**Multicore Processor Project – Computer Architecture Course**

Ofir Guthman & Yonatan Gartenberg & Rony Kositsky

Tel Aviv University

We will go over all the files and code in our project. We will go in top-down approach – we will explain the general structure of the project and deepen its implementation.

Main

Text

Description automatically generated

The main is simple – we initialize all the files, memory structure and 4 cores structure (we will be explained more deeply in the next pages).  
The main loop iterate over all the 4 cores and doing one bus iteration. It will continue until all the 4 pipelines are flushed and then continue to tear downs, prints and closing files.

**Helpers**

Some definitions we will use in all the project.

Text

Description automatically generated

**Opcode**

**Text

Description automatically generated**

**Text

Description automatically generated**

**Text

Description automatically generated**

Basic definitions of the opcodes. We define and handle almost all the opcode operations in this module (except branch and memory).

OpcodeMapping – functions array.

Opcode\_fucntion\_params\_s – struct which we will pass with all the relevant data for the calculations.

**Files**

Text

Description automatically generated

This module handles the opening, closing and assigning the files. Each core will hold its relevant file.

**Memory**

Graphical user interface

Description automatically generated with low confidence

The memory is a simple module. We hold the memory size value, the initialize function and the final print.

Text

Description automatically generated

In the implementation of the module, we will define the memory address struct and the memory array.   
counter - we bill used for counting the clock cycles in IO transaction.  
gMemoryTransaction – global flag to signal that the memory in transaction.

bus\_transaction\_handler – handle transaction from the bus.

get\_memory\_length– getting the memory length.

**Core**

Text

Description automatically generated

A screenshot of a computer

Description automatically generated

This is the core module. We are holding 4 modules which run in parallel.   
Each core holds its instructions, the pc, the register and the pipeline module which will be executing the instructions.

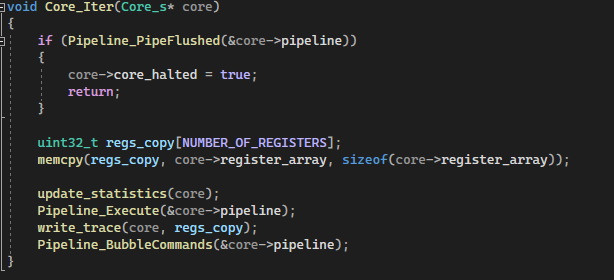
Core\_Init – initialize the core.  
Core\_Iter – one core iteration, happens each cycle.  
Core\_Teaddown – teardown of the core.  
Core\_Halted – return true if the pipeline is flushed.

This is the initialization function:

Text

Description automatically generated

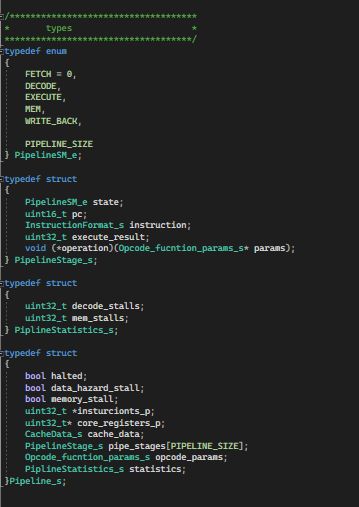
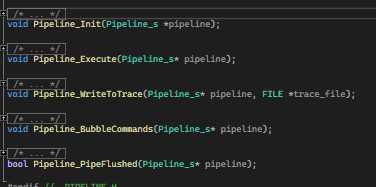
And this is the iteration function:



This is the flow:

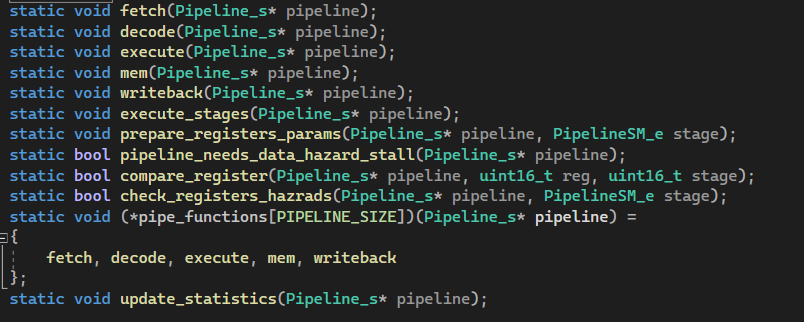
* Checking if the pipeline is flushed.
* Creating register copies (used for the prints).
* Updating the statistics.
* Executing one pipeline cycle.
* Writing for the trace.
* Bubbling the command down the pipeline.

