

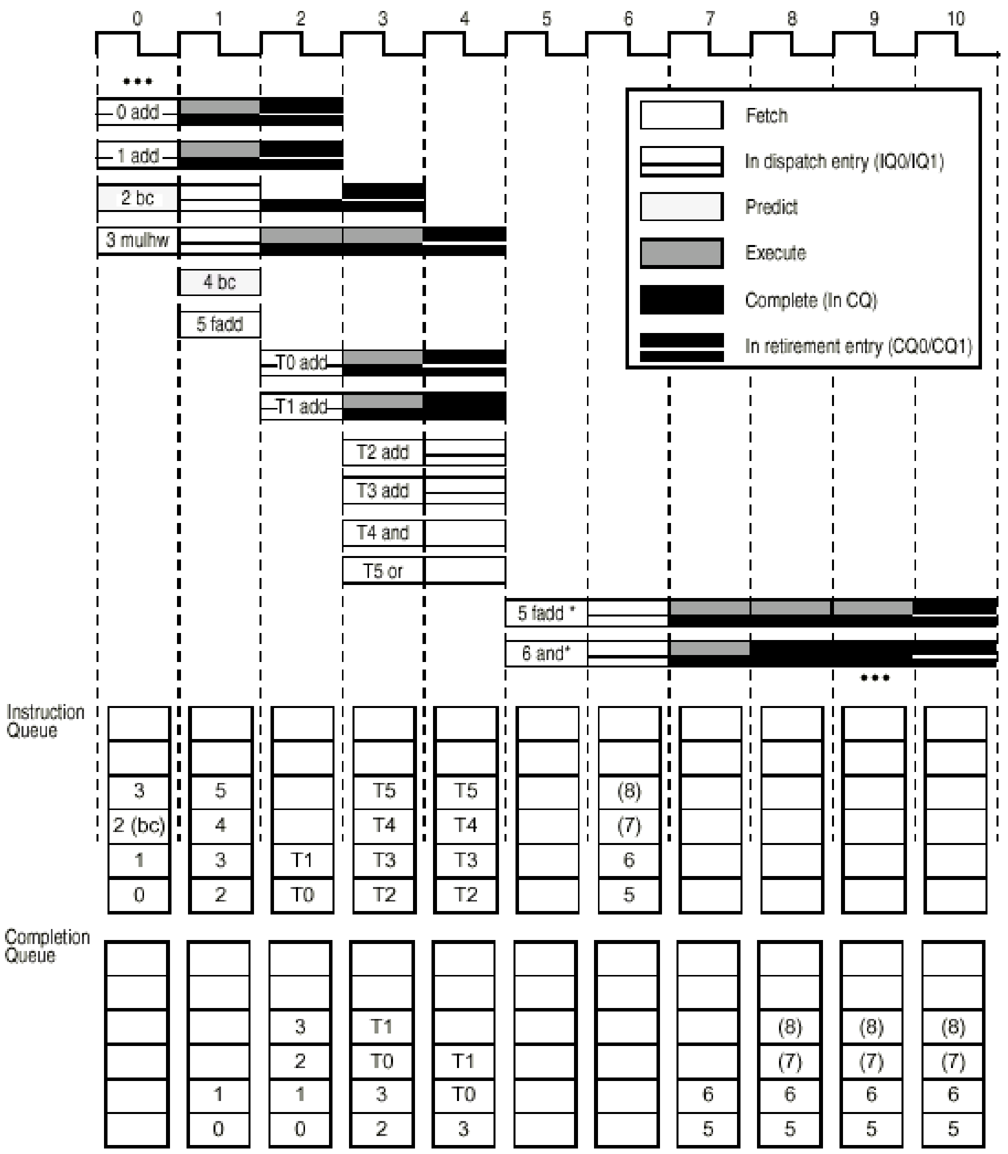
In the above, it is assumed that the branch target instruction is **add1**. So for the example of a BTIC hit, we see that the instruction queue is initially full and there is a branch instruction within it. On clock cycle 1, the first two adds from clock cycle 0 have issued. Since the branch was taken, the queue is flushed and populated with the first two instructions from the BTIC. These instructions issue on clock cycle 2 and the next four instructions are fetched. If the BTIC misses, then an attempt is made to fetch the instruction from the instruction cache. Since this requires another clock cycle, we see that clock cycle 1 is empty. On clock cycle 2, we see that the four branch target instructions have been fetched from the instruction cache.

The 750CXe can handle a maximum of two layers of prediction – meaning that it can predict one branch, and if another branch appears in the predicted instructions, it can also predict that one. Any instruction following a predicted branch may not be retired from the instruction queue until the prediction has been resolved. If a branch is mispredicted, then all instructions following the branch are flushed from the completion queue and the instruction queue. The correct program path then begins [227]. Thus misprediction is obviously an expensive occurrence.

