ELEC3441: Computer Architecture
Second Semester, 2023–24

Homework 2 (r1.0)

Due: 7 Apr, 2024, 11:59pm

Instruction: Submit your answers electronically through Moodle.

There are 3 parts in this homework. Part A includes questions that aim to help you with understanding the lecture materials. They resemble the kind of questions you will encounter in quizzes and the final exam. Part A is an individual portion of this homework. You should be completing and turning in *your own work*.

Part B of this homework are hands-on exercises that require you to design and evaluate processor systems using various software and hardware tools, including the RISC-V compilation tool chains. They are designed to help you understand real-world processor design and the use of various tools to help you along the way. Part B and C are group work. Each group should turn in only 1 set of project file and 1 project report.

The following summarizes the 2 parts.

Part	Type	Indv/Grp
A	Basic problem set	Individual
B	Hands-on	Group of up to 2 persons

In all cases, you are encouraged to discuss the homework problems offline or online using Piazza. However, you should not ask for or give out solution directly as that defeat the idea of having homework exercise. Giving out answers or copying answers directly will likely constitute an act of plagiarism.

Part A: Problem Set

A.1 Resolving Branch Hazards

You are designing the branch logic for a new deeply pipelined processor with 12 pipeline stages. Branches are resolved in stage 5 (stages count from 1). You are faced with three architectural choices to resolve control hazard:

s1. Introduce 4 delay slots;

s2. Speculate branch always-not-taken. Kill instructions if speculated wrong.

s3. Use a branch predictor. Your branch predictor is correct 95 % of the time.

Due to the increased hardware required for hazard detection and to kill the wrongly fetched instructions, the minimum cycle time of the processor is affected as follows:

s1	s2	s3
1 ns	1.5 ns	2 ns

Now, consider the following code sequence for this question:

```
I1|       addi a1, zero, 10
I2|       addi a0, zero, 0
I3| loop: lw   a2, 0(a3)
I4|       lw   a4, 0(a5)
I5|       addi a3, a3, 4
I6|       addi a5, a5, 4
I7|       add  a2, a2, a4
I8|       srli a4, a2, 1
I9|       andi a4, a4, 1
I10|      beqz a4, skip
I11|      add  a0, a0, a2
I12|skip: nop
I13|      addi a1, a1, -1
I14|      bnez a1, loop
I15|      add  a7, zero, a0
I16|      addi a1, zero, 23
I17|      add  a0, a0, a1
```

A.1.1 Assume the array pointed to by `a3` and `a5` contains the following values:

```
int A[10] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 27}; // a3
int B[10] = { 3, 1, 4, 1,  5,  9,  2,  6,  5,  3}; // a5
```

Trace through the above code and record the sequence of branch and directions for the 2 branch instructions.

Sequence	Instruction Address	Instruction	Branch Taken? (Y/N)
1	I10	beqz	Y
2	I14	bnez	Y
3			
4	⋮	⋮	⋮

A.1.2 Consider solution s1 above. Assume pipeline has full forwarding network and consider RAW only. Insert nops and/or rearrange instructions as needed such that the code executes correctly. How many cycles are needed to complete the above code segment?

A.1.3 Consider solution s2 above and assume pipeline has full forwarding network and consider RAW data hazards only. How many cycles are needed to complete the above code segment? How many times does the processor need to kill wrongly fetched instructions?

A.1.4 Consider solution s3 above and assume pipeline has full forwarding network and consider RAW data hazards only. Assume your branch predictor has the same prediction accuracy for both branch instructions. How many cycles are needed to complete the above code segment?

A.1.5 Taking cycle time into consideration, which solution results in the best performance?

