ECE 153A HW 2

Oviya Seeniraj

Question 1

Part A

Branch delay slots allow the instruction immediately following a branch to be executed before the branch takes effect, often with conditional code. This effectively "hides" the latency of the branch and improves code throughput by avoiding pipeline stalls. In the MicroBlaze architecture, if the branch is taken, the next instruction in the pipeline (already fetched) is executed, which reduces the time wasted on branch processing.

Part B

Hardware divide instructions take many cycles to execute when compared to most other instructions, around 30 vs 4 cycles. If an interrupt occurs during the execution of a hardware divide, the processor cannot service the interrupt until the divide instruction completes. This introduces latency, which can be a problem in systems where quick response to interrupts is important.

Part C

With an initial stack pointer at 0x0b0, the available stack space grows down to 0x000, which gives us 176 bytes to work with. Assuming 4 bytes for the return address, 7 * 4 bytes for the parameters (assuming all parameters are saved onto the stack), and 4 bytes for the link register, 36 bytes are used per call. Each recursive call will consume stack space for return addresses and parameters and add up rather than clearing the stack as with normal calls. This gives us 176/36 = 4.88 ~ **4 recursive calls** as fractional calls are not possible. Any more than 4 recursive calls will result in stack overflow, where the stack pointer moves into unallocated memory or memory reserved for other purposes, causing unpredictable behavior.

Part D

- Non-static arrays are typically allocated on the **stack** if they are local variables. The storage for the 6x6 array of long ints will be pushed onto the stack when the function is called.
- Static arrays are allocated in **initialized memory** (data segment). This memory persists between function calls, so the array won't be reallocated on the stack each time the function is entered. However, it only exists in the context of the program and will be automatically deleted when the program ends, unlike heap memory.

Part E

The transition from BRAM (Block RAM) to DRAM increases access time. BRAM is tightly coupled to the MicroBlaze processor and has low latency, whereas DRAM involves longer access times and potential contention with other system components where DRAM access is delayed due to other processes. If interrupt handlers or other critical code rely on fast memory access, using DRAM could lead to timing issues, causing the described missed interrupts or delays in servicing them.

Question 2

To prevent the compiler from optimizing away the check of var, we can declare var as volatile. This tells the compiler that the value of var can change at any time, outside the normal program flow (e.g., due to hardware changes), and that it should not optimize the code based on assumptions about its value.

We can rewrite the code as:

```
static volatile int var;

void bar(void){
    var = 0;
    while (var != 255);
}
```

This ensures the compiler checks the value of var in each iteration of the loop, instead of assuming it won't change.

Question 3

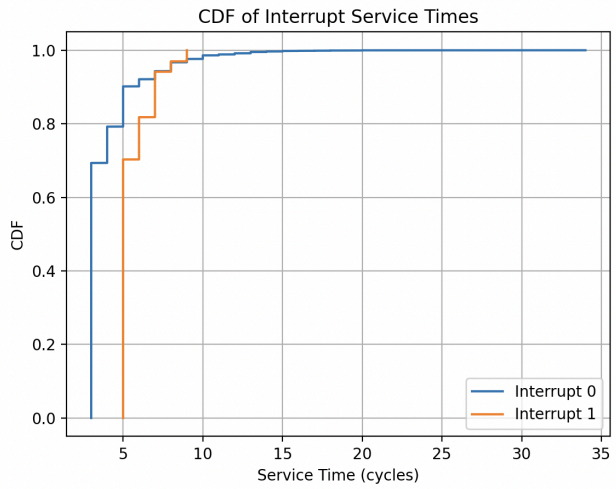
Part A

Output from main.c:

```
Number of missed interrupt 0's: 7836
Max latency for interrupt 0: 34
```

```
Number of missed interrupt 1's: 443
Max latency for interrupt 1: 9
```

Statistical analysis and graphs from my Python code of an *individual* trial:

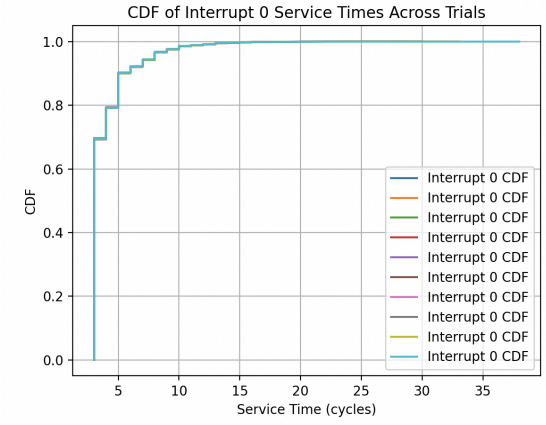


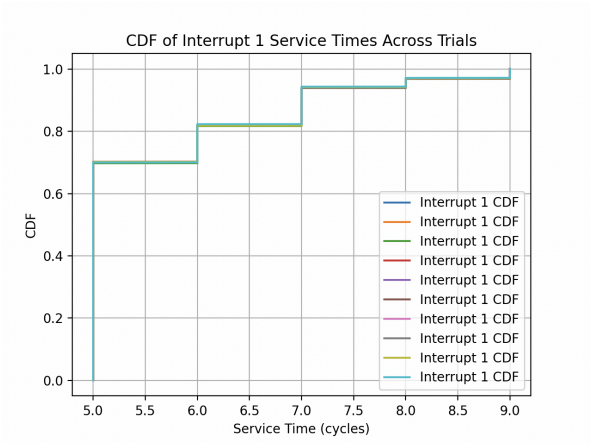

```
Interrupt 0