**Pipeline**

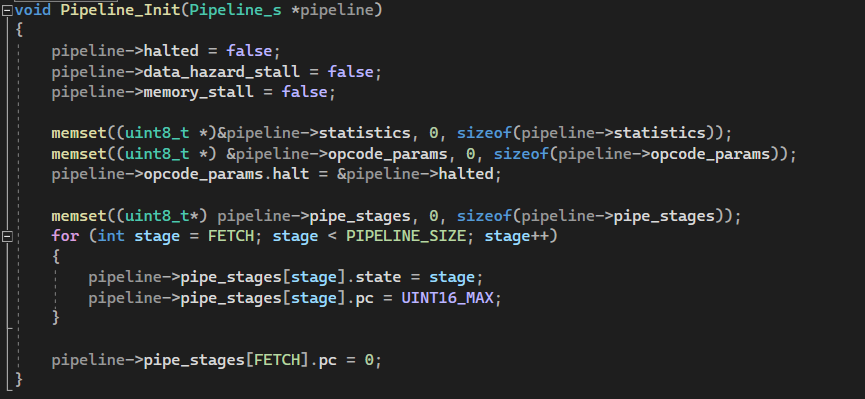
This is the module which control the pipeline and handle the command executation.

PipelineStage\_s – the pipeline stage struct. Contains relevant data for each stage, each pipeline has 5 stages.  
Pipeline\_s – the pipeline. Holds the logic flags, pointer to the core instructions memory, the core register, the cache and 5 stages.  
Pipeline\_Init – initialize the pipeline.  
Pipeline\_Execute – execute one pipeline iteration.   
Pipeline\_WriteToTrace – writing relevant data to the trace.  
Pipeline\_BubbleCommands – bubble commands down the pipeline.  
Pipeline\_PipeFlushed – returns true if the pipeline is flushed.

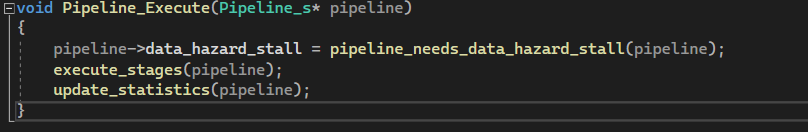
Now, we will show the pipeline implementation.   


The initialization function:

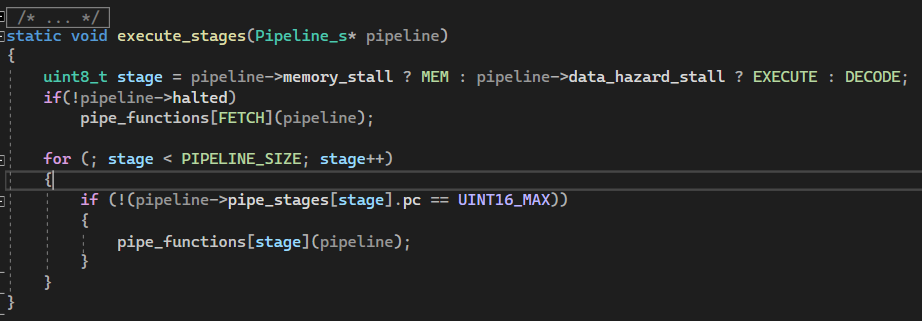
Each stage initialize to UINT16\_MAX which we defined as uninitialized.



