**TERM PROJECT, CODE – REPORT**

**Introduction:**

Based on summary 2, we know that Many machines that run on integrated CPU-GPU systems use *Cache-Coherent Shared Memory* (CCSM) to enable communication between the CPU and GPU. This communication is in a way that the CPU acts as a *producer* and the GPU acts as a *consumer*. The purpose of this code is to illustrate how a single producer and single consumer operation is using a shared buffer through mutual exclusion.

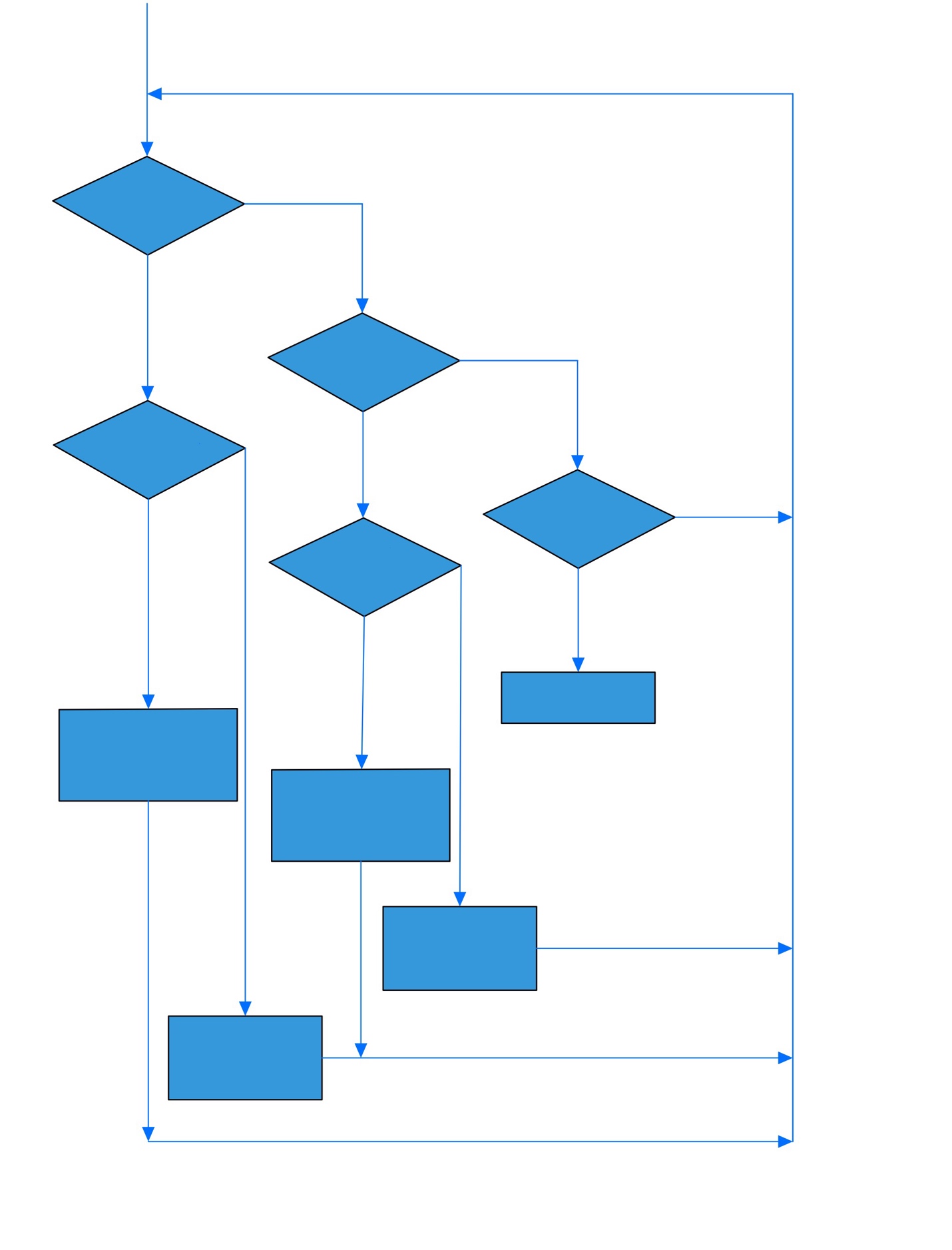
In this basic level shared memory cache coherence problem is avoided by having mutual exclusion between the consumer and producer. Only one process can run at a time while accessing the shared memory, the second process starts only when the first process is finished running. This way update on shared variables is done only by 1 process at a time and then making it available to other processes for later use. Without mutual exclusion, multiple processes would update the shared variable and make it incorrect which is similar to the cache inconsistency in CPU-GPU systems.

