Assignment 6\_Part 2: TLP **Date: 17th november 2024**

Shafan Nazeer Ahmed

005030047

**MinorCPU**

The MinorCPU and BasicMinorCPU files in the Gem5 directory contain the configuration of the functional elements of a CPU. The fundamental pipeline segments of a process, which include fetch, decode, execute, and writeback, are primarily defined in this segment. Furthermore, these classes provide us with information regarding opLat and issueLat. The opLat is the operation or execution latency of the Functional Unit (FU), which is the duration that the FU is blocked for a process.

**FloatSimdFU**

The BasicMinorCPU refers to the FloatSimdFU as a functional unit that is capable of performing operations related to floating point and Single Instruction, Multiple Data (SIMD). In this scenario, opLat and issueLat are represented by a variety of combinations, including (1, 6), (2, 5), (3, 4), (4, 3), (5, 2), and (6, 1), respectively, with the understanding that opLat + issueLat equals 7. The testing process has been automated by modifying the configuration, rebuilding the Gem5, and running the simulation using the DAXPY binary file to evaluate the efficiency of each configuration.

<https://github.com/sahmed30047/MSCS-531-M51_-Assignment-6/blob/main/basicminorCPU.py>

A computer screen shot of text

Description automatically generated

A screenshot of a computer program

Description automatically generated

The script initiates by defining a sequence of (opLat, issueLat) pairings, where opLat represents the operation latency and issueLat represents the issue latency of the FloatSimd functional unit within the MinorCPU model in gem5). This is accomplished by selecting pairs that maintain a consistent sum, which in this instance is seven cycles. The total latency for processing operations within the functional unit is consistent across various configurations as a result of this constraint. This enables a fair comparison of the performance impact of varying the balance between operation and issue latencies.  
  
The script creates the paths to critical files and directories at the beginning. The following are the URLs to the BasicMinorCPU.py file, which contains the definitions of the functional units for the MinorCPU model; the directory where simulation results will be stored; and the gem5 binary and the simulation configuration script. In order for the script to modify the simulation environment and save and retrieve data from the appropriate locations, it is essential to ensure that these paths are accurate.  
  
The modify\_fyupool function is a critical component of the script that is responsible for modifying the BasicMinorCPU.py file to update the opLat and issueLat parameters for the FloatSimd functional unit. The function reads the current configuration from the file, locates the FloatSimdFU block, and updates the latency values with the current configuration tuple that is being processed. By enabling the dynamic adjustment of the simulation parameters, this method enables the execution of a sequence of simulation runs with varying configurations without the need for manual intervention.  
  
The script automates the process of rebuilding the gem5 simulator to incorporate the changes using the scons build system after the configuration file is modified. It then executes the simulator by employing a predefined command that specifies the binary for the daxpy kernel, which executes a double-precision array operation that is frequently employed as a benchmark in scientific computing. In order to guarantee that the data from each configuration is preserved for subsequent analysis, the output from each simulation run is redirected to a file within the results directory that is uniquely designated.  
  
By iterating over each configuration tuple, the script applies the modifications, rebuilds the simulator, and executes the simulation in sequence. The script's objective is to investigate the performance space defined by various balances of operation and issue latencies in the FloatSimd functional unit, and this loop is essential to that. Extensive performance testing that would be labor-intensive to undertake manually is facilitated by the script's automation of this process.

**Multi-Threaded DAXPY Kernel Simulation**

The DAXPY kernel commences by allocating memory for two vectors, x and y, each of which has a size defined by VECTOR\_SIZE. The initialization of these vectors is such that each element of x begins at 0.01 and increases by 0.01 for each subsequent element, and similarly for y, which begins at 0.02. This configuration establishes a consistent operations baseline that is applicable to various threads and cycles.  
  
Implementation of Threading  
The computation is parallelized across multiple threads, with the number of threads being determined by NUM\_THREADS. In order to ascertain the portion of the vector that each thread will process, the vector size is divided by the number of threads. For example, if VECTOR\_SIZE is 1,000,000 and NUM\_THREADS is 4, each thread will process 250,000 elements of x and y.  
  
The daxpy\_thread function is executed by each thread, and it is passed an integer that designates the segment of the vector they are responsible for. The thread calculates the DAXPY operation for its designated segment within this function. It effectively updates the y vector in situ by scaling each element of x by a and adding it to the corresponding element of y.  
  
Coordination  
The main function utilizes pthread\_join to wait for the completion of all threads' execution after they are created and assigned specific computations. This synchronization guarantees that the main program will only continue after all parallel computations have been completed, which is essential for achieving consistent and precise results throughout the entire vector.  
  
Resource Management  
The allocated memory for vectors x & y is freed to prevent memory leaks upon the conclusion of the computations, a critical practice in the effective management of resources in C programming. Furthermore, the kernel outputs the initial ten elements of the updated y vector to facilitate a rapid visual confirmation of the computation's accuracy.  
  
Practical Consequences  
The efficiency advantages that can be achieved through parallel processing are illustrated by this implementation of the DAXPY operation. In comparison to a single-threaded approach, the kernel can substantially accelerate computations on large data sets by distributing the workload across multiple threads. This model is especially advantageous in situations where the rapid processing of large vectors is necessary, such as in numerical simulations and data analysis tasks.

A screen shot of a computer

Description automatically generated

A computer screen shot of text

Description automatically generated

A black background with white text

Description automatically generated with medium confidence

<https://github.com/sahmed30047/MSCS-531-M51_-Assignment-6/blob/main/Multi-Threaded%20DAXPY%20Kernel.c>

**Performance Analysis**

The multi-threaded DAXPY kernel's performance is analyzed using a variety of opLat and issueLat parameter configurations in the manner previously described. In all instances where the opLat increases, the operational latencies also increase, resulting in an increase in the overall simulation time. Conversely, the issueLat is decreasing as opLat increases, resulting in a greater number of pipeline delays and a less efficient parallel processing process.   
The parallel speedup increases as the number of threads increases; however, the parallel speedup is reduced by higher latencies (for both opLat and issueLat) and threads must wait for the availability of functional units. Similar observations are present for Instructions Per Cycle (IPC), as the inactive period for functional units is extended as a result of the increasing values of opLat and issueLat. In order to achieve effective performance, it is necessary to use lower Cycles Per Instruction (CPI) values, as each instruction requires fewer cycles to execute. However, the instructions may necessitate additional cycles for execution when the latencies, such as opLat and issueLat, increase. Additionally, the inefficiency of resource scheduling is exacerbated by the increasing values of opLat and issueLat, which may result in thread synchronization issues. Consequently, in order to optimize the utilization of functional units, it is necessary to have a high FloatSimdFU. The performance of a system is reduced as a consequence of the delayed synchronization caused by higher opLat and issueLat. In parallel systems, the speedup can be reduced by synchronization overhead, such as locks and barriers.