A.2 Cache Access

Consider the following sequence of memory accesses to the main memory in a 32-bit processor:

Address (hex)	Type
BA443E98	R
BA443FA8	W
54E99A88	R
54E99BB8	R
30013C80	R
54E99A84	R
54E99BA0	R
BA443FA4	W
54E99A90	W
BA443E88	R

A.2.1 Assume the following data cache organization:

- Capacity: 512 B
- Line size: 8 words
- Organization: direct map
- Policy: write back, write allocate

Below is the current status of your cache:

Cache contents:

index	tag	valid	dirty	data							
				7	6	5	4	3	2	1	0
0	-	0	0	-	-	-	-	-	-	-	-
1	-	0	0	-	-	-	-	-	-	-	-
2	-	0	0	-	-	-	-	-	-	-	-
3	-	0	0	-	-	-	-	-	-	-	-
4	5D221F	1	1	BEEF0007	BEEF0006	0EEE3441	0EEE3441	BEEF0003	BEEF0002	BEEF0001	0EEE3441
5	-	0	0	-	-	-	-	-	-	-	-
6	-	0	0	-	-	-	-	-	-	-	-
7	-	0	0	-	-	-	-	-	-	-	-
8	-	0	0	-	-	-	-	-	-	-	-
9	-	0	0	-	-	-	-	-	-	-	-
10	-	0	0	-	-	-	-	-	-	-	-
11	-	0	0	-	-	-	-	-	-	-	-
12	-	0	0	-	-	-	-	-	-	-	-
13	-	0	0	-	-	-	-	-	-	-	-
14	-	0	0	-	-	-	-	-	-	-	-
15	-	0	0	-	-	-	-	-	-	-	-

You can assume all data in your main memory contain the value `0x0EEE3441`. Trace through the above memory access and answer the following:

(i) For each access, is it a hit or a miss?
(ii) Reason for miss (e.g. Conflict miss due to access X; Compulsory miss, etc)
(iii) Show the final content in the cache, including the tag.

A.2.2 Repeat A.2.1 but with a different cache organization:

- Organization: 2-way set associative
- Policy: true LRU, write back, write allocate

A.2.3 Repeat A.2.1 but with a different cache:

- Organization: Fully Associative
- Policy: write through, no write allocate

A.2.4 Assume you are allowed to make the following modification to the cache:

Cache Parameter	Range
capacity	4 B to 512 B
line size	1 to 256 words
set associativity	1 (direct map) to 4
write miss policy	any valid policy

With the above sequence of memory access, design a cache that would result in the *minimum* number of misses (N_{miss}) using *minimum* capacity (C). That is, you want to minimize the quantity:

$$N_{\text{miss}} \times C$$

Explain your answer.

A.3 Simple Pipeline

In this question, you will examine the operation of pipelined processors using the following assembly code:

```
    # a0 <- 0x00A00000
    # a1 <- 0x00A10000
    add   s0, zero, a0
    add   s1, zero, a1
    add   s2, zero, a2
    add   s3, zero, a3
    addi  t0, zero, 100  # <-- N=100
L1: lw    t1, 0(s0)
    lw    t2, 0(s1)
    add   t3, t1, t2
    add   a0, a0, t3
    sw    t3, 0(s2)
    sub   t3, t1, t2
    add   a1, a1, t3
    sw    t3, 0(s3)
    addi  s0, s0, 4
    addi  s1, s1, 4
    addi  s2, s2, 4
    addi  s3, s3, 4
    addi  t0, t0, -1
    bne   t0, zero, L1
    add   a0, a0, a0
    add   a0, a0, a0
    add   a0, a0, a0
    add   a0, a0, a0
    add   a1, a1, a1
    add   a1, a1, a1
    add   a1, a1, a1
    add   a1, a1, a1
```

Consider a processor, P, which has the following properties:

- It has a pipeline with the following 7 stages:

  ```
  IF1 IF2 ID EX MA1 MA2 WB
  ```

- It only has *one forwarding path from EX to ID*, i.e., only from output of the ALU at the end of EX stage to the pipeline register between ID and EX. Any data hazards that cannot be resolved by the forwarding path will cause the processor to stall while bubbles are inserted.
- Cache hit takes 2 cycles. Cache access is fully pipelined, so one new access can be requested every cycle.
- Branches are resolved in EX stage in parallel to the ALU. The processor speculates "always not-taken" on branches, i.e., the processor continues fetching and decoding instructions and kill the speculated instructions if branch is taken.