The execute funciton:



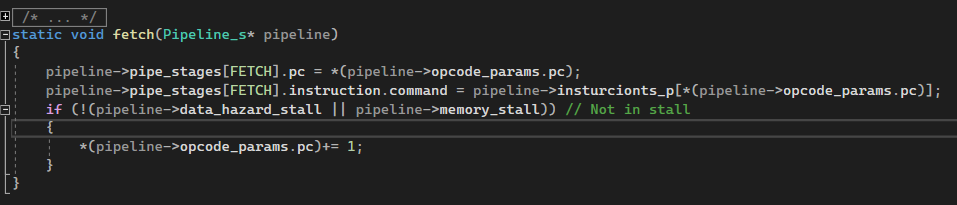
It manages the data hazard flag, and continuing for the execution:

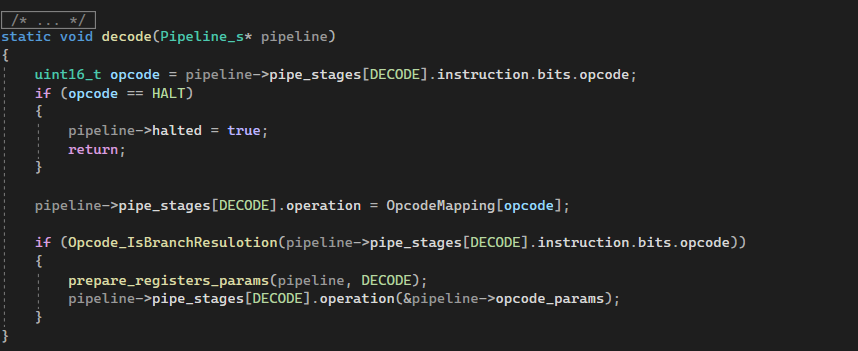


We go over the stages by this definition:

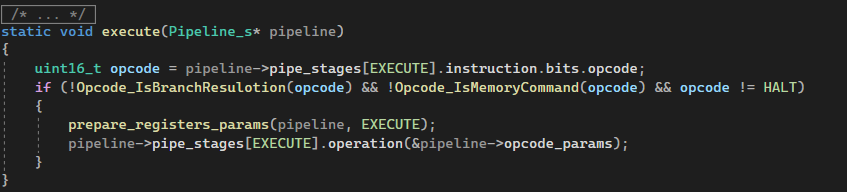
* We always Fetch (If we in stall, the pc won’t progress so nothing will happen).
* If we in data hazard, we will start the loop in Execute stage.
* If we in memory stall, we will start the loop in Memory stage.
* Else, for each initialized stage we will execute it.

This are the stages:

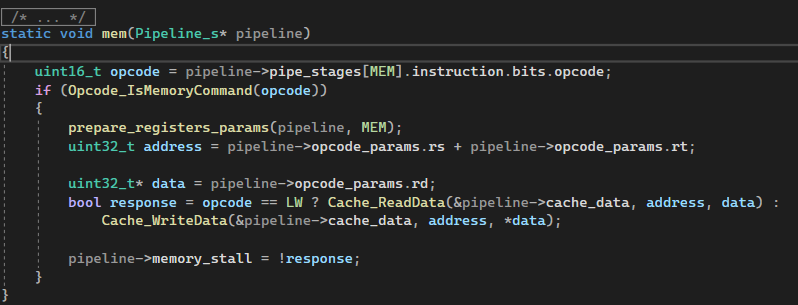




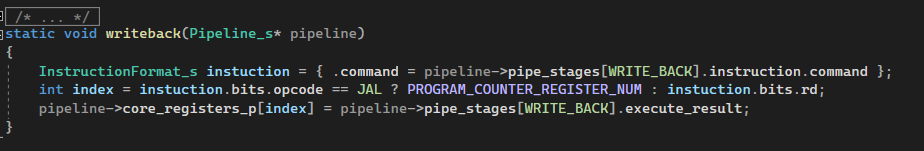
If this is a branch operation we are doing branch resulotion.



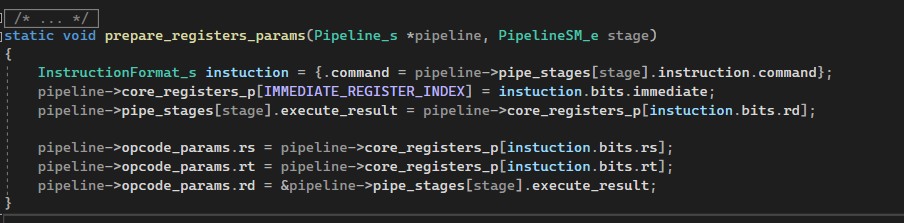
Executing the command we received.



