Design Overview:

In this extra-credit project, we focused on optimizing the cache replacement policies and synthesizing the design to achieve improved performance and correct functionality in a hardware system. Below is a summary of the design improvements, differences from the baseline, the benefits of these optimizations, and the overheads associated with them.

Directory
├── demo3
├── extra_credit
│ ├── dynamic_branch
│ │ ├── verilog
│ │ └── branchFSM.v *(for quick reference to our predictor)*
│ ├── LRU_cache
│ │ ├── verification
│ │ │ ├── mytests
│ │ │ │ └── *two additional tests files*
│ │ │ └── results
│ │ │ │ └── *list of summary.log test files*
│ │ └── verilog
│ │ │ └── *typical processor files (LRU changes are inside cache_controller.v and mem_system.v)*
├── synth
│ └── *contains a list of reports produced by synopsys*
├── post-synth
│ └── *contains all the files needed to run post synthesis simulation*

1. Cache Replacement Policy Improvement

In the baseline design, the I-Cache and D-Cache used a Pseudo-Random Replacement Policy for a 2-way Set-Associative Cache. This policy determines which cache block to evict by randomly selecting between the two available ways. The Pseudo-Random replacement policy utilizes a flip-flop to decide the eviction, typically based on the valid/invalid status of the blocks in each set.

For the extra-credit optimization, we implemented the Least Recently Used (LRU) replacement policy for both I-Cache and D-Cache. The LRU policy works by tracking which cache blocks have been accessed least recently and evicting the least recently used block when a new one needs to be loaded. This approach takes advantage of spatial locality, ensuring that frequently

accessed data remains in the cache, which improves cache hit rates and reduces memory latency. By minimizing cache misses, LRU ensures more data stays in the cache, reducing main memory accesses and CPI. This improvement comes from fewer cycles needed to access memory, especially when accessing the same blocks repeatedly or using nearby memory locations.

To evaluate the improvement from the baseline, we ran tests from Phase 3 and two additional tests we created. The CPI comparisons are as follows:

Baseline CPI:
Test 0:

```
-------------------------------------------------
Final log, saved in summary.log
test.asm SUCCESS CPI:7.5 CYCLES:799 ICOUNT:106 IHITRATE: 65.8 DHITRATE: 13.8
[nandwani@snares-03] (79)$
```

Test 1:

```
-------------------------------------------------
Final log, saved in summary.log
test1.asm SUCCESS CPI:5.2 CYCLES:271 ICOUNT:52 IHITRATE: 51.9 DHITRATE: 23.8
[nandwani@snares-03] (80)$
```

LRU-Optimized CPI:
Test 0:

```
Final log, saved in summary.log
test.asm SUCCESS CPI:5.9 CYCLES:629 ICOUNT:106 IHITRATE: 52.5 DHITRATE: 25.6
(base) [ahuang62@royal-27] (29)$
```

Test 1:

```
-------------------------------------------------
Final log, saved in summary.log
test1.asm SUCCESS CPI:4.9 CYCLES:257 ICOUNT:52 IHITRATE: 51.2 DHITRATE: 27.6
(base) [ahuang62@royal-27] (27)$
```

The results demonstrate an improvement in CPI for our additional tests as well as a reduction in some within our Phase 3 tests, highlighting the effectiveness of the LRU policy in reducing cache misses and improving overall performance. The reduction in CPI indicates that the LRU policy, by maintaining recently accessed blocks in the cache, helps exploit the locality of reference, leading to fewer memory accesses.

The LRU cache replacement policy introduces two potential overheads. First, eviction decision delay, which requires LRU to compare status bit flags to determine which cache block was accessed least recently. This comparison adds a small delay during cache replacements. Second, while LRU reduces the overall number of cache misses, it can result in a potential increase in cache miss penalty. The additional time required to update the flags and make eviction decisions may slightly increase the latency of individual cache misses compared to simpler policies like pseudo-random replacement.

2. Synthesizing the Design

The synthesis was done using Synopsys DC Compiler on a 45 nm standard cell library namely gscl45nm taken from `/u/k/a/karu/courses/cs552/cad/Synopsys_Libraries/libs`.
The baseline design focuses primarily on ensuring core functionality with an emphasis on optimizing CPI. The design preserves hierarchy, keeping modules unflattened to maintain a clear structure for easier analysis and debugging. While the baseline design ensures functional correctness, its main priority is simplicity and CPI optimization rather than more complex improvements.

