Superscalar out-of-order simulator

A detailed report

CSEN 702: Microprocessors
Dr. Mohamed Taher
Eng. Jailan Salah El-Din

Name	ID
Rana Saleh El-Garem	28-3554
Menna El-Kashef	28-3216
Aly Khaled Yakan	28-1294
Kareem Tarek	28-1181
Ahmed Etefy	28-3954
Moustafa Mahmoud	25-3751

Introduction

Packages' Contents:

The following table lists the java files inside each package:

InputParse	FunctionalUnits	MemoryHierarchy	Tomasulo
BufferedReaderExample	FunctionalUnit	Block	Instruction
Parser	AddFunctionalUnit	Cache	InstructionBuffer
RISCDecoder	JALRFunctionalUnit	Memory	RegisterFile
	LoadFunctionalUnit	MemoryHierarichy	RegisterStatusTable
	MultiplyFunctionalUnit	Set	ReorderBufferTable
			ReorderBufferEntry
			ResirvationStation
			Tomasulo

Implementation

InputParse.BufferedReaderExample

Responsible for reading a file, containing the program code and configurations, and returning each line separately as a string in an array list.

InputParse.RISCDecoder

Takes an assembly instruction as a string and encodes it to 16 bit in binary.

InputParse.Parser

This is responsible for initializing everything in our architecture. It retrieves the configuration and initializes the caches, memory hierarchy, tomasulo as well as encoding the assembly instructions, placing them inside our memory along with the program data (if any) specified by the user.

Tomasulo.Instruction

Represents an instruction in the form of: Type, Rs, Rd, Rt, imm. These fields are set in the constructor according to the opcode of each instruction using the 16 bits fetched from memory.

Tomasulo.InstructionBuffer

Holds a certain number of instructions (specified in the configuration) fetched from memory which are waiting for the issue stage. Its head points to the instruction which is next in turn for issuing and a tail pointing at the next empty place for fetching an instruction (if any).

Tomasulo.RegisterFile

Holds an array containing the registers' values, from R0 -> R7. R0's value is final.

Tomasulo.RegisterStatusTable

Holds an array the size of the register file. Each cell will indicate if its corresponding register is occupied by the Reorder Buffer and will also indicate which entry holds the register.

Tomasulo.ReorderBufferEntry

Represents a Reorder Buffer Entry which has a type, destination, value and a ready bit.

Tomasulo.ReorderBufferTable

It has an array of Reorder Buffer Entries with a size specified by the user. Its head points at the entry which is next in turn to commit, while the tail points at the next empty space (if any).

Tomasulo.ReservationStation

Represents a normal Reservation Station with additional attributes:

- **FunctionalUnit FU**: Each reservation station has a corresponding functional unit.
- **boolean start_execute**: indicates whether the FU has started executing the instruction or not.
- **int cycles:** total number of cycles of the functional unit.
- **int exec_cycles left:** number of cycles left in execution.
- **boolean calculated_address:** for load RS. Indicates if the load/store address is calculated or not.

- **int cache_level:** Represents the cache level which we will load from (for load instructions only). Keeps decrementing until we reach level 1.
- **boolean accessed_cache:** Determines whether the cache_level is accessed or not by the RS.
- **String lowByte:** the lowByte of the data to be loaded (Only for Load RS)
- **boolean lowByteSet:** indicates whether or not the low byte is loaded from the memory hierarchy.
- **boolean start_store:** set to true when the store address is calculated
- **boolean write_low_byte:** Only in stores, if it is true, the low byte is written
- **boolean write_high_byte:** Only in stores, if it is true, the high byte is written

FunctionalUnits.FunctionalUnit

An abstract Functional Unit class, that all the functional units inherit from. Each reservation station has an instance of this class. Below are all the Functional units and their operations

