# Nirbhay Sharma (B19CSE114)

# lab-2 - Digital Systems Lab

**Ans-1**. yes, there are some observable changes that are visible in stats.txt file while running sorting algorithm code vs running a hello world program, the reason for this is, we know that sorting algorithms are more complex and big instructions (takes time (nlogn)) than simple hello world printing program so we can also say that number of instructions are more in sorting algorithm case so simulation time and memory both will eventually increase in mergesort/quicksort case as compared to hello world program.

the stats file in step1 is obtained with less inputs in merge sort algorithm and stats file in step2 is obtained with more number of inputs in same algorithm, so there are slight differences among both the files, listed below:

- simulation time is slightly more in stats.txt_step2 file (obvious reason would be that more number of inputs required more simulation time)
- it requires more simTicks in step2 file rather than step1 file which means that more cpu ticks are required
- the number of instructions and number of operations simulated are more in stats.txt_step2 file rather than step1 file, reason may be, because since more inputs are there in file2 so the number of operations and instructions will increase with input size and hence more simulation time is required
- more number of cpu cyles are required in step2 simulation than step1

yes, the same trend is obtained in files obtained in step3, step4 because they follow the same way as step1 & step2, just the algorithm here is quicksort instead of mergesort, in stats.txt_file3, stats.txt_file4 we can observe the same differences as in step1 and step2 files has like simulation instructions, operations, simulation time, cpu cycles etc. all follows the same trend same as in step1, step2

**Ans-2**. yes, the two_level.py file is understood clearly and from that three_level_cache.py is implemented from scratch which is given below.

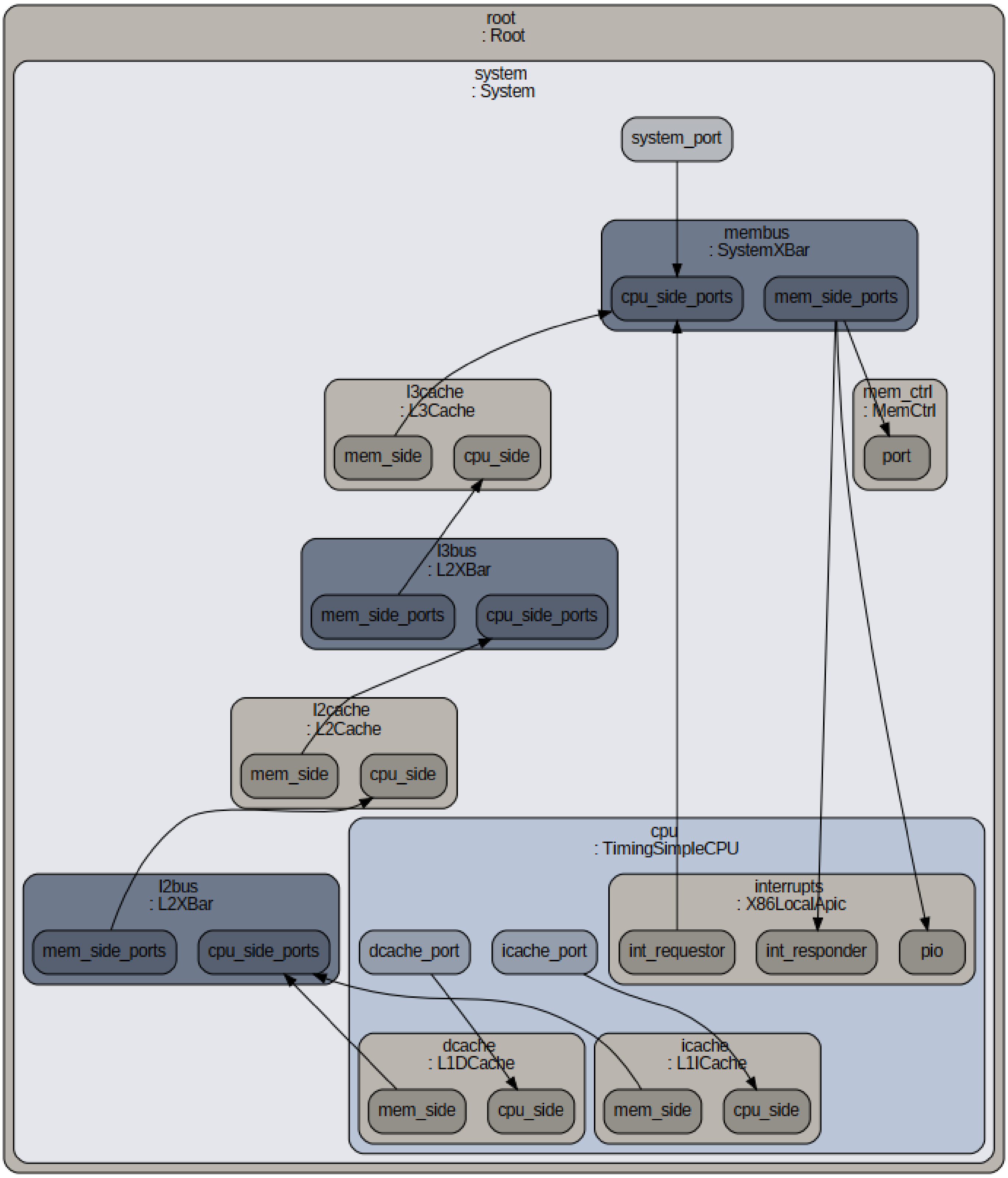
```
    # there is one class which needs to be implemented in caches.py files
given below

    class L3Cache(Cache):
    """Simple L2 Cache with default values"""

    # Default parameters
    size = '256kB'
    assoc = 8
    tag_latency = 20
    data_latency = 20
    response_latency = 20
    mshrs = 20
    tgts_per_mshr = 12

    SimpleOpts.add_option('--l3_size', help="L3 cache size. Default: %s" %
size)
```

```
    def __init__(self, opts=None):
        super(L3Cache, self).__init__()
        if not opts or not opts.l3_size:
            return
        self.size = opts.l3_size

    def connectCPUSideBus(self, bus):
        self.cpu_side = bus.mem_side_ports

    def connectMemSideBus(self, bus):
        self.mem_side = bus.cpu_side_ports
```

```
# import the m5 (gem5) library created when gem5 is built
import m5
# import all of the SimObjects
from m5.objects import *

# Add the common scripts to our path
m5.util.addToPath('../../')

# import the caches which we made
from caches import *

# import the SimpleOpts module
from common import SimpleOpts

# get ISA for the default binary to run. This is mostly for simple testing
isa = str(m5.defines.buildEnv['TARGET_ISA']).lower()

# Default to running 'hello', use the compiled ISA to find the binary
# grab the specific path to the binary
thispath = os.path.dirname(os.path.realpath(__file__))
default_binary = os.path.join(thispath, '../../../',
    'tests/test-progs/hello/bin/', isa, 'linux/hello')

# Binary to execute
SimpleOpts.add_option("binary", nargs='?', default=default_binary)

# Finalize the arguments and grab the args so we can pass it on to our
objects
args = SimpleOpts.parse_args()

# create the system we are going to simulate
system = System()

# Set the clock fequency of the system (and all of its children)
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '1GHz'
system.clk_domain.voltage_domain = VoltageDomain()

# Set up the system
system.mem_mode = 'timing'               # Use timing accesses
system.mem_ranges = [AddrRange('512MB')] # Create an address range
# system.mem_ranges = [AddrRange('512MB')] # Create an address range

# Create a simple CPU
system.cpu = TimingSimpleCPU()

# Create an L1 instruction and data cache
system.cpu.icache = L1ICache(args)
system.cpu.dcache = L1DCache(args)

# Connect the instruction and data caches to the CPU
system.cpu.icache.connectCPU(system.cpu)
system.cpu.dcache.connectCPU(system.cpu)

# Create a memory bus, a coherent crossbar, in this case
system.l2bus = L2XBar()


system.l3bus = L2XBar()




# Hook the CPU ports up to the l2bus
system.cpu.icache.connectBus(system.l2bus)
system.cpu.dcache.connectBus(system.l2bus)

# Create an L2 cache and connect it to the l2bus
system.l2cache = L2Cache(args)
system.l2cache.connectCPUSideBus(system.l2bus)

system.l3cache = L3Cache(args)
system.l3cache.connectCPUSideBus(system.l3bus)
# Create a memory bus
system.membus = SystemXBar()

# Connect the L2 cache to the membus
system.l2cache.connectMemSideBus(system.l3bus)

system.l3cache.connectMemSideBus(system.membus)

# create the interrupt controller for the CPU
system.cpu.createInterruptController()

# For x86 only, make sure the interrupts are connected to the memory
# Note: these are directly connected to the memory bus and are not cached
if m5.defines.buildEnv['TARGET_ISA'] == "x86":
    system.cpu.interrupts[0].pio = system.membus.mem_side_ports
    system.cpu.interrupts[0].int_requestor = system.membus.cpu_side_ports
    system.cpu.interrupts[0].int_responder = system.membus.mem_side_ports
```

```
# Connect the system up to the membus
system.system_port = system.membus.cpu_side_ports

# Create a DDR3 memory controller
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.mem_side_ports

system.workload = SEWorkload.init_compatible(args.binary)

# Create a process for a simple "Hello World" application
process = Process()
# Set the command
# cmd is a list which begins with the executable (like argv)
process.cmd = [args.binary]
# Set the cpu to use the process as its workload and create thread contexts
system.cpu.workload = process
system.cpu.createThreads()

# set up the root SimObject and start the simulation
root = Root(full_system = False, system = system)
# instantiate all of the objects we've created above
m5.instantiate()

print("Beginning simulation!")
exit_event = m5.simulate()
print('Exiting @ tick %i because %s' % (m5.curTick(),
exit_event.getCause()))
```

