Exercises for the Lecture: "Architecture and Programming Models for GPUs and Coprocessors" Exercise Sheet № 2

Priv.-Doz. Dr. rer. nat. Stefan Zellmann

WS 2021/22

2 Parallel Architectures

Ex. 2.1

a.)
We are given a hypothetical CPU with two arithmetic logical units (ALU0 and ALU1), as well as two memory management units (MMU0 and MMU1). The instruction set includes the instructions `LD`, `ST`, `ADD`, and `MUL`. Those require 4, 4, 1, and 2 clock cycles, respectively.

The `LD` and `ST` instructions occupy an ALU during the first clock cycle and an MMU during the remaining ones. In addition, the `ST` can only be scheduled on MMU1. The `ADD` and `MUL` instructions require ALU only. An instruction that was scheduled on a specific resource cannot be rescheduled on another resource of the same type in the following clock cycle (e.g., we cannot schedule the first clock cycle MMU on MMU0 and the second clock cycle MMU on MMU1).

Based on those assumptions, specify all possible reservation tables for the four instructions.

b.)
We are given a hypothetical CPU with one MMU and one ALU. Memory access instructions occupy the MMU for two clock cycles. Arithmetic instructions occupy the ALU for one clock cycle. For simplicity, we can assume that during each loop iteration all required data items are loaded from memory into registers with a single `LD` instruction.
Specify the reservation table for the first iteration of the following loop.

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

Determine and name the forbidden latencies. What is the earliest allowable time slot to schedule the second loop iteration? Based on that, also specify the reservation table for the first two iterations. Determine and name the forbidden latencies. What is the earliest allowable time slot to schedule another loop iteration?

Ex. 2.2

We are given a hypothetical CPU with the following instruction set:

Instruction	Description	Latency
`ADD $R1` C	Add the constant value C to the content of register `$R1`	4 ns
`ADD $R1 $R2`	Add register contents `$R1` and `$R2`, store the result in `$R1`	4 ns
`MUL $R1 $R2`	Multiply register content `$R1` and `$R2`, store the result in `$R1`	4 ns
`LD $R1` $[S]$	Load the content at memory item $[S]$ into register `$R1`	8 ns
`ST` $[S]$ `$R1`	Store the content of register `$R1` in memory item $[S]$	8 ns
`JNZ $R1 label1`	If content of register `$R1` $\neq 0$, jump to label `label1`	1 ns

and the following program:

```
LD $R1 [0xFF-0]
LD $R2 [0xFF-1]
LOOP:
ADD $R1 -1
LD $R3 [0xC0-$R1]
ADD $R2 $R3
LD $R4 [0x80-$R1]
MUL $R2 $R4
MUL $R2 $R2
JNZ $R1 LOOP
ST 0xFF $R2
```

The content of registers `$R1`, `$R2`, `$R3`, and `$R4` are initialized to 0 and the initial value stored at memory address `0xFF` has the constant value 2.

a.)
Draw the data dependency graph for the program's loop.

b.)
Unroll the loop. Schedule the resulting loop iteration on a hypothetical CPU with a single ALU that executes all the arithmetic instructions, and an MMU that executes all the memory access instructions.

Ex. 2.3

a.)
Compile the file `queue.cpp` that accompanies this exercise sheet with a C++11 compiler. The `gcc` would look as follows.

```
g++ queue.cpp -std=c++11 -o queue
```

The program `queue` can be executed in a terminal but will crash with an error message as it contains programming errors. We will fix those in the following.

b.)
The template class `sync_queue` is simultaneously used as a message queue by multiple threads. The internal data structure for that is a `std::deque<T>` from the standard template library (STL). The `std::deque<T>` template class is not thread-safe.

The interface of the queue is comprised of two functions: `push_back()` to push elements to the back of the queue and `pop_front()` to retrieve the frontmost element and remove it from the queue. The `main()` function creates several threads to implement a unit test for the class.

Extend the class so that it becomes thread-safe to avoid the *data races*:

- `pop_front()` should only be executed when there are messages in the queue. Otherwise, the function should block. You can use a semaphore for this (see the file `semaphore.h`). As long as the queue is empty, we wait on the semaphore. When the queue gets filled with messages, we notify the waiting threads to resume their current operations (`semaphore.notify()`).
- Accesses to the underlying `std::deque` object are synchronized using a critical section. This can be implemented with the STL classes `std::mutex` and `std::unique_lock`.

On success, the class `sync_queue` should allow the threads to exchange messages without data races.

The exercise sheet will be discussed on Nov 24, 2021.