Functional Unit	Operations
AddFunctionalUnit	Responsible for add, subtract, nand and branch
LoadFunctionalUnit	Responsible for calculating the address for load and store instructions
MultiplyFuncionalUnit	Responsible for multiply
JALRFunctionalUnit	Responsible for JALR instructions

Tomasulo.Tomasulo

This is the class that simulates the implementation of the microprocessor. It has the following attributes

- **MemoryHierarchy mem_hierarchy:** represents the Memory
- **ReservationStation [] reservation_stations:** Array of Reservation stations
- **RegisterFile register_file**
- **RegisterStatusTable register_statuses**
- **ReorderBufferTable reorder_buffer**
- **InstructionBuffer instruction_buffer**
- **int way:** number of instructions that can be issued simultaneously
- **int cycles:** Number of cycles spanned
- **int instructions_completed:** Number of instructions completed
- **int PC:** PC register pointing to the instruction to be fetched
- **int PC_END:** Address of PC where program ends
- **int instructions_executed:** Number of instructions executed
- **ArrayList<Integer> RS_indices:** Indices of RSs in the order they were issued
- **int branches:** Number of branches encountered
- **int mispredictions:** Number of miss predictions encountered
- **boolean jump_stall:** stalling the fetch stage, if the register in the jump instruction is not ready
- **boolean low_byte_set:** set to true when low byte of instruction is read (In fetch stage)
- **String low_byte:** low byte of instruction being fetched
- **Instruction jump_instruction**: save the jump instruction to be able to get it after stalling

The Class has Five main methods each representing a stage (Fetch, Issue, Execute, Write, Commit).

Fetch():

If the PC doesn't reach the end of the program and the Instruction Buffer is not full, instruction is fetched from the memory; using a helper method **getInstructionFromMem** which returns the instruction only when the access time of the memory hierarchy is over. If the helper method returns null, the fetch method is left. Otherwise, first the instruction is added to the instruction buffer then it is checked if the instruction is jump, return, jalr or branch. The PC is changed to a new value specified by any of the previous instructions, only if its Rt register is ready. Otherwise the fetching stage is stalled until the register becomes ready. If the instruction is none of the above, the PC is incremented by 2

Issue():

We check for an available entry in the Reorder Buffer and a reservation station available for the instruction. If any of them are not available, we can't issue an instruction at this cycle and leave the method. Otherwise, we remove the instruction from the instruction buffer and assign a reservation station to it, and add the index of the reservation station in the **ArrayList RS_indices** (to insure writing in order). We set the attributes of the RS according to the instruction and the same goes for the ROB entry. And we change the status of the destination register in the **register_statuses** to the index of the ROB entry However, if the instruction is a jump or return, we only check for an available ROB entry. We repeat this process for as many times as the number of ways set. But if any instruction can't issue, we stop because issuing has to be in order.

Execute():

We iterate through all the reservations stations, and if the reservation station is busy, we check on the register readiness. If they are ready, we decrement the **execution_cycles_left** which is equivalent to the cycles of the corresponding Functional Unit. When the **execution_cycles_left** reaches 0, we set the result of the RS to the value returned from the Functional Unit, and we increment the number of instructions executed. The branch instruction has a few extra checks to indicate if the prediction was correct or not correct. This goes for all instructions except for load, store. For a store instruction, address is calculated. For a load instruction, address is calculated in one cycles and the cache level where the data is the found is set in the RS. In the next cycle, we decrement the number of cycles of the cache level set. when it reaches 0, we decrement the cache level and do this again until we reach level one; where we read the data from. when the data is read, it is set to the Result of the RS.

Write():

We iterate through all reservation stations using the ArrayList RS_indices (Which ensures order) and check if the **execute_cycles_left** reaches 0. If so, we set the corresponding ROB ready bit to true, and set the value of the ROB entry to the result of the RS. And we iterate through all the reservation stations again to check if any of them needs the value that was just written and reset its corresponding Vj or Vk. Then we remove the RS from the **RS_indices.** This goes for all instructions except for store. In the case of a store, we do the same thing done in load in the execution stage, until the address we want to write in is cached. And then we take two more cycles to store the low byte then the high byte. This will be done as many times as the number of ways, if instructions are ready to write.

