A screenshot of a document

Description automatically generated

**Abstract**

The goal of this design project is to design and create an SDRAM controller as well as a cache memory system, using a cache controller and an SRAM cache. The CPU is to issue 16-bit address words (may be less if not possible), to the cache controller, and it is to act accordingly. The cache controller must be capable of performing 4 cases:

1. Write a word to the cache (hit)
2. Read a word from the cache (hit)
3. Read or write from or to the cache (miss) with the dirty bit set to 0
4. Read or write from or to the cache (miss) with the dirty bit set as 1

The proper functionality can be tested with the simulated waveforms in order to ensure the project is functional.

**Introduction**

In nearly all modern computers, there are multiple different memories, all of which have a purpose of storing, reading, and processing data and information, however, each level has a different purpose or usage within the system. As an example, RAM provides very fast read/write but this memory is lost or reset when the system is turned off. ROM is also a form of memory; however, it is non-volatile and stores read-only data which is permanently stored. Withing RAM, there are many categories such as SRAM (static RAM), DRAM (dynamic RAM), SDRAM (synchronized DRAM), and more. These memories, as well as others such as main memory, secondary storage are placed into a memory hierarchy. As a general rule, the closer the memory is to the CPU, the faster the access speed, but smaller the size of available storage, and more expensive it is.

**System Specifications**

There are 4 behavioral cases we must implemented in the cache controller, these being:

1. When the cache controller receives a request to write from the CPU, a cache hit is detected. In this situation, the index and offset data provided by the CPU is then moved over to the SRAM as the address where the data will be written. Next, the multiplexer writes data to the SRAM received from the CPU only if/when the write bit is enabled, otherwise it is not able to do so.
2. The second case is the cache receiving a read request from the CPU. Here, a cache hit should be happening. The index and offset are transmitted to SRAM and the data read from that address is then sent back to the CPU.
3. The third case occurs when a request is received from the CPU, but the requested block is not found in the cache, as well as the dirty bit being set to 0. The block must be fetched from memory and is placed in the cache. The valid bit is then set to 1 as the tag register is replaced. Then, the request can operate without issues.
4. The final case occurs when a situation similar to the previous case occurs, but the dirty bit is set to 1 instead. In this case, the appropriate block is required to be written to the SDRAM before the system continues reading from the main memory.

**Device Design**

1. **Symbols**

**CPU:**

**A diagram of a computer chip

Description automatically generatedA computer circuit diagram with black text

Description automatically generated**

**Figure 1: CPU Symbol**

**Cache Controller:**

**A diagram of a computer component

Description automatically generatedA white sheet with black text

Description automatically generated**

**Figure 2: Cache controller symbol**

**SDRAM Controller:**

**A diagram of a computer component

Description automatically generatedA white paper with black text

Description automatically generated**

**Figure 3: SDRAM controller symbol**

**Cache SRAM:**

**A diagram of a computer component

Description automatically generatedA screenshot of a computer

Description automatically generated**

**Figure 4: Cache SRAM symbol**

**2-to-1 Multiplexer:**

**A black and white image of a diagram

Description automatically generatedA close-up of a white card

Description automatically generated**

**Figure 5: 2-to-1 multiplexer symbol**

**1-to-2 Demultiplexer:**

**A black and white image of a white object with arrows

Description automatically generated with medium confidenceA close-up of a sign

Description automatically generated**

**Figure 6: 1-to-2 demultiplexer symbol**

The **figures** above **(1-6)** illustrate the individual components used in the block diagram used to create the entire cache controller. Each symbol was created by stating components in each file. After each component was verified, the block was created using “create schematic symbol”. After this part, we opened a schematic file to link each symbol together with wiring between the appropriate inputs and outputs. This can be seen in the block diagram **Figure 7 and 8**.

