# Georgia Institute of Technology

## CS4290/CS6290/ECE4100/ECE6100

Programming assignment #5
Progm due: Nov. 30 (F) 6:00 pm T-square. (+5:55 grace period) Automatic extension without any penalty (Dec 7 (F) 11:55 pm)
Prof. Hyesoon Kim
Prof. Moinuddin Qureshi
For CS6290/ECE6100 Students: Instead of the term project, you can choose to the extension version of Prog#5. The deadlines are the same.
This is an individual assignment. You can discuss this assignment with other classmates but you should do the assignment individually.

### Part 1: Adding Counters in the Simulator (10 points)

### Overview

In this assignment, you will extend your simulator to calculate power and energy. To simplify the assignment, we measure power and energy consumption for some important hardware structures only.
The basic approach to calculate power consumption is that you add counters in your simulators. After the end of simulation, you use the collected counter values to calculate power and energy consumption.
You have to download McPAT from McPAT website or here . McPAT is a multicore power and area simulator.
You have to add counters to your simulator. You also need to provide the output values to McPAT as an XML file, since McPAT takes XML files as input. You do not have to generate an XML file directly from your simulator. Your simulator should generate a text file, power_counters.txt, containing the counter values, and then you might manually type the values into an XML file. You can use Excel or other XML editors to edit the XML file. The format of power_counters.txt is not specified. Since we are building a simplified powr model, we do not generate many counter values. For counter values that are not mentioned in this assignment, you just use the default values in the original XML file. Here are the counters that you have to add in your simulator. We also show the how these counters map to attributes (counters) in a McPAT XML file.

- *Overall*
  - Total number of instructions (McPAT: total_instructions, committed_instructions)
  - Total simulation cycles (McPAT: total_cycles, busy_cycles)
  - Number of floating point operations (McPAT: fp_instructions, committed_fp_instructions)
  - Number of integer operations (McPAT: int_instructions, committed_int_instructions)
  - Number of branch instructions (McPAT: branch_instructions)
  - Number of load instructions (McPAT: load_instructions)
  - Number of store instructions (McPAT: store_instructions)
- *Fetch Stage*

  - Number of I-cache accesses (McPAT:icache read_accesses)
  - Number of branch mispredictions (McPAT: branch_mispredictions)

- *Decode Stage*

  - Number of scheduler accesses (i.e. the number of times a scheduler is accessed. Since we do not have a scheduler in our simulator, if there is an instruction in the decode stage, you just increment this counter value. Based on instruction type, you have to increment floating point and integer instructions. McPAT: ROB_reads, ROB_writes, rename_accesses, fp_rename_accesses, inst_window_reads, inst_window_writes, inst_window_wakeup_accesses, fp_inst_window_reads, fp_inst_window_writes, fp_inst_window_wakeup_accesses)
  - Number of register reads for integer operations (McPAT: int_regfile_reads, ialu_access)
  - Number of register reads for fp operations (McPAT: float_regfile_reads, fpu_access) (You can use the number of src operands.)

- *Execution Stage*

  - Number of multiply instructions (OP_IMUL, OP_MM in simulator) (McPAT: mul_accesses)

- *Memory*

  - Number of data cache reads (McPAT: dtlb:total_accesses (read+write), dcache:read_accesses)
  - Number of data cache writes (McPAT: dtlb:total_accesses (read+write), dcache:write_accesses)
  - Number of memory accesses (McPAT: system.Noc0:total_accesses = 4 times of number of memory accesses, system.mem.Memory accesses, system.mem.memory_reads, system.mem.memory_writes, (system.mc.memory_accesses, = 4 times of number of memory accesses system.mc.memory_reads = system.mc.memory_writes = 2 times of number of memory accesses))

- *WB*

  - Number of register writes for integer operations (int_regfile_writes)
  - Number of register writes for fp operations (float_regfile_writes)

- Since there will be multiple answers depending on your simulator design, grades are mainly based on your reports.

**Part 2: Setting up McPAT (10 points)**
You copy Niagra1.xml that comes with McPAT as lab5.xml file and change certain values to match the configuration of your pipelined processor. You need to turn in lab5.xml file. (10 points) The values that you have to modify are:

  - Number of cores:1
  - Number of L1Directories:1
  - Number of L2s: 0
  - Number of memory controllers:1
  - Number of hardware threads: set up based on your simulation configuration.

- pipeline depth: 5-stage.
- Branch predictor (global_predictor_entries, chooser_predictor_entries): the number of entries in the PHT. (determined by the GHR size for G-share)
- Icache/Dcache: icache_config, dcache_config: set up based on cache size.
- set all zeros for L2/L3 accesses (read/write)
- Counter values: Use the outputs from your simulator.
- Set to be zero: function_calls, context_switches

Please see the README file provided with McPAT to learn how to use McPAT.