Commit():

The ROB entry the head is pointing to is checked. If it is null, we leave the method. Otherwise, we check if it is ready. and then the register file is accessed and the value of the destination value is changed to the value of the ROB entry, and reset the status of the destination register in the **register_statuses.** Then the ROB entry is removed and increment the instructions completed. If the instruction is branch, we increment the number of branch instructions and check if the branch was predicted correctly or not. If it was mis predicted, the reorder buffer, register_statuses, the reservation stations, instruction buffer is flushed, and the PC is set to the address of the instruction that should have been fetched. However, if it was predicted correctly, the ROB entry is just removed.
In the case of store, jump and return, we only remove its corresponding ROB entry and increment the instructions completed. This is done as many as the number of ways, but if an instruction can not commit, nothing can commit afterwards because committing must be in order.

MemoryHierarchy.Block

Represents a block in a set of a cache. It has a tag, valid bit, dirty bit and array of bytes in which its length is determined by the block size.

MemoryHierarchy.Set

Represents a set in a cache. It has an array of Blocks. Its size is determined by the associativity of the cache it belongs to.

MemoryHierarchy.Memory

Represents the main memory, and has an array of bytes of size 65536 and an access_time. Also it has the following additional attributes:

- **boolean being_accessed:** indicates if the memory is being accessed by a load instruction or not
- **int fetch_cycles_left:** to count the number of cycles left to return the an instruction byte in the fetch stage
- **int load_cycles_left:** to count the number of cycles left to return the byte needed by the load instruction
- **int total_cycles:** Total number of cycles spent to access the memory through the whole program

It has the following main methods:
WriteToMemory: writes that takes an index of the bytes array that represents an address in the memory, and the String data that represents the data to be written. And it places this byte, in the given index
ReadFromMemory: takes an index of the array bytes and returns the corresponding String.

MemoryHierarchy.Cache

Represents a single cache with the attributes: Size, blockSize, m, array of Sets, cycles, writePolicyHit, writePolicyMiss, hits, misses and the following additional attributes:

- **boolean fetch_accessed:** Indicates if an instructions is currently being fetched from the cache
- **int fetch_cycles_left:** to count the number of cycles left, to read an instruction byte from the cache in the fetch stage
- **boolean being_accessed:** Indicates if the cache is being accessed by a load instruction. Ensuring that no two load instructions can access the cache at the same time

- **int load_cycles_left:** to count the number of cycles left, taken by a load instruction to read a byte from the cache

It has the following main methods:
read(String address): takes a String address, it determines the index from the address and iterates through all the blocks of the Set with the specified index. It compares the tag of each block in the set, with the tag taken from the address. If a block's tag matches the tag of the address and its valid bit is set, the byte specified by the offset is returned from this block.

write(String address, String data): takes a String address, it determines the index from the address and iterates through all the blocks of the Set with the specified index. It compares the tag of each block in the set, with the tag taken from the address. If a block's tag matches the tag of the address and its valid bit is set, the byte specified by the offset in the found block is set to the data given as an input. And if the cache's writePolicyHit is **writeBack,** the dirty bit of the block is set.

hitOrMiss(String address): takes a String address, it determines the index from the address and iterates through all the blocks of the Set with the specified index. It compares the tag of each block in the set, with the tag taken from the address. If a block's tag matches the tag of the address and its valid bit is set, then true is returned. Otherwise, the method returns false.

