Department of Computer Science and Engineering
Computer Architecture

January Exam
January, 17th 2020

ATENTTION:

- Please read carefully all the questions before starting.
- Books, notes and calculators are not allowed.
- Mobile phones must be disconnected during the test (disconnected, not muted).
- Only answers in ink will be corrected. Please, do not use pencil.
- **Time limit is 75 minutes.**
- **Give every exercise in separate sheets. If you do not answer an exercise, please give a blank sheet with the exercise number.**

FULL NAME:

NIA:

Exercise 1 [1.5 points]:

A given processor has an instruction set where all instructions take exactly 20 cycles. In this computer an application has the following distribution of instructions used:

- Arithmetic instructions: 40% of total number of instructions.
- Branch instructions: 20% of total number of instructions.
- Load/store instructions: 35% of total number of instructions.
- Other instructions: 5% of total number of instructions.

By redesigning the processor one (and only one) of the following choices may be applied:

a) Improve arithmetic instructions with an average speedup of 4.
b) Improve branch instructions with an average speedup of 4.5.
c) Improve load/store instruction with an average speedup of 2.5
d) Improve other instruction with an average speedup of 10.

Answer the following questions:

1.1 [1 point]: Which of the four choices would you recommend? Justify your answer.

1.2 [0.5 points]: Assume you can only apply the load/store improvement. What should be the speedup for load/store instructions so that the global speedup of your application is 1.5.

SOLUTION:

1.1: For each choice we may apply Amdahl's Law:

S = 1 / (0.6+0.4/4) = 1 / (0.6+0.1) = 1/0.7 = 1.429

S = 1 / (0.8 +0.2/4.5) = 1/ (0.8 + 0,044) = 1/0.844 = 1.184

S = 1 / (0.65 + 0.35/2.5) = 1 / (0.65 + 0,14) = 1 / 0,79 = 1.266

S = 1 / (0.95 + 0.05/10) = 1 / (0.95 + 0.005) = 1 / 0.955 = 1.047

Option a seems to be the best.

1.2:

S = 1 / [(1-F)+F/Sm]

1.5 = 1 / (0.65 + 0.35/Sm)

1.5 (0.65 + 0.35/Sm) = 1

0.975 + 0.525/Sm = 1

0.525/Sm = 1-0.975

Sm = 0.525/0,025 = 21

Department of Computer Science and Engineering
Computer Architecture

January Exam
January, 17th 2020

Exercise 2 [3 points]:

Given the following code fragment:

```
loop:   lw $t1, 0($t0)          #I1
        lw $t2, 1000($t0)       #I2
        mul $t2, $t1, $t2       #I3
        lw $t1, 2000($t0)       #I4
        add $t2, $t1, $t2       #I5
        sw $t2, 3000($t0)       #I6
        addi $t0, $t0, 4        #I7
        addi $t3, $t3, -4       #I8
        bnz $t3, loop           #I9
```

This code fragment is executed on a machine where the processor has a pipeline with enough fetch and decode bandwidth to start the execution of one instruction in each clock cycle.

In this processor there are stalls due to data dependencies. In the case of data dependency, the start of an instruction incurs an additional latency that will depend on each instruction:

- lw: Loads a value from memory. Requires 2 additional latency cycles before its result is available.
- mul: Multiplies two values. Requires 6 additional latency cycles before its result is available.
- add: Adds two values. Requires 4 additional latency cycles before its result is available.

The rest of the instructions do not require additional latency cycles.

2.1 [0.5 points]: Identify all RAW dependencies.

2.2 [1 point]: Identify all the stalls that occur for the first iteration of the loop. Compute the number of cycles per iteration.

2.3 [1.5 point]: Perform the loop unrolling (degree 2) when scheduling is also applied. Calculate the number of cycles per iteration.

SOLUTION:

2.1 The following RAW dependencies are identified:

- \$t1: I1→I3
- \$t2: I2→I3
- \$t2: I3→I5
- \$t1: I4→I5
- \$t2: I5→I6

2.2:

```
loop:   lw $t1, 0($t0)          #I1
        lw $t2, 1000($t0)       #I2
        STALL x 2
        mul $t2, $t1, $t2       #I3
        lw $t1, 2000($t0)       #I4
        STALL x 5
        add $t2, $t1, $t2       #I5
        STALL x 4
        sw $t2, 3000($t0)       #I6
        addi $t0, $t0, 4        #I7
        addi $t3, $t3, -4       #I8
        bnz $t3, loop           #I9
```

A total of 20 cycles per iteration are required

2.3:

```
loop:   lw $t1, 0($t0)
        lw $t2, 1000($t0)
        lw $t4, 4($t0)
        lw $t5, 1004($t0)
        mul $t2, $t1, $t2
        STALL
        mul $t5, $t4, $t5
        lw $t1, 2000($t0)
        lw $t4, 2004($t0)
        STALL
        STALL
        add $t2, $t1, $t2
        STALL
        add $t5, $t4, $t5
        STALL
        STALL
        sw $t2, 3000($t0)
        STALL
        sw $t5, 3004($t0)
        addi $t0, $t0, 8
        addi $t3, $t3, -8
        bnz $t3, loop
```

A total of 22 cycles per two iterations are required. That is 11 cycles per iteration.

