MIPS_Simulator

Done by
E.Nitin(**CS18B012**)
Nagarjuna.K(**CS18B018**)
Satya Sai(**CS18B024**)

- A List named **reg** of size 32 is defined for representing 32 registers.
- A dictionary named **memory_dictionary(**in which KEY is memory address and VALUE is value stored in that address**)** of size 1024 is defined for representing 4kb of memory.
- The given assembly file is read line by line and each line is stored as sub_list of **list S[ ]**.
- All the spaces and empty lines are removed.
- All the data elements from the given assembly file is stored in a dictionary named **data_elements.**
- Index of **".main"** is found using a while loop and it is stored in variable **p.**
- All the labels after main are stored in dictionary named **labels**.
- All the instructions present after the **".main"** are executed using a while loop.All these instructions are accessed from the **list S[ ].**

Instructions that can be executed:-

- **li** (load immediate)
- **add**(addition)
- **sub**(subtraction)
- **bne**(branch on not equal)
- **beq**(branch on equal)
- **addi**(add immediate)
- **slt**(set on less than)
- **slti**(set on less than)
- **sll**(shift left logical)
- **la**(load address)
- **lw**(load word)

- **sw**(store word)
- **j**(jump)
- **move**
- **syscall**
- **jr**(jump register)

■ All the above instructions represents the standard instructions of MIPS(32-bit) assembly language

Phase 2:

- Implemented pipelining in the simulator.
- We used 5 variables namely ins_fetch, ins_decode, execute, memory_stage, writeback for simulating pipeline.
- We start a while loop and activate or deactivate stages in pipeline using the 5 variables according to the flow of the pipeline and stalls detected.
- We incorporated latches for the pipeline stages namely insf_insd, insd_ex, ex_mem, mem_wb.
- We have analysed different cases for getting a stall and implemented them in our simulator.
- If a stall is detected in any stage by the variables for stall detection then we increment the stall variable and stall the pipeline and wait for the next cycle.
- After each cycle we update all the latches.
- In the end we print Register and Memory contents, number of stalls, cycles, instructions and IPC.

Phase 3:

- Implemented 2 levels of cache in the simulator.
- First we **read** the **cache_details.txt** and get the necessary information to build the cache.
- We calculate the number of blocks in each level of cache.
- After that we create **list of sets** based on the associativity(for fully 1 set is created).
- Each set is again **list of blocks**. Each block is a **list of elements** based on the block size.
- Each **element** holds **tag** and **its data value**.
- On seeing a “lw” or “sw” instruction **cache controller** function is called.
- On calling cache controller function first requested address is checked in L1 cache, if it is a **hit** that block is **moved to the starting of the list (LRU policy implementation)**, if it is **miss** then **L2 cache is checked** for that address.
- If it is a hit in L2 that block is moved to the starting of the list **(LRU policy implementation)** and that block is inserted in L1 cache, if it is a miss then main memory is checked for the address. After fetching the value from main memory it is inserted to L1 cache.

get_tag_ind_off(a,b):

It reads the decimal address and converts into **binary(32 bits)** and then returns a list containing tag bits, index and offset.

insert_L1_cache(a,value):

It gets the tag,index and offset of the address and goes into the cache and **checks** if there is **space to insert**.
If there are **no blocks** with empty spaces, then it will **pop out** element from the **last block** and insert the given value into that block and move it to starting of the list.The **popped out** element is **inserted into L2 cache**.

Same procedure is done for inserting into L2 cache.

- **Miss rate** is calculated as (no.of misses in that level/cache accesses to that level)
- We designed cache according to **writeback policy**. We have implemented **LRU policy.**

Note:-

- A sample input file**(bubble_sort.asm)**(This file has been updated after phase 2)attached along with the code.In case of changing the input file,update the **input_file** present in **MIPS_sim.py** at **line no.7**
- In case of updating the data of given **bubble_sort.txt** file,update the values in $s3,$s4 as they represent N-2,N respectively.N represents number of elements to be sorted
- A sample input file**(cache_details.txt)** is attached along with the code. For specifying(inputs) cache size, block size, associaivity, latency update this file.
- If any output is generated due to syscall then it can be viewed in the **output.txt** file.