A.3.1 Assuming all elements in the array pointed to by `a0` and `a1` contain the *values* 100 and 17 respectively. Trace through the code. What is the value of `a0` and `a1` when the code fragment completes?

A.3.2 How many instructions are being *executed to completion* during run time?

A.3.3 List out all data hazards in the above code that would cause a stall in the pipeline.

A.3.4 Counting the number of instruction committing at WB stage, what is the average CPI of the above code?

A.4 Streaming Cache Performance

You are investigating the cache performance of your processor regarding the following code segment:

```
// int i, N, a, b;
// int y[], x[];
for (i = 0; i < N; i++) {
  y[i] = a*x[i] + b*y[i];
}
```

Now the above C code is compiled into the following RISC-V instructions:

```
# a0 initially contains the constant N
# a1 is base address of x[]
# a2 is base address of y[]
# constant a is stored in register a3
# constant b is stored in register a4
00| loop: lw    t1, 0(a1)
04|       lw    t2, 0(a2)
08|       mult  t0, t1, a3
0C|       mult  t1, t2, a4
10|       add   t0, t0, t1
14|       sw    t0, 0(a2)
18|       addi  a1, a1, 4
1C|       addi  a2, a2, 4
20|       addi  a0 a0 -1
24|       bne   a0, zero, loop
```

A.4.1 Cache Hit or Miss Assume array `x` is stored at memory address starting at `0xEA000000` while the array `y` is stored at memory address `0xEB000000`.

The processor data cache is initially empty and has the following properties:

- Capacity: 1 MiB
- Organization: 2-way set associative
- Line size: 8 words
- Replacement policy: true LRU
- Write policy: write through, no write allocate

A large write buffer is available such that the processor can resume running immediately after writing the data to the buffer.

Let $N = 2^{10}$, trace through the above code, then show and explain the sequence of cache hit or miss that will occur. Use the notation **RH** for read hit, **WH** for write hit, **RM** for read miss, and **WM** for write miss.

A.4.2 Based on your result above, what is the overall data cache miss rate of the above code when executed in the main CPU? Consider BOTH read and write access. Assume the write and read miss penalties are both 300 cycles. Hit time is 1 cycle. What is the average memory access time (AMAT) for this code?

A.4.3 For simplicity, assume there is no pipeline in the processor. Furthermore, assume `add`, `addi` and `bne` takes 1 cycle; `mult` takes 4 cycles; and performance of `lw` and `sw` depends on the cache performance. What is the total run time of the above code in cycles? You may leave the variable N in your answer.

A.4.4 If instead the size of the array $N = 2^{23}$, how would the access pattern and cache miss rate be affected?

A.4.5 Write Back Cache If the write policy of the cache is changed to *write back, with write allocate*, how would the following be affected? Consider 2 cases: (i) with write buffer, (ii) without write buffer.

(i) Read miss penalty
(ii) Write miss penalty
(iii) Overall performance of the above code

Explain your answer by tracing through the execution of the above code, highlighting any different in the required cache content handling with the use of write back cache.

Part B: Open-ended Project

B.1 Adapting Cache Characteristics

In this exercise, you will investigate and try to improve the cache performance of a set of benchmarks. To evaluate the benchmark performance, you will be using a RISC-V ISA simulator called `spike`. The `spike` simulator simulates the behavior of the RISC-V ISA with limited hardware implementation details. It features a built-in cache simulator that captures every memory access generated from the processor and collect statistics accordingly.

B.1.1 The ISA simulator is already installed on `tux-1.eee.hku.hk`. If you prefer to run the simulator on your own machine, you need to get the latest RISC-V toolchain source code from:

`https://github.com/ucb-bar/riscv-tools`

To obtain the files for homework 2, perform the following on `tux-1.eee.hku.hk`:

```
tux-1$ cd ~
tux-1$ tar xzvf ~elec3441/elec3441hw2.tar.gz
tux-1$ cd elec3441hw2
tux-1$ export HW2ROOT=$PWD
```

In the downloaded file you will find different benchmark programs located in its own individual directory.