MemoryHierarchy.MemoryHierarchy

This class represents the Memory Phase. It has an instance of Memory representing the main memory, and an array of caches. And has the following main methods:

loadCacheLevel(String address): It is used by the load instruction in the execute phase (In Tomasulo class). It returns the cache level, the target byte is found in, or if it is in the memory.

loadValue(String address): After the byte needed by the load instruction is being cached to Level 1, loadValue is called to return the byte needed after taking the cycles needed by cache level 1 (Data Cache)

cacheCyclesLeft(int cacheLevel, String address): called by the load instruction in the execute phase. Using the address given and the cacheLevel it is in, it returns the number of cycles left to read the byte from that cacheLevel. If the number of cycles left reaches 0, the block that has the target byte is cached in the next cache level before it returns 0.

fetchInstruction(String address): It's implementation depends on the fact that instruction can only be fetched one at a time. It keeps on returning null until the cycles of reading the instruction and caching it to level 1(Instruction Cache) are done. It then uses **read_instruction(String address)** to return the needed instruction byte.

read_instruction(String address): This method is the one responsible for caching of instructions. It iterates through all cache levels until the target byte is found, and then keeps on caching it as a block to the next level. When it reaches the instruction cache (Level 1), it returns the target instruction byte.

The caching is done through the following 3 methods:
read_from_lower_level(int i, String address, boolean instructionOrNot) or read_block_from_memory(String address): Both do the same thing, they return the block in which the target byte specified by the address is in. The block returned is the size of the next level. For example, if caching is done from Level 3 to level 2, and the block size in level 2 is 8 bytes. The block returned will be of size 8 bytes.

Then method **writeBlock(Block block, String indexBits, int cacheLevel):** Takes the block to be cached and the index of the set it should be written in and the level of the cache the block will be written to. It writes the block in this cache level. **writeBlock** uses random replacement policy and if the block that is chosen to be replaced is dirty, **replaceBlock** is called. It writes the block in the lower level.

write(String address, String data): It checks if the data is found in the first level of cache. If it not, the data is cached first to all levels until level 1. Then **write_to_cache** is called that keeps on writing the data recursively to the caches until a cache with a write policy hit of **writeBack** or it reaches the memory.

Division of work

Memory Hierarchy Phase

Menna El-kashef
Rana El-Garem

Tomasulo Phase

Skeleton: Ahmed Etefy, Moustafa Mahmoud, Aly Yakan, Kareem Tarek

Fetch stage: Ahmed Etefy

Issue stage: Moustafa Mahmoud

Execute stage: Aly Yakan, Kareem Tarek

Write stage: Aly Yakan, Kareem Tarek

Commit stage: Aly Yakan, Kareem Tarek

Memory Integration: Rana El-Garem, Menna El-kashef

FileReader and Parser

Ahmed Etefy

Report

User guide

1. First the user should write a program in a .txt file. The format of the program should be exactly like this.

```
MemoryHierarchy
numberOfCacheLevels:
Cache1
S:
L:
M:
writePolicyHit:
writePolicyMiss:
cacheCycles:
mainMemoryCycles:
HardwareOrganization
simultaneousIssuesOfInstructions:
sizeInstructionBuffer:
sizeOfROB:
ADDRS:
ADDCycles:
MULRS:
MULCycles:
LWRS:
LWCycles:
JALRRS:
JALRCycles:
AssemblyProgram
.org
// Assembly Program goes here
endofAssembly
ProgramData
//Data required goes here
//Format: address;value
endofData
```

Following this format and filling in the empty spaces, user should first specify the memory hierarchy inputs:

- Number of cache levels
- S, L, M, writePolicyHit, writePolicyMiss, cacheCycles for each cache level
- Access time for main memory

Then the hardware configurations:

- Number of ways (Number of instructions that can be issued simultaneously)
- Size of instruction Buffer
- Size of reorder buffer
- Number of Reservation Stations and number of cycles needed for the following Functional units available
    - Add Functional Unit
    - Multiply Functional Unit
    - Load Functional Unit
    - JALR Functional Unit

Then the user should the write the assembly program after the line **AssemblyProgram** shown above.