**Part 3: Simulations (80 points)**

Your simulations are composed of two steps. In the first step, you run your simulator to get hardware performance counter values and in the second step you type the counter values into an XML file and generate Power numbers via McPAT.
The trace is not specified. You can choose what you want to simluate.
For the default configuration (only one case!), you draw stack column graphs for power (Runtime Dynamic) and energy (Runtime Dynamic x execution time) that show the breakdown of the following components: Instruction fetch unit, load&store unit, memory management unit, execution unit, L2, First level directory, NoCs,and MCs. To show more detailed power consumption of the core, you should draw one more diagram from the information that you can collect about core (drop L2, First level directory, NOCs and MCs) from print_level 4 in McPAT.
For all simulations, you need to present performance (IPC), Power (W), Energy (J), and EDP in your report for the following 4 kinds of simulations . You also need to present counter values in your report. Please discuss why certain configuration can save/increase power/energy.

The default configuration is shown below except parameters that are specified in each simulation.

- Simulation 1: Clock frequency effects
  Vary clock frequency (target_core_clockrate): 900, 1200, 1500

- Simulation 2: Branch Prediction effects
  Vary branch predictor size 4, 8, 12

- Simulation 3: Cache size effects
  1MB, 2MB, 4MB

Simulation 4: MT simulations
Vary the number of threads (1,2,4)
The default configuration is as follows:

gshare history length: 12
cache size, way: 1MB 8 way
Dcache latency: 5 cycles
Default clock frequency: 1200
DRAM row buffer hit latency: 100 cycles

DRAM row buffer miss latency: 200 cycles
Number of threads:1
the rest of the parameters are default knob values.

**Submission Guide**

You must include an XML file for the default configuration. The name of the file should be **lab5.xml** Please follow the submission instructions. If you do not follow the submission file names, you will not receive full credit. Please check the class homepage to see the latest updates. Your code must run on the shuttle cluster with g++4.1.

Please do not turn in pzip files(trace files). Trace file sizes are huge so they will cause a lot of problems.
(Tar the lab5 directory. Gzip the tarfile and submit lab5.tar.gz file on T-square)
Please make sure the directory name is lab5!

```
cd lab5
make clean
rm *.pzip
cd ..
tar cvf lab5.tar lab5
gzip lab5.tar
```

---

**Extension of Lab #5 for CS6290/ECE6100 Students**

(40 points) If a student chooses Lab#5 extension instead of the project, the student needs to complete both power simulation and also implement another branch predictor. You will extend your Lab#4 simulator to have one more branch predictor, Perceptron branch predictor.
Virtual memory and MT features will not be tested in this branch predictor evaluation. You can choose between a gshare-branch predictor and perceptron branch predictor. You have to add the following four new KNOBS
KNOB_PERCEP_HIS_LENGTH (default: 20)
KNOB_PERCEP_TABLE_ENTRY (default: 512)
KNOB_PERCEP_CTR_BITS (default: 8 bits)
KNOB_PERCEP_THRESH_OVD (default: 0)
You add one more branch predictor type into bpred_type in both bpred.h and bpred.cpp
bpred_type 4 selects the perceptron branch predictor.

---

Report submission guide.
Submit a report that answers the following questions and also your source code. Your grade is mainly coming from the report. The source code will be used only to check whether you implemented perceptron branch predictor or not.
Use at least two traces to write a report. long_trace1.pzip long_trace2.pzip

(1) (20 points) Compare performance results of g-share branch predictor and the perceptron branch predictor for the following sizes. 1KB, 4KB, 8 KB, and 16KB. Report MPKI. (misses per

thousand instructions.) Discuss the performance why gshare or perceptron predictor performs well. Compute the branch history length for each branch predictor size.

(2) (10 points) For the perceptron predictor only, For 4KB and 8KB, vary the number of table entries (128, 256, 512) and compare the performance. You must adjust the branch history length accordingly for each branch predictor size. Report MPKI.

(3) (10 points) For the 4KB perceptron branch predictor, try at least 3 different threshold and discuss the effects.

---

The following questions to help your final exam preparation.

- [0 point] Show a software-pipelined version of this loop. Assume that you have infinite number of registers.

```
LOOP;  LD F0, 0 (R1)
       Add F4, F0, F2
       Mul F5, F4, F3
       Store F5, 0 (R1)
       Add R1, R1, #-8
       Br R1, R2, LOOP
```

You may omit the start-up and clean-up code.

Without start-up/clean-up code

```
LOOP;  Store F5, 0(R1)
       Mul F5, F4, F3
       Add F4, F0, F2
        LD F0, -24(R1)
       Add R1, R1, #-8
       Br R1, R2, LOOP
```

With all start-up/clean-up code

```
       LD F0, 0(R1)
       Add F4, F0, F2
       Mul F5, F4, F3
       LD F0, -8(R1)
       Add F4, F0, F2
       LD F0, -16(R1)
LOOP;  Store F5, 0(R1)
       Mul F5, F4, F3
       Add F4, F0, F2
        LD F0, -24 (R1)
       Add R1, R1, #-8
       Br R1, R2, LOOP
```

```
Store F5, -8(R1)
Mul F5, F4, F3
Store F5, -16(R1)
Add F4, F0, F2
Mul F5, F4, F3
Store F5, -24(R1)
```

- [0 point]Compare and contrast super-scalar architecture and VLIW architecture.