As previously mentioned, the 750CXe supports static branch prediction. Basically, all branch instructions contain a field that a compiler can use to specify whether the branch should be predicted as taken or not. Static branch prediction is not the default behavior and so a special register (HID0[BHT]) must be cleared to indicate that the branch history table (and thus dynamic branch prediction) will not be used [228]. I find this to be a very powerful and elegant way of handling branch prediction. It allows for the use of compiler technology and program-specific optimization, but will default to use a reliable hardware-based prediction method.

I will conclude this section with a branch prediction timing example as given in the user manual. The example will use the instructions listed below [228] and the timing diagram follow [229]:

```
0    add
1    add
2    bc
3    mulhw
4    bc T0
5    fadd
6    and
     add
T7   add
T8   add
T9   add
T10  add
T11  or
```



Before commenting on the diagram, I should note that the instruction listing is somewhat incorrect and confusing. Following instruction 6 is an add that has no label. From the diagram, it appears that the instructions T7-T11 should be shifted upward one instruction and the final or instruction should be labeled T12. Moreover, the diagram refers to the branch targets starting with T0 to indicate the first target. But the instruction listing appears to label the first target as T7.

Clock cycle zero shows an example of a three-instruction issue. The first two adds are issued to IU1 and IU2 and the branch instruction is issued to the BPU where it is predicted that the branch will be not taken. In clock cycle one, the first branch and the mulhw instruction begin execution. Since the first branch updates one of the system registers, it must continue through execution. The second branch enters the instruction queue and it is predicted as taken. It does not modify a system register and so it may be eliminated (or "folded") from execution. It is given that this branch depends on mulhw and so it cannot be resolved until the instruction completes.

In cycle two, the second branch has been removed (folded) and the fsub instruction has been flushed. The first two branch target instructions have been placed in the queue after being grabbed from the BTIC. In cycle three, execution continues as normal along the predicted instruction path.

Cycle four is where things get interesting. The outcome of the second branch depended upon the mulhw instruction. This instruction completes in cycle four, proving the branch prediction wrong. The instruction and completion queues are flushed, and the correct instructions are determined during cycle five. The instructions are placed in the instruction queue on clock cycle six, and their execution begins on cycle seven.

Memory

Reference 2 tells us that the Gekko processor has both a data and instruction cache at the L1 level. Both of these caches are 32KB and 8-way associative. It also has an L2 cache that is 256KB and is 4-way associative. Main memory is 24MB with a latency of less than 10ns at all times. The disc can hold 1.5GB and has an average access time of 128 ns.

The 750CXe contains a memory management unit (MMU) that is used primarily to translate effective addresses into physical addresses. In general, loads and stores are the only instructions that require these address translations. The processor maintains two translation look-aside buffers (TLBs), one for instructions and one for data. In fact, the 750CXe can be thought of as having two MMUs since instruction memory and data memory is handled separately [175].

The processor has an effective address space of 4GB with a page size of 4KB and a segment size of 256MB. The size of a block varies from 128KB to 256MB depending on software programming. The translation look-aside buffers are two-way set associative and contain 128 entries. The MMU also supports a UNIX-like permissions feature, giving the ability to declare parts of memory as “no execute” or “read only” [177]. The user manual provides an in-depth description of the 750CXe memory system which can be seen beginning on page 175.

Conclusion

In conclusion, I was pleased to find that I understood much of what I read in the 750CXe user manual. Before taking this course, much of the manual would have been outside my realm of knowledge. Processor pipelining, register renaming, branch prediction, multiple issue, TLBs, and memory caches are all things learned in CDA5155, and all things that are described by the user manual. I found that things are a bit more complicated than in book, since this is a real life example and not a hypothetical textbook scenario.

There are several things that I would have liked to have researched but did not have the time and resources. First, it would have been nice to have the user manual for the Gekko processor so that I could have analyzed that instead of the processor it is based upon. Additionally, if I had more room and time, I would have liked to examine the “Flipper”, the GameCube's graphics chip. The GameCube is capable of performing 10.5 GFLOPS and producing 6 to 12 million polygons per second (reference 2). The whole system is built to produce beautiful graphics as fast as possible and it would be interesting to examine how the hardware is built to accomplish this. I would have also liked to have been able to know the definite differences between the Gekko and the 750CXe. Aside from the addition of more instructions, I was unable to find what else was changed between the two processors.

As a fan of both computers and the Nintendo GameCube, I enjoyed gaining insight into the inner workings of my favorite gaming console. It is very rewarding to be able to apply what I have learned in class to a real-world system. In the future, I hope to follow the evolution of microprocessor design so that I may continue to understand this highly important aspect of modern technology.

References

1. http://www.anandtech.com/printarticle.html?i=1566

2. http://www.nintendo.com.au/gamecube/system/index.php

3. IBM PowerPCâ„¢ 750CX/750CXe RISC Microprocessor User's Manual Version 1.1 (http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF7785256996005B8B61)

4. http://www.2002.arspentia.org/rla/gamecube_review