Exercise 3 [2 points]:

A given application uses a *sample* data structure to represent samples obtained by a sensor:

```
struct sample {
  long id;
  int x, y, z;
  double value;
  int valid;
};
```

The program uses a function average to compute the average value of all samples:

```
double average(sample v[], int n) {
  for (int i=0;i<n;++i) {
    if (v[i].id>0) {
      v[i].valid = 1;
    }
    else {
      v[i].valid = 0;
    }
  }
  float r = 0.0;
  for (int i=0;i<n;++i) {
    if (v[i].valid) r+= v[i].value;
    else v[i].value = 0.0;
  }
  return r/n;
}
```

In the target machine the following representation sizes are used by your compiler:

- long: 8 bytes.
- int: 4 bytes.
- double: 8 bytes.

This function is invoked with an array of size 1024 in a system with a L1 data cache of 32 KB which is 8 ways associative. The block size is 128 bytes. The array v is aligned to a memory address multiple of 128. Assume that local variables `i` and `r` are mapped to registers.

You are asked to determine:
3.1 [0.25 points]: The number of cache misses.

3.2 [0.25 points]: The average miss rate.

3.3 [0.75 points]: Propose an alternative implementation of function average that uses loop merging. Compute the average miss rate.

3.4 [0.75 points]: Propose an alternative implementation of function average that keeps coordinates x, y, and z in a separate array so that there are two parallel arrays: coordinates (containing x, y, and z) and data (containing id, value, and valid). Compute the average miss rate.

SOLUTION:

3.1: In the target machine the size of one sample requires 32 bytes (4 samples per cache line). The array has 1024 entries of 32 bytes each. Requiring a total of 32 KB fitting completely in the cache.

There will be one miss for every 4 entries in the array for the first loop. The second loop does not cause any miss. Consequently, the number of misses is 1024/4 = 256 misses.

3.2: To compute the miss rate, the number of memory references needs to be obtained:

- First loop:
  - Read from v[i].id.
  - Write to v[i].valid.
- Second loop:
  - Read from v[i].valid.
  - Read or write v[i].value.

The total number of memory references is 1024 * 2 + 1024 * 2 = 4096.

The miss rate is:

m = 256 / 4096 = 1/16 = 0.0625

3.3

```
double average(sample v[], int n) {
  float r = 0.0;
  for (int i=0;i<n;++i) {
    if (v[i].id>0) {
      v[i].valid = 1;
      r+= v[i].value;
    }
    else {
      v[i].valid = 0;
      v[i].value = 0.0;
    }
  }
  return r/n;
}
```

Now the number of misses is the same, but the number of memory references is 1024 * 3 = 3072.

The new miss rate is:

m=256/3072 = 1/12 = 0.083

3.3

```
struct coordinates {
  int x, y, z;
};
struct data {
  long id;
  double value;
  int valid;
};

double average(data v[], int n) {
  for (int i=0;i<n;++i) {
    if (v[i].id>0) {
      v[i].valid = 1;
    }
    else {
      v[i].valid = 0;
    }
  }
  float r = 0.0;
  for (int i=0;i<n;++i) {
    if (v[i].valid) r+= v[i].value;
    else v[i].value = 0.0;
  }
  return r/n;
}
```

The new data needs 20 bytes per entry. The whole array needs 20*1024 = 20 KB.

The total number of cache lines involved are 160 lines. This leads to 160 misses.

The miss rate is now:

m = 160 / 4096 = 5/128 = 0.039

Exercise 4 [1.5 point]: Given the following code.

```
#include <thread>
#include <atomic>
#include <iostream>

std::atomic<int> x=0, y=0, z=0;

void f() {
  x.store(1);
  y.store(2);
}

void g() {
  while (y.load()==0) {}
  z.store(3);
}


void h() {
  while (z.load()==0) {}
  std::cout << x.load() << "\n";
}

int main() {
  std::thread tf{f};
  std::thread tg{g};
  std::thread th{h};
  tf.join();
  tg.join();
  th.join();
}
```

4.1 [0.25 points]: Which is the memory consistency model for this program as written?
4.2 [0.5 points]: What are the possible values to be printed with that memory consistency model?
4.3 [0.75 points]: Write the most relaxed program that you can design that is semantically equivalent to the given program.

SOLUTION:
4.1: The default model is sequential consistency.
4.2: The only possible value is 3.
4.3:

```
#include <thread>
#include <atomic>
#include <iostream>

std::atomic<int> x=0, y=0, z=0;

void f() {
  x.store(1, std::memory_order_relaxed);
  y.store(2, std::memory_order_release);
}
```

```
void g() {
  while (y.load(std::memory_order_acquire)==0) {}
  z.store(3, std::memory_order_release);
}


void h() {
  while (z.load(std::memory_order_acquire)==0) {}
  std::cout << x.load(std::memory_order_relaxed) << “\n”;
}

int main() {
  std::thread tf{f};
  std::thread tg{g};
  std::thread th{h};
  tf.join();
  tg.join();
  th.join();
}
```