Handling memory command. If we are in stall because IO operation we will configurate it with the response variable. The memory IO and bus operations will be explained later.

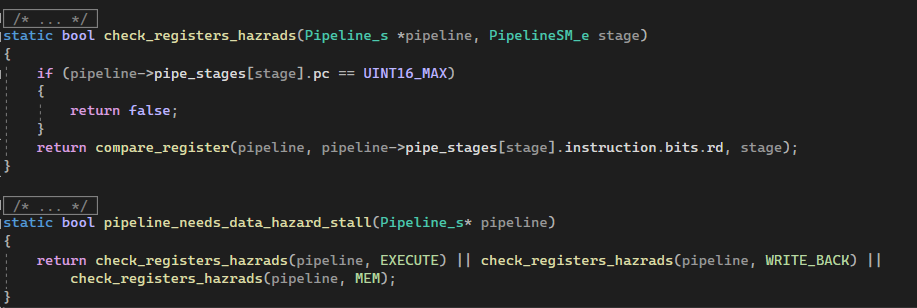


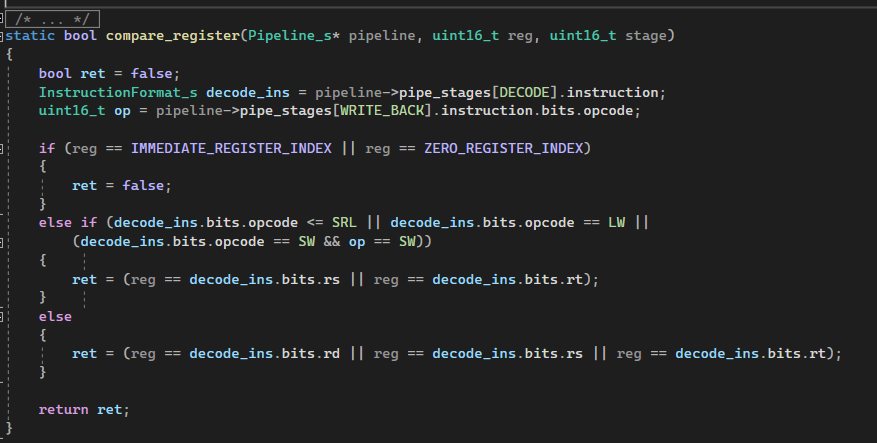
Writing the values of the register.



This function prepares the struct which we will pass to the execution function with all the relevant data.

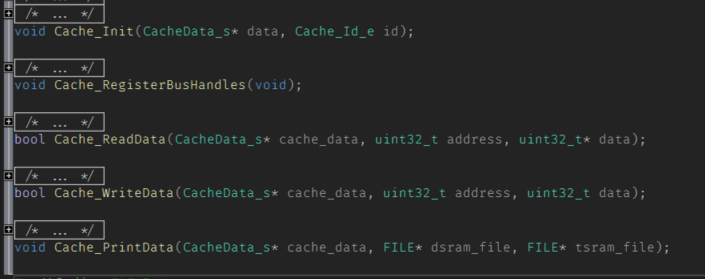
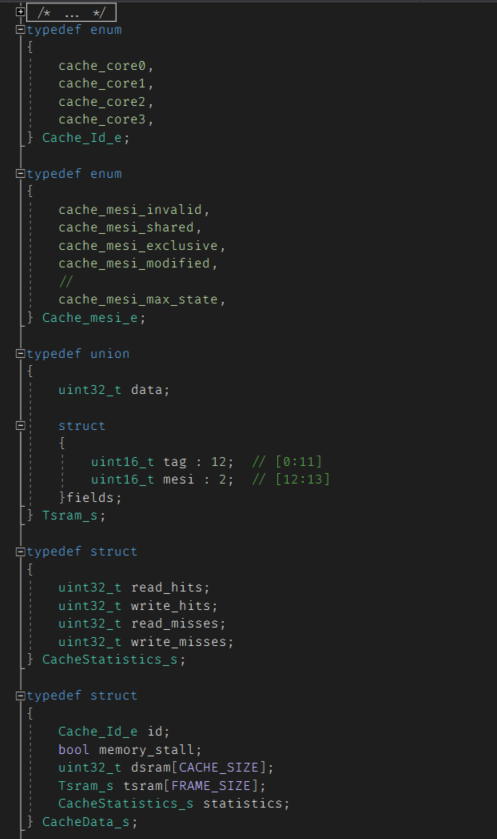
This is our data hazard resulotion:





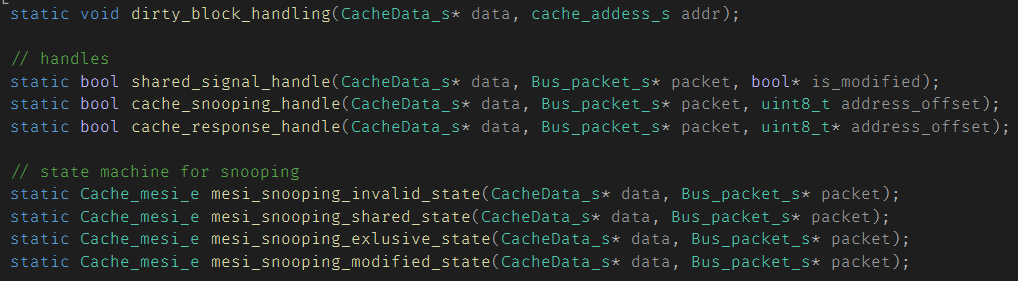
We actually looking if a data hazard is occuring, and raising the flag if we find one.

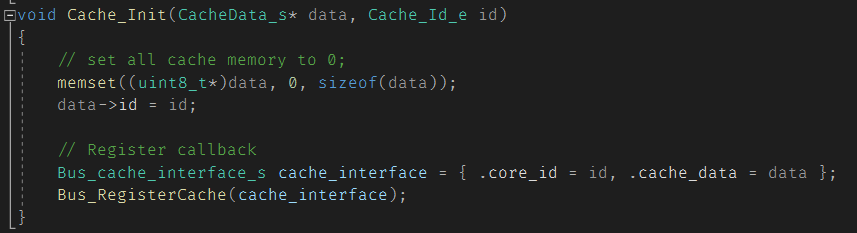
**Cache**

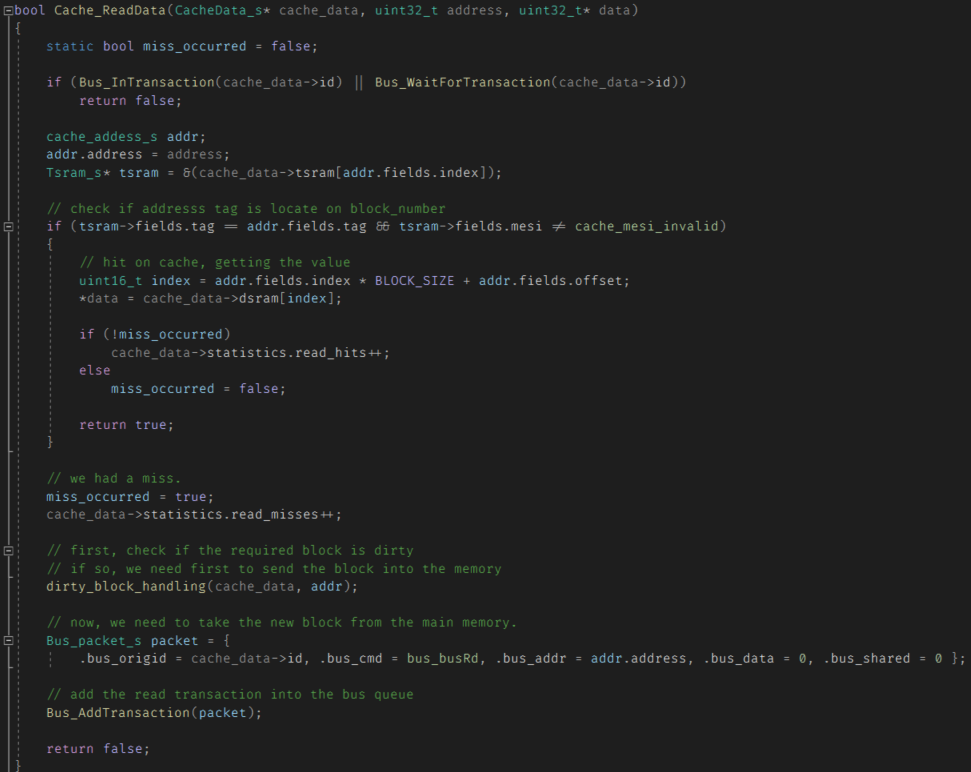


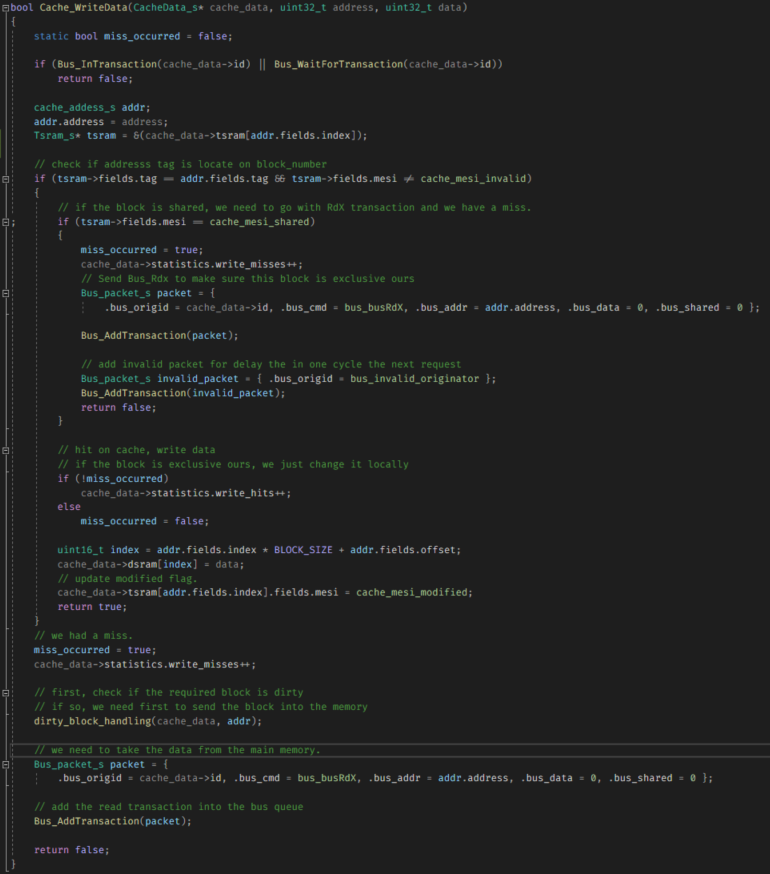
This is the module is control the caching and handle the transaction through the bus.

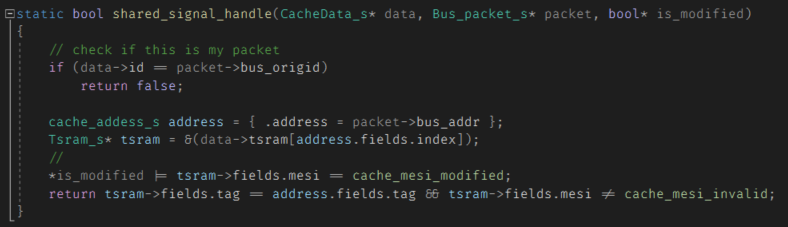
CacheData\_s – the cache struct. Contains relevant data for caching, each pipeline has this struct.  
Cache\_Init – initialize the cache.  
Cache\_RegisterBusHandles - register cache callback handles to the bus.  
Cache\_ReadData – execute reading from the cache.   
Cache\_WriteData – execute writing to the cache.  
Cache\_PrintData – print cache memory data (dsram, tsram) to file.