B.1.2 Compiling Benchmark Programs The ISA simulator may execute any valid RISC-V program. If you examine the source code in each subdirectory, you will notice they are no different from any other normal program. Feel free to write your own benchmark if you are curious.

You must set up your environment correctly to make use of the RISC-V tool chain. On `tux-1`, you can use the following command:

```
tux-1$ source ~elec3441/elec3441.bashrc
```

To compile the provided benchmark, you may either perform a `make` command in each director, or make use of the top-level makefile that is provided for you:

```
tux-1$ cd ${HW2ROOT}/benchmarks
tux-1$ make
```

B.1.3 Running Simulator You may now execute your compiled program using the `spike` ISA simulator. Execute the target binary with an L1 instruction cache as follows:

```
tux-1$ cd ${HW2ROOT}/benchmarks/kmean
tux-1$ spike --ic=128:2:64 pk kmean
Veirification passed!
I$ Bytes Read:            132094268
I$ Bytes Written:         0
I$ Read Accesses:         33023567
I$ Write Accesses:        0
I$ Read Misses:           489
I$ Write Misses:          0
I$ Writebacks:            0
I$ Miss Rate:             0.001%
tux-1$
```

In the above line, the last argument `kmean` specifies the RISC-V binary that you are simulating using `spike`. The argument `pk` before that stands for **p**roxy **k**ernel, and it tells `spike` to use the native Linux kernel to handle any system calls.

Finally, the argument `--ic=128:2:64` tells `spike` to simulate an **i**nstruction **c**ache that has

- 128 entries
- 2-way set associative
- 64-byte block

If you multiply the three parameters, you get the capacity of the instruction as $128 \times 2 \times 64 = 16\,\text{KiB}$.

You can also specify the use of a data cache with the argument `--dc=<S>:<W>:<B>` and a unified L2 cache with the argument `--l2=<S>:<W>:<B>`. For example:

```
tux-1$ spike --ic=128:2:64 pk ...
tux-1$ spike --ic=128:2:64 --dc=128:2:64 pk ...
tux-1$ spike --ic=128:2:64 --dc=128:2:64 --l2=1024:4:64 pk ...
```

B.1.4 Cache Evaluation Now, run all the benchmark programs with the following 2 memory hierarchies, one with L2 and the other without L2 cache:

- `--ic=128:1:32 --dc=256:1:32`
- `--ic=128:1:32 --dc=256:1:32 --l2=1024:2:128`

Collect statistics about the memory hierarchies from the output and answer the following questions.

1. For each benchmark, what is the miss rate for L1 I$, L1 D$ and L2$? Which benchmark has the best and which one has the worst cache performance? *Hint:* Consider automating the process, as you will probably need to regenerate a lot of similar statistics in the rest of this homework.
2. What is the cache access time for the two L1 caches according to Table B.1?
3. Based on your above answer, what is the cycle time of the pipeline? Assuming on a cache hit, data should be returned in 1 cycle. Also assume that critical path of all non-memory pipeline stages is 600 ps. In other word, your cycle time is limited by the cache if the cache access time is longer than 600 ps. Otherwise, cache access is not the bottleneck.
4. Calculate the average CPI for the benchmarks *without L2 cache*. You can use the following formula to calculate average CPI, where MP = Miss Penalty, CT = Cycle Time. Use $\text{CPI}_{\text{base}} = 1.2$.

Assume the backside of L1 caches are connected to a single DRAM with 100 ns access time, i.e., accessing the first word from the DRAM takes 100 ns. Each subsequent word takes additional 5 ns.

$$\text{CPI} = \text{CPI}_{\text{base}} + \frac{\#\text{L1\_I\$ misses} + \#\text{L1\_D\$ misses}}{\#\text{instructions}} \times \left\lceil \frac{\text{MP}}{\text{CT}} \right\rceil$$

Note that the number of instruction is the `I$ Read Accesses` number in the output of `spike`.