the simulation file config.dot.pdf file is here:



**Ans-3**. The changes observed in stats.txt obtained in step1 and step7 are listed below:

- simulationTime - it is slightly more in case of step7 (stats.txt)
- simTicks - more in case of step7 (stats.txt)
- host seconds- slightly more in case of step7 (stats.txt)
- number of instructions and operations are more in step7
- number of cycles are more in case of stats.txt obtained in step7
- moreover in step7 stats.txt file we can observe that some new properties are also included in stats.txt file like cacheHit, cacheMiss, cachehitRate, cache-missLatency etc. which is not observed in stats.txt_step6 file.

yes, the same trend is obtained in files obtained in step3 and step6, because here also the coding fashion is same just the algorithm is replaced by quicksort from merge sort, so step3 and step6 (stats.txt files) have differences among simulation time, simulation ticks, simulation cycles, simulated instructions/ operations etc.

**Ans-4**. The differences in stats.txt_step8 and stats.txt_step9 is presented below:

| **property** | **step8(cpu=4)** | **step9(cpu=16)** |
|---|---|---|
| hostSeconds | 0.05 | 0.21 |
| hostTickRate | 183244 | 48402 |
| hostMemory | 695160 | 726388 |

the above differences are due the differences in number of cpu's/processor used in the action. like in first one only 4 cpu's are used and in second one 16 cpu's/processor are used which took more hostMemory but take less hostTickRate than 4 cpu one. Moreover it uses cache and each of the cpu's has it's own cache-controller, memory-controller, directory-controller and they uses CCP (cache coherence protocols for performing read/write operations.).