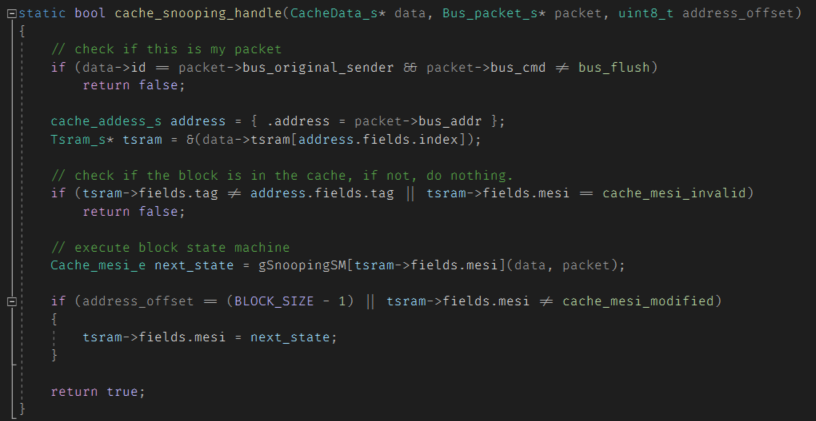
Now, we will show the cache implementation.   


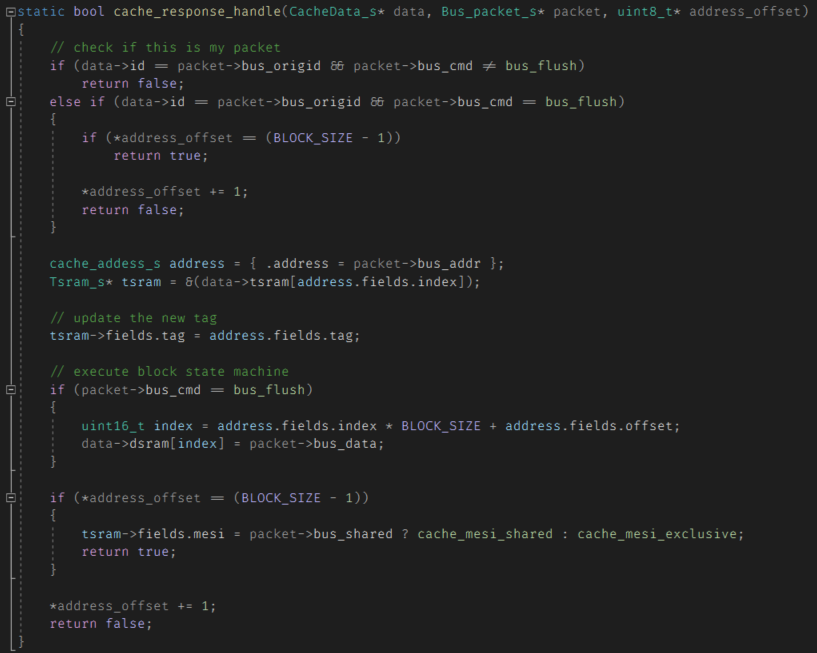
The initialization function:  
Initialize all cache data to 0 and register the cache callback from the bus.   


The reading funciton:  
Getting the values from the cache when hit, otherwise sending a transaction to the bus for new data  


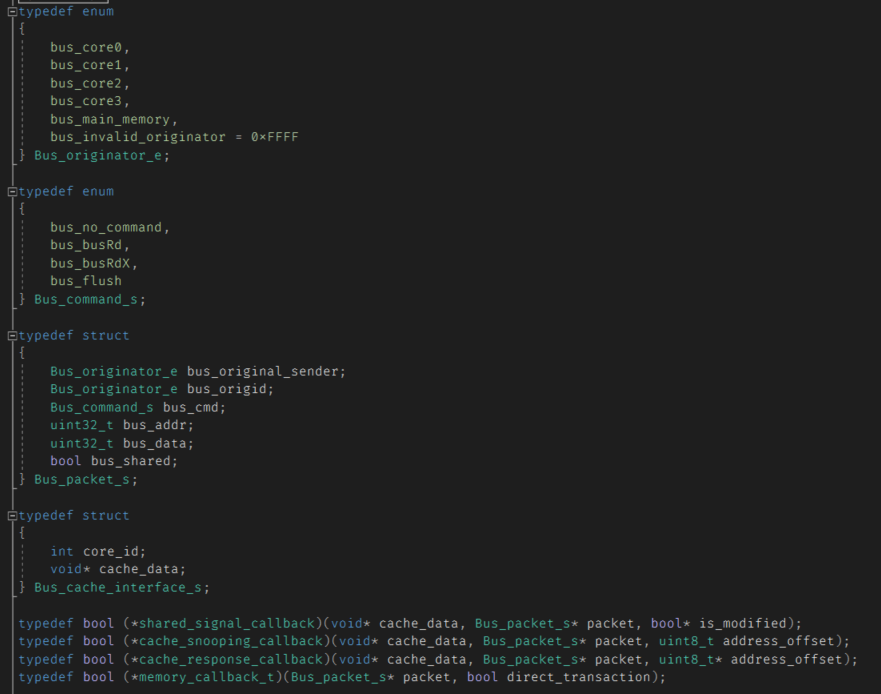
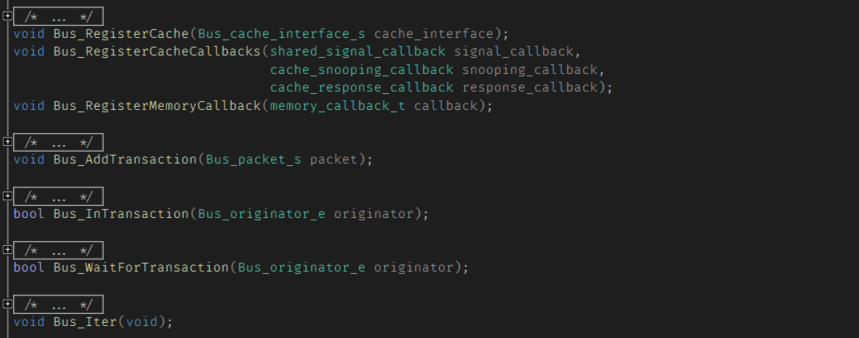
The writing function:  
Like in the reading function handling here the miss and the hit to the cache.