**Approach:**

In the one producer and one consumer situation, the two processes share a common fixed buffer. We assume a fixed buffer length of 3 items for this program. The buffer being fixed, the producer cannot add items when the buffer is full, at the same time the consumer cannot consume items when the buffer is empty. Therefore, we declare two variables for handling the buffer, *full* and *empty*.

The variable *full*isinitialized to 0 (no items in buffer) and gets incremented when the producer produces items or decremented when the consumer consumes the item. The variable *empty* is initialized to 3 (buffer is empty) and gets decremented when the producer produces an item or incremented when the consumer consumes an item. Also, we need a variable to check for mutual exclusion, *mutex*is initialized to 1 and gets decremented to 0 by either process while running and gets incremented to 1 at the end of each process so the next process can enter (note that *mutex* must be 1 for any process to start running). We also define a variable *item*to keep track of the total number of items in the buffer. Figure 1 shows the overall flow of this program.

The whole process is run in an infinite loop until the user chooses to terminate. The program prompts the user with 3 options. Option 1 to enter producer process, option 2 to enter consumer process and option 3 to terminate the program. In option 1, if the buffer is not full and the mutex is 1, the producer process produces an item or else print that the buffer is full (producer cannot produce more items). In option 2, if the buffer is not empty and the mutex is 1, the consumer process consumes an item or else prints that the buffer is empty (there are no items to consume from the buffer).



False

True

False

True

False

True

False

True

True

False

User Input

If option = 3

If option = 2

Print Buffer Full

full != 0

If mutex = 1 &

empty != 0

If mutex = 1 &

Print Buffer Empty

empty--

item++

mutex--

mutex++

full++

Exit

If option = 1

full--

Item--

empty++

mutex++

mutex--

Figure 1 Program Flow

**Solution:**

The above-mentioned approach has been successfully coded and executed in MIPS assembly language. Figure 2 shows the successful completion of the assembly operation.

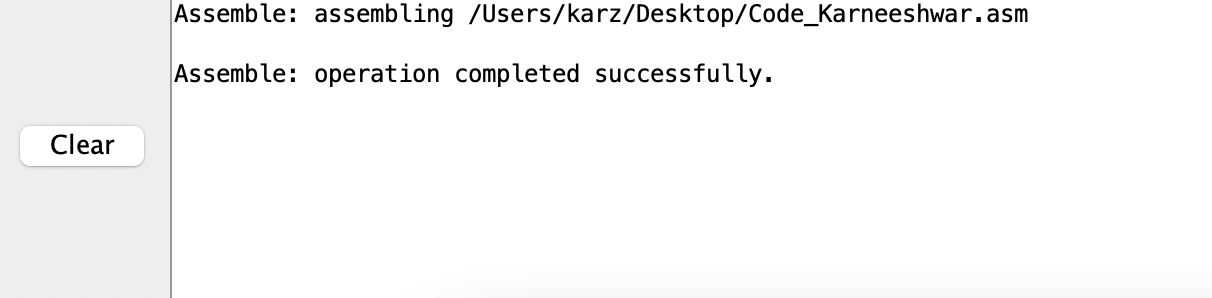


Figure 2 Assembly Completion

Figure 3 shows the results for all the cases. First consume is called on an empty buffer, second 3 items are processed by calling producer and it shows that the buffer is full when called 4th time, next 3 items are consumed by calling consumer and it shows that the buffer is empty when called 4th time, and at last the process is terminated.

Text

Description automatically generated

Figure 3 Results

**Conclusion:**

Therefore, a simple illustration of one producer and one consumer process with a fixed common buffer has been used to show how the cache inconsistency is avoided with shared variables using mutual exclusion. In integrated CPU-GPU systems, there will be multiple processes that would run in parallel and complex manner occupying the cache, the higher the complexity of the processes, the higher the complexity of cache coherence protocols used.