| VLIW | SuperScalar |
|---|---|
| statically combined | dynamically combined |
| more than one instruction | more than one instruction |
| Static Issue | Dynamic Issue |
| Compiler optimized | Runtime optimized in hardware |
| Can't react to latencies | Can react to latencies in caches, memory, etc |
| Larger view of program may lead to better optimization | Smaller window for optimization |
| Less complex hardware, less power consumed | Complex hardware, more power |
| And more.. | |

- [0 point] For a VLIW machine, the compiler has to find independent instructions that can be executed at the same cycle. Supposed that there is a 3-wide VLIW machine. (i.e., there are three-instruction slots.) For the first and second slots, the processor can execute any types of instructions. For the third slots, the processor can execute all instructions except loads, stores. Arrange the following code to maximize the performance. The processor is an in-order processor. The compiler needs to insert nops if it cannot find any instructions for a slot.

```
0xa000 ADD R0, R0, R2
0xa004 ADD R1, R1, R0
0xa008 FADD F3, F2,  F3
0xa00B ADD R1, R1, R0
0xa010 FADD F3, F2,  F3
0xa014 LD R2, MEM[R6]
0xa008 ADD R1, R1, R0
0xa01B FMUL F1, F5, F4
0xa020 LD F2, MEM[R0]
0xa024 FADD F4, F1, F2
```

```
0xa028 LD F3, MEM[R0]
0xa030 STORE MEM[R5], F4
0xa034 FADD F4, F3, F4
0xa038 ADD R2, R2, R0
0xa03B BR R2, 0x8000
```

ADD: simple ALU instruction (1-cycle latency)
FADD: floating point instruction (1-cycle latency)
FMUL: floating point instruction (2-cycle latency)
LD: load instruction (2-cycle latency)
STORE: store instruction (1-cycle latency)
BR: branch instruction (1-cycle latency)

```
I1: 0xa000 ADD R0, R0, R2
I2: 0xa004 ADD R1, R1, R0
I3: 0xa008 FADD F3, F2,  F3
I4: 0xa00B ADD R1, R1, R0
I5: 0xa010 FADD F3, F2,  F3
I6: 0xa014 LD R2, MEM[R6]
I7: 0xa008 ADD R1, R1, R0
I8: 0xa01B FMUL F1, F5, F4
I9: 0xa020 LD F2, MEM[R0]
IA: 0xa024 FADD F4, F1, F2
IB: 0xa028 LD F3, MEM[R0]
IC: 0xa030 STORE MEM[R5], F4
ID: 0xa034 FADD F4, F3, F4
IE: 0xa038 ADD R2, R2, R0
IF: 0xa03B BR R2, 0x8000

ADD: simple ALU instruction (1-cycle latency)
FADD: floating point instruction (1-cycle latency)
FMUL: floating point instruction (2-cycle latency)
LD: load instruction (2-cycle latency)
STORE: store instruction (1-cycle latency)
BR: branch instruction (1-cycle latency)

Rename I9:F2 to F6
       IA:F2 to F6
       IB:F3 to F7
       ID:F3 to F7

Now the VLIW schedule looks like:

          -----------------------------------------------
         |       I1      |       I6      |       I8      |
          -----------------------------------------------
         |       I9      |       IB      |       I2      |
          -----------------------------------------------
         |       I4      |       I3      |       IE      |
          -----------------------------------------------
         |       I7      |       IA      |       I5      |
          -----------------------------------------------
         |       IC      |       ID      |       IF      |
          -----------------------------------------------
```

- [0 point] Many of x86 instructions are actually broken into multiple RISC type uops when they are executed inside a processor. If one of the uops generates an exception, how should we handle them?

None of the uops from the same instruction can be retired.

- [0 point] (H&P 5.2) You are investigating the possible benefits of a way-predicting level 1 cache. Assume that the 32 KB two-way set-associative single-banked level 1 data cache is currently the cycle time limiter. As an alternate cache organization you are considering a way-predicted cache modeled as a 16KB direct-mapped cache with 85% prediction accuracy. Unless stated otherwise, assume a mispredicted way access that hits in the cache takes one more cycle. Assume that the access time of 32-KB 2-way cache is 2 cycles and that of 16KB direct-mapped cache is 1 cycle.

  - What is the average memory access time of the current cache versus the way-predicted cache?
  - If all the components could operate with the faster way-predicted cache cycle time (including the main memory), what would be the impact on performance from using the way-predicted cache?
  - Way-predicted caches have usually only been used for instruction caches that feed an instruction queue or buffer. Imagine you want to try out way prediction on a data cache. Assume you have 85% prediction accuracy, and subsequent operations (e.g., data cache access of other instructions, and dependent operations, etc.) are issued assuming a correct way prediction. Thus a way misprediction necessitates a pipe flush and replay trap, which requires 15 cycles. Is the change in average memory access time per load instruction with data cache way prediction positive or negative, and how much is it?

- [0 point] A processor has an ECC in register files and all the memory hierarchy. But still it is not free from faults? What kind of faults can still exist? All faults (transient, intermittent, permanent faults) in combinational logics can exit. For example, one of the input to ALU can be always 1 (stuck-at-1 fault).