1. **Block diagrams**

**A diagram of a computer component

Description automatically generated**

**Figure 7: Complete cache system**

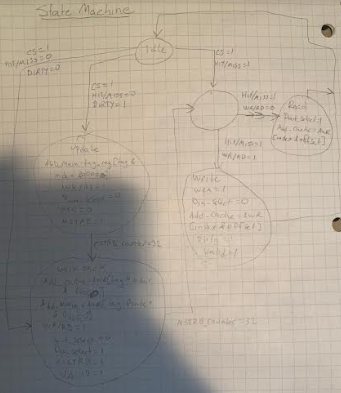
**A diagram of a computer hardware

Description automatically generated**

**Figure 8: Block Diagram**

**Figure 8** shows the full block diagram of connections for the cache controller with each component wired, as per the manual instructions shown in **Figure 7.** The thin wires have only 1 bit connection, while thicker wires represent multiple bits to be transmitted. The main inputs are clock and reset. Clock signal is used to synchronize the CPU, cache controller, SDRAM and SRAM. The CPU transmits an address to cache controller. The cache controller then must read or write to SRAM, which depends on statuses in cache controller. Along with this process, the CPU will also transmit an address to the SRAM using the 2-to-1 multiplexer with the selector bit coming from the cache controller itself. The multiplexer and demultiplexer are used to transmit data to CPU, data from the SRAM and comparing values in order to determine whether a cache hit or miss has occurred. The SRAM which is connected to the (de)multiplexers is the cache’s memory component which stores data. This component receives signals from all other components. The SDRAM’s job is to evaluate data transmitted by the cache controller and also taking input from the clock and demultiplexer.

1. **State diagrams**



**Figure 9: State diagram**

The figure above indicated a finite state machine (FSM) used by our cache controller. First, the controller must fetch an address from the CPU. The CPU then issues the read/write enable bits as well as the control signals required such as addresses to be fetched by the cache controller. Then, the cache controller will search the SRAM and see if there is data in that specified location (hit) or if there is not any (miss). If a hit is detected by the cache controller, it will then begin to read/write the data from SRAM and send that information to the CPU. Afterwards, the signal will be switched to indicate that the cache controller is in an idle state, and it is awaiting further instructions from the CPU. Once it receives instructions, it will return to the first state of checking whether this value is in the cache or not. If a miss is detected by the cache controller, the cache controller moves to a state that reflects this, depending on the value of the dirty bit. If the dirty bit is 1, it will enter state where it writes the block to SDRAM, then the controller will switch states to read/write to/from SRAM. However, if the miss was detected, and dirty bit is 0, the cache controller will skip the write operation and will read directly to SDRAM. Finally, it will switch states after retrieving data from SDRAM in order to store it in SRAM cache memory. It will then wait for another instruction from the CPU.

1. **Process diagrams**

**A diagram of a computer system

Description automatically generated**

**Figure 10: Process diagram**

**Figure 10** above illustrates the process diagram of our cache controller. The process starts when the CPU sends instructions as well as an address to the cache controller. Next, the cache controller will check the SRAM memory to determine if there is data in that location of the cache (hit or miss). If a hit is detected, the controller is able to read/write to/from the SRAM. But, if it a miss, the controller must read the dirty bit first, in order to determine its next step. If the dirty bit is a 0, the SDRAM will write its information at that address to the SRAM and the cache controller is able to continue with the original request. The controller then lets the CPU know that it is awaiting further instructions. However, if the dirty bit is a 1, then the cache controller will copy the block back to the SDRAM before the new information is given to SRAM.

**Results**

A black and green screen with white text

Description automatically generated

**Figure 11: Functional simulation results**

**Figure 11** is a demonstration of the simulation depicted. At the 630 ns mark, we see the state switch, where instructions are fetched from CPU. The chip select signal is at 0 and the read/write signal is high, meaning a write operation is to be executed. Hit is a 1, meaning that a hit has been detected and a state change is done. Because of this, we see at 640 ns that Dout\_SRAM is made to be same as Din\_CPU, which works as required as the read operation to CPU was done properly. The controller goes back to next state and sits idly waiting for next instructions form CPU. It then repeats a similar process.

**Conclusion**

It appears that the cache controller designed was implemented as should be. Thanks to this project, greater knowledge, and insight into the workings of a cache controller and the cache system was developed. The importance was also showcased greatly. Why dirty bits and valid bits are used and why they increase optimization of data flow and control was also showcased during this project. This project greatly showed how memory controllers can affect time and efficiency of programs and information to be shared with other components.

**Appendix/References**

**References:**

<https://courses.torontomu.ca/d2l/le/content/792271/viewContent/5322259/View>

<https://courses.torontomu.ca/d2l/le/content/792271/viewContent/5323145/View>

<https://courses.torontomu.ca/d2l/le/content/792271/viewContent/5322257/View>

<https://courses.torontomu.ca/d2l/le/content/792271/viewContent/5322256/View>

<https://courses.torontomu.ca/d2l/le/content/792271/viewContent/5325205/View>

**Sources:**

A screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

A screenshot of a computer code

Description automatically generated

A screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generatedA black text on a white background

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screen shot of a computer code

Description automatically generated

A computer code with red and blue text

Description automatically generated

A computer code on a white background

Description automatically generated

A white background with black text

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer program

Description automatically generated

A computer screen shot of a program

Description automatically generated