- The origin of the program is first inserted as shown above with a space separating between org and the address where you want the program to start.
- Program should be written on the line following .**org**
- **endofAssembly** must be written after the end of the program.

Then the user should write the program data if any after the line **ProgramData** in the format of "address;value". The last line should be **endOfData**

Here is an example of a program file:

```
MemoryHierarchy
numberOfCacheLevels:1
Cache1
S:256
L:32
M:4
writePolicyHit:writeThrough
writePolicyMiss:writeAllocate
cacheCycles:1
mainMemoryCycles:2
HardwareOrganization
simultaneousIssuesOfInstructions:1
sizeInstructionBuffer:10
sizeOfROB:4
ADDRS:2
ADDCycles:2
MULRS:2
MULCycles:7
LWRS:1
LWCycles:10
JALRRS:1
JALRCycles:2
AssemblyProgram
.org 100
LW reg1,reg2,50
ADD reg1,reg1,reg1
endofAssembly
ProgramData
50;2
endofData
```

2. The file must be placed in the project's workspace

3. In the Parser.java class, the user should create a new Parser instance in the class of the main method with the name of the program file

```
public static void main(String[] args) {
    Parser s = new Parser("example.txt");

}
```

4. The user can now run the program and the results will be displayed in the console cycle by cycle. This screenshot shows what appears at each cycle. And at the end, The performance metrics are shown.

```
Cycle:1
----------------------
Instrcutions Completed: 0
Instrcutions Executed: 0
----------------------
Branches Encountered: 0
----------------------
Branches Mispredictions: 0
----------------------
Total Cycles spent to access Memory: 1 cycles
----------------------
Cache 0: Accesses: 1, Misses: 1
Cache 1: Accesses: 0, Misses: 0
----------------------
Instruction Buffer
Head: 0
Tail: 0
----------------------
ROB
Head: 0
Tail: 0
----------------------
Reservation Stations
Name: ADD1, Busy:false, Op:, Vj: 0, VK: 0, Qj: -1, Qk: -1, Dest:0, A: 0
Name: ADD2, Busy:false, Op:, Vj: 0, VK: 0, Qj: -1, Qk: -1, Dest:0, A: 0
Name: MUL1, Busy:false, Op:, Vj: 0, VK: 0, Qj: -1, Qk: -1, Dest:0, A: 0
Name: MUL2, Busy:false, Op:, Vj: 0, VK: 0, Qj: -1, Qk: -1, Dest:0, A: 0
Name: LW1, Busy:false, Op:, Vj: 0, VK: 0, Qj: -1, Qk: -1, Dest:0, A: 0
Name: JALR1, Busy:false, Op:, Vj: 0, VK: 0, Qj: -1, Qk: -1, Dest:0, A: 0
----------------------
Register Status
R0: -1, R1: -1, R2: -1, R3: -1, R4: -1, R5: -1, R6: -1, R7: -1,
----------------------
Register File
R0: 0, R1: 0, R2: 0, R3: 0, R4: 0, R5: 0, R6: 0, R7: 0,
------------------------------------------------------------------------
```

Programs

Program 1:

Assembly Program:

.org 100
ADDI reg1,reg1,2
SW reg1,reg0,6
LW reg5,reg0,6
MUL reg2,reg5,reg5
NAND reg3,reg2,reg5

First Configuration: 2 Cache Levels
MemoryHierarchy
numberOfCacheLevels: 2
Cache1: S:128, L:8, M:1,
writePolicyHit:writeThrough
writePolicyMiss:writeAllocate
cacheCycles:1
Cache2, S:256, L:32, M:4
writePolicyHit:writeThrough
writePolicyMiss:writeAllocate
cacheCycles:5
mainMemoryCycles:6
Hardware Organization
simultaneousIssuesOfInstructions:1
sizeInstructionBuffer:10
sizeOfROB:10
ADDRS:2, ADDCycles:2
MULRS:2, MULCycles:7
LWRS:1, LWCycles:10