The optimized design utilizes synthesis to help improve both area efficiency and timing performance. Key improvements include flattening the design hierarchy, which removes unnecessary intermediate module overhead and enables more effective optimizations at the gate level. Redundant combinational logic was identified and removed to simplify the design, and critical timing paths were optimized to reduce delays. Furthermore, smaller and faster gates were chosen for frequently used operations, while underutilized resources were eliminated to save area. Certain modules were also protected from optimization through "don't touch" constraints to maintain their integrity. These optimizations helped to improve the timing, reduce area, and potentially cut down on power consumption, all while improving the design's overall efficiency.

Synopsys produced the following report files explained below :

Timing Report:

```
clock clk (rise edge)                                    1.00       1.00
clock network delay (ideal)                              0.00       1.00
IFID/dff_f_instr/dff01[0]/state_reg/CLK (DFFPOSX1)       0.00       1.00 r
library setup time                                      -0.05       0.95
data required time                                                  0.95
--------------------------------------------------------------------------
data required time                                                  0.95
data arrival time                                                  -2.30
--------------------------------------------------------------------------
slack (VIOLATED)                                                   -1.35
```

- Worst Negative Slack (WNS): -1.35 ns, indicating that the design fails to meet timing constraints under current operating conditions.
- Critical paths primarily traverse through the fetch and execute modules, which might involve complex logic or high fanout nets.

Timing violations suggest that either the clock frequency of 1 GHz (1 ns) is too high for the current logic depth, or certain paths need optimization.

Area Report:

```
Library(s) Used:

    gscl45nm (File: /filespace/n/nandwani2/Downloads/demo3/verilog/libs/gscl45nm.db)

Number of ports:                        10352
Number of nets:                         79457
Number of cells:                        70982
Number of combinational cells:          56492
Number of sequential cells:             12980
Number of macros/black boxes:               0
Number of buf/inv:                      18384
Number of references:                      21

Combinational area:             140088.864126
Buf/Inv area:                    30102.309431
Noncombinational area:          103555.735703
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:                243644.599829
Total area:                 undefined
1
```

- The design occupies a total cell area of 243,644.6 µm², which includes both combinational and sequential logic:
    - Combinational Area: 140,088.86 µm² (57.5% of total area).
    - Non Combinational Area: 103,555.74 µm² (42.5% of total area).
- A significant portion of the combinational area comes from logic operations and buffers/inverters, with buffers/inverters contributing 30,102.31 µm².
- The design consists of 70,982 cells, with most of them being combinational cells (56,492), while 12,980 sequential cells handle stateful operations.

This breakdown indicates a balanced design, where sequential cells (flip-flops and latches) complement the combinational logic effectively. The presence of 18,384 buffers/inverters highlights their role in managing signal integrity and fanout.

Power Report:

```
  Cell Internal Power  = 183.7349 mW   (99%)
  Net Switching Power  =   1.1338 mW    (1%)
                         ---------
Total Dynamic Power    = 184.8687 mW  (100%)

Cell Leakage Power     =   1.1975 mW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000              0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000              0.0000             0.0000  (   0.00%)
clock_network    182.5574           0.0000              0.0000           182.5574  (  98.11%)  i
register       8.1604e-02         3.2770e-02          7.1361e+05           0.8280  (   0.44%)
sequential         0.0000           0.0000              0.0000             0.0000  (   0.00%)
combinational      1.1025           1.1011          4.8388e+05             2.6874  (   1.44%)
--------------------------------------------------------------------------------------------------
Total            183.7415 mW        1.1339 mW       1.1975e+06 nW        186.0728 mW
1
```

- The design uses a total dynamic power of 184.87 mW, distributed as:
    - Cell Internal Power: 183.73 mW (99%), which makes up the majority of the power usage.
    - Net Switching Power: 1.13 mW (1%), indicating low switching activity.
- Leakage Power: 1.20 mW, indicating a small contribution from static power dissipation.

The high internal power suggests most of the power is being used within the cells during state transitions. The low switching power points to efficient signal toggling, though leakage power could become more of an issue in smaller process nodes.

3. Post Synthesis Functional Verification

After completing the synthesis with Synopsys, we used the netlist file "proc.syn.v" for post-synthesis functional verification.