The shared signal callback handle:  
Notify other caches if the block exist in the current cache.  


The snooping callback handle:  
Execute the snooping block state machine like learned on class.   


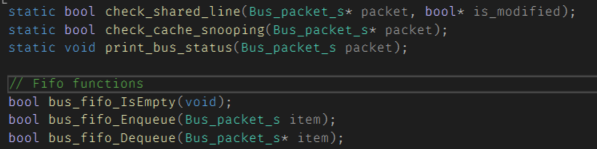
The response callback handle:  
Handling the response from the bus at the end of transacion execution.  


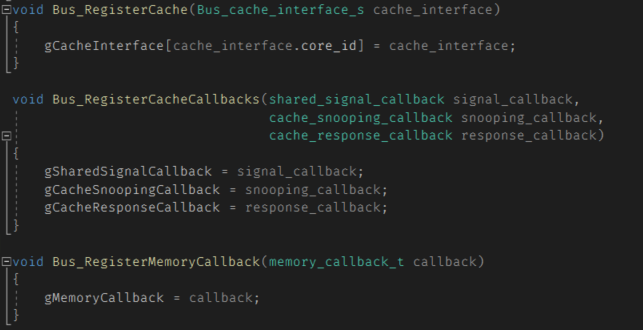
**Bus**

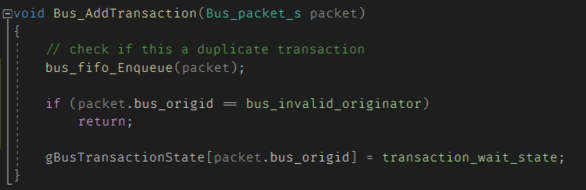
 

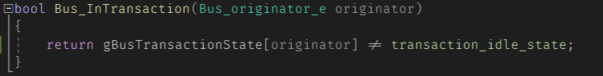
This is the module handle the bus and manage the transaction.

Bus\_packet\_s – the packet struct. Contains all bus transaction data.  
Bus\_RegisterCache – register cache to the bus.  
Bus\_RegisterCacheCallbacks - register cache callback functions to the bus.  
Bus\_RegisterMemoryCallback – register memory callback function to the bus.   
Bus\_AddTransaction – add new transation to bus.  
Bus\_InTransaction – check if bus is in transaction.  
Bus\_Iter – the main iteration of the bus.

Now, we will show the cache implementation.   


The registration functions:  
Register all callbacks from memory and caches.  


The add transation function:  
adding the new request into FIFO buffer in order to preform a round-robin arbitration  


The in transacion function:  


The bus iteration function:  
Taking the wiating transaction and running through the bus mechanism and calling all callback handlers for executing the MESI cache coherence protocol.