Obtained Results:

```
---------------------
Register File
R0: 0, R1: 2, R2: 4, R3: -1, R4: 0, R5: 2, R6: 0, R7: 0,
---------------------------------------------------------------------
Performance Metrics
---------------------
Total Execution Time: 43 cycles
IPC: 0.11627906976744186
AMAT: 2.8
Branch Misprediction Percentage: NaN%
Cache 1 (Instruction) --> hit ratio: 0.8
Cache 1 (Data) --> hit ratio: 0.8
Cache 2 --> hit ratio: 0.3333333333333333
---------------------------------------
```

Second Configuration: 1 Cache Level
MemoryHierarchy
numberOfCacheLevels: 1
Cache1: S:256, L:32, M:4
writePolicyHit:writeThrough
writePolicyMiss:writeAllocate
cacheCycles:5
mainMemoryCycles:6
Hardware Organization
simultaneousIssuesOfInstructions:1
sizeInstructionBuffer:10
sizeOfROB:10
ADDRS:2, ADDCycles:2
MULRS:2, MULCycles:7
LWRS:1, LWCycles:10

Obtained Results:

```
---------------------
Register File
R0: 0, R1: 2, R2: 4, R3: -1, R4: 0, R5: 2, R6: 0, R7: 0,
---------------------------------------------------------------------
Performance Metrics
---------------------
Total Execution Time: 60 cycles
IPC: 0.08333333333333333
AMAT: 5.9
Branch Misprediction Percentage: NaN%
Cache 1 (Instruction) --> hit ratio: 0.9
Cache 1 (Data) --> hit ratio: 0.8
---------------------------------------
```

Program 2:

.org 100
LW reg1,reg2,50
LW reg4,reg2,52
ADD reg2,reg2,reg1
SUB reg4,reg4,reg1
BEQ reg4,reg0,2
JMP reg0, -8
ADDI reg7,reg7,9
endofAssembly
ProgramData
50;1
52;10
endofData

First Configuration: Size of Instruction buffer is 2
MemoryHierarchy
numberOfCacheLevels: 1
Cache1: S:256, L:32, M:4
writePolicyHit:writeThrough
writePolicyMiss:writeAllocate
cacheCycles:5
mainMemoryCycles:6

Hardware Organization
simultaneousIssuesOfInstructions:1
sizeInstructionBuffer:2
sizeOfROB:10
ADDRS:2, ADDCycles:2
MULRS:2, MULCycles:7
LWRS:1, LWCycles:10

Obtained Results:

```
--------------------
Register File
R0: 0, R1: 1, R2: 10, R3: 0, R4: 0, R5: 0, R6: 0, R7: 9,
---------------------------------------------------------------------
Performance Metrics
--------------------
Total Execution Time: 186 cycles
IPC: 0.22580645161290322
AMAT: 2.7848837209302326
Branch Misprediction Percentage: 10.0%
Cache 1 (Instruction) --> hit ratio: 0.9883720930232558
Cache 1 (Data) --> hit ratio: 0.75
---------------------------------------
```

Second Configuration: Size of instruction buffer is 6
MemoryHierarchy
numberOfCacheLevels: 1
Cache1: S:256, L:32, M:4
writePolicyHit:writeThrough
writePolicyMiss:writeAllocate
cacheCycles:5
mainMemoryCycles:6
Hardware Organization
simultaneousIssuesOfInstructions:6
sizeInstructionBuffer:10
sizeOfROB:10
ADDRS:2, ADDCycles:2

MULRS:2, MULCycles:7
LWRS:1, LWCycles:10

Obtained Results:

```
--------------------
Register File
R0: 0, R1: 1, R2: 1, R3: 0, R4: 9, R5: 0, R6: 0, R7: 9,
--------------------------------------------------------------------
Performance Metrics
--------------------
Total Execution Time: 40 cycles
IPC: 0.15
AMAT: 2.95
Branch Misprediction Percentage: 100.0%
Cache 1 (Instruction) --> hit ratio: 0.9333333333333333
Cache 1 (Data) --> hit ratio: 0.75
----------------------------------------
```