5. What is the AMAT (in ns) of the L2$ for all benchmarks? Use the following formula to calculate AMAT. (HT=Hit Time, MR=Miss Rate, MP=Miss Penalty) Assume the backside of the L2$ is connected to a DRAM with 100 ns access time. Note that L2 AMAT is the time calculated after L1 access. So HT is the time to access L2 cache after L1, MR is the local L2 miss rate, etc. Assume L2 access time is 10 ns.
$$\text{AMAT}_{L2} = \text{HT}_{L2} + \text{MR}_{L2} \times \text{MP}_{L2}$$

6. Calculate the average CPI for the benchmarks with L2 cache. Use the following formula to calculate CPI, assuming the L2$ is running asynchronously on its own clock domain. Use the same base CPI as above.
$$\text{CPI} = \text{CPI}_{\text{base}} + \frac{\#\text{L1\_I\$ misses} + \#\text{L1\_D\$ misses}}{\#\text{instructions}} \times \left\lceil \frac{\text{AMAT}_{L2}}{\text{CT}} \right\rceil$$

7. Based on your answers above, does the L2$ help with performance?

B.1.5 Design Space Exploration for L1 Data Cache Configuration Find an optimal L1 D$ configuration that maximizes performance of the benchmarks. You can assume L1 I$ is perfect and does not affect processor performance. There is no L2$. Pick one configuration from the following design space:

- Total capacity: up to 2 MiB
- Associativity: $1, 2, 4, 8, 16, 32, 64$
- Cache line size: 4, 8 (words)

Note the following:

- Your cache organization may affect your processor cycle time. Your architecture requires that L1 cache returns data in 1 cycle.
- Use geometric mean of normalized performance of the benchmark programs as the overall performance metric.

Hint: You should write a program or use a spreadsheet program like Excel to help you find the optimal L1 configuration. Remember, you need to compute the geometric mean of speed up of different configurations relative to the baseline.

B.1.6 Optional: Improved Benchmarks Now, given your optimal L1 cache obtained from above, try to implement a faster version of the *slowest* benchmark program than the provided implementation. You may use any modifications to data structure or code, as long as the same calculation is still performed at runtime, i.e., you cannot simply hard code the answer at compile time. Evaluate the improvements you make using the geometric mean of speedups that you achieve over the baseline that you determined in the previous question.

B.1.7 Submission Submit your answer to B.1.4, B.1.5, and the optional part in B.1.6. Make your answer concise, but make sure you have data to support your analysis.

(a) cache line size = 4 words

assoc \ size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB	2MB
1	0.21	0.27	0.32	0.38	0.46	0.54	0.69	0.85	1.14	1.35
2	0.37	0.44	0.50	0.54	0.57	0.64	0.73	1.00	1.24	1.51
4	0.44	0.49	0.52	0.57	0.61	0.69	0.79	1.02	1.31	1.58
8	N/A	0.67	0.70	0.75	0.77	0.83	0.90	1.18	1.61	1.98
16	N/A	N/A	1.05	1.08	1.13	1.17	1.24	1.46	2.43	2.74
32	N/A	N/A	N/A	1.83	1.84	1.91	2.00	2.44	2.85	4.97
64	N/A	N/A	N/A	N/A	3.39	3.67	3.87	4.25	4.72	6.06

(b) cache line size = 8 words

assoc \ size	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB	2MB
1	0.24	0.28	0.30	0.36	0.47	0.54	0.70	0.92	1.15	1.62
2	0.51	0.53	0.54	0.56	0.60	0.65	0.75	0.96	1.27	1.80
4	N/A	0.73	0.74	0.76	0.80	0.83	0.87	1.07	1.44	2.21
8	N/A	N/A	1.13	1.14	1.18	1.21	1.24	1.66	2.24	3.14
16	N/A	N/A	N/A	1.80	1.94	1.97	1.98	2.63	3.34	4.47
32	N/A	N/A	N/A	N/A	2.55	2.37	3.56	3.85	4.65	6.35
64	N/A	N/A	N/A	N/A	N/A	4.50	5.23	5.78	6.47	7.20

Table B.1: Cache access time (in ns) for various cache configurations.