The first challenge we encountered was obtaining the gscl45nm library in a .v format to make it compatible with the wsrun.pl script for vsim. The second challenge involved adapting the proc_hier_pbench testbench to work with the newly optimized netlist. Synopsys had abstracted and removed many of the signals from the original proc design, which caused compatibility issues. We resolved this by using escape characters for Verilog signal names, which allowed us to restore proper signal references. Below is an example of how this was implemented:

```
  // Decode stage signals
assign RegWrite = DUT.p0.decode0.regFile0.writeEn;     // Register write enable
// Is register file being written to, one bit signal (1 means yes, 0 means no)

assign WriteRegister = {DUT.p0.\DMWB_RD<2> , DUT.p0.\DMWB_RD<1> , DUT.p0.\DMWB_RD<0> };  // Register destination address
// The name of the register being written to. (3 bit signal)

assign WriteData = {DUT.p0.wb0.\WB<15> , DUT.p0.wb0.\WB<14> , DUT.p0.wb0.\WB<13> ,
                    DUT.p0.wb0.\WB<12> , DUT.p0.wb0.\WB<11> , DUT.p0.wb0.\WB<10> ,
                    DUT.p0.wb0.\WB<9> ,  DUT.p0.wb0.\WB<8> ,  DUT.p0.wb0.\WB<7> ,
                    DUT.p0.wb0.\WB<6> ,  DUT.p0.wb0.\WB<5> ,  DUT.p0.wb0.\WB<4> ,
                    DUT.p0.wb0.\WB<3> ,  DUT.p0.wb0.\WB<2> ,  DUT.p0.wb0.\WB<1> ,
                    DUT.p0.wb0.\WB<0> };
```

After converting our old signals and finding all the relevant signals in our post synthesis netlist, we connected them to the testbench and successfully ran post synthesis simulation.

During post-synthesis testing, we encountered an issue where the PC would default to 0xXXXX after the reset was deasserted, indicating it wasn't initializing correctly. To troubleshoot, we worked closely with Manu and followed a systematic approach to identify the root cause. We started by reviewing waveform signals to ensure the reset and clock were functioning as expected. Despite confirming that the reset was properly deasserted, the PC wasn't receiving the expected values, which suggested a deeper issue.

Next, we checked the connections to the PC port, verifying if any misconfigurations could be causing the problem. After adjusting the connections and re-running tests, the issue persisted. We then modified our CLA logic to test the PC value before the architecture was flattened, but this also didn't resolve the problem. We also tried removing any signals conditional to the PC and tried a simple run with just an add instruction, but to no avail.

Upon further discussion with Manu and to rule out timing issues, we tested the design at a lower frequency of 100 MHz instead of 1 GHz.This allowed the design to synthesize successfully and meet timing requirements, even though it didn't fully fix the problem. Despite these troubleshooting steps, we were ultimately unable to pinpoint the exact cause of the issue, leaving it unresolved for the time being.

4. Dynamic Branch Prediction

We aimed to improve our processor's performance by implementing a 2-bit dynamic branch predictor to reduce CPI. Our baseline configuration used a static "branch not taken" policy, which struggled with large and random instruction streams. Based on Professor Tannu's lectures, we hypothesized that a 2-bit FSM-based predictor could adapt to branching patterns and improve prediction accuracy.

For our extra-credit design, we used a 2-bit finite state machine (FSM) with four states: Strongly Taken, Weakly Taken, Weakly Not Taken, and Strongly Not Taken. The predictor transitions between these states based on past branch behavior, enabling it to adapt over time. When a misprediction occurs, a signal triggers the decode stage to flush the incorrect instructions and realign the pipeline.

Although we did not fully resolve the issue of misprediction signals not properly redirecting the program counter (PC), we successfully implemented the FSM logic and the flushing mechanism for incorrect instructions. This demonstrates the core functionality of a dynamic branch predictor, even though further optimization is still required.

The dynamic predictor introduces trade-offs, such as added complexity in the pipeline and minor delays due to FSM transitions. However, it significantly enhances branch prediction accuracy, particularly in cases where a static prediction policy would frequently fail.

In summary, our design illustrates how dynamic branch prediction can improve processor performance by adapting to real-time branch behavior. While there are still challenges to address, this approach provides a strong foundation for future improvements.