Program 3:

.org 30
ADDI reg1,reg1,36
JALR reg2,reg1
BEQ reg0,reg0,6
ADDI reg5,reg0,5
ADD reg6,reg1,reg1
RET reg2
LW reg7,reg0,10
endofAssembly
ProgramData
10;7
endofData

First Configuration: Block Size of level 1 is 8, Block Size of level 2 is 32
MemoryHierarchy
numberOfCacheLevels: 2
Cache1: S:128, L:8, M:1,
writePolicyHit:writeThrough

writePolicyMiss:writeAllocate
cacheCycles:1
Cache2, S:256, L:32, M:4
writePolicyHit:writeThrough
writePolicyMiss:writeAllocate
cacheCycles:5
mainMemoryCycles:6
Hardware Organization
simultaneousIssuesOfInstructions:4
sizeInstructionBuffer:10
sizeOfROB:10
ADDRS:2, ADDCycles:2
MULRS:2, MULCycles:7
LWRS:1, LWCycles:10

Obtained Results:

```
------------------------------------------------------------------------
Performance Metrics
--------------------
Total Execution Time: 32 cycles
IPC: 0.21875
AMAT: 2.1838235294117645
Branch Misprediction Percentage: 100.0%
Cache 1 (Instruction) --> hit ratio: 0.8235294117647058
Cache 1 (Data) --> hit ratio: 0.5
Cache 2 --> hit ratio: 0.5
----------------------------------------
```

Second Configuration: Block Size of level 1 is 2, Block Size of level 2 is 4
MemoryHierarchy
numberOfCacheLevels: 2
Cache1: S:128, L:2, M:1,
writePolicyHit:writeThrough
writePolicyMiss:writeAllocate
cacheCycles:1
Cache2, S:256, L:4, M:4
writePolicyHit:writeThrough

writePolicyMiss:writeAllocate
cacheCycles:5
mainMemoryCycles:6
Hardware Organization
simultaneousIssuesOfInstructions:4
sizeInstructionBuffer:10
sizeOfROB:10
ADDRS:2, ADDCycles:2
MULRS:2, MULCycles:7
LWRS:1, LWCycles:10

Obtained Results:

```
---------------------
Register File
R0: 0, R1: 36, R2: 34, R3: 0, R4: 0, R5: 5, R6: 72, R7: 7,
-----------------------------------------------------------------------
Performance Metrics
---------------------
Total Execution Time: 43 cycles
IPC: 0.16279069767441862
AMAT: 2.766544117647059
Branch Misprediction Percentage: 100.0%
Cache 1 (Instruction) --> hit ratio: 0.5882352941176471
Cache 1 (Data) --> hit ratio: 0.5
Cache 2 --> hit ratio: 0.375
---------------------------------------
```

Discussion of obtained results

Looking at each program's result, few observations were found. First, Adding another cache level and decreasing the block size of the first level affected the AMAT significantly from **5.9** to **2.8**, and significantly decreased the number of cycles to from **60** to **43**.

Second, Increasing the size of the instruction buffer affected the program execution time significantly. Program run with an instruction buffer with size 2 took **186 cycles**, while increasing the size to 6 finished in **40 cycles**.

Finally, Increasing the block size in each level affected the AMAT, Execution time and the hit ratio. Increasing the block size of level 1 from 2 to 8 and of level 2 from **4** to **32**, changed the execution time from **43** cycles to **32** cycles; AMAT to **2.766** to **2.18**; and increased the hit ratio of the instruction cache from 0.588 to 0.82 and the hit ratio of level 2 from **0.375** to **0.5**.

In conclusion, we found that certain factors can increase the program's performance metrics. Adding another level of cache, Increasing the size of the instruction buffer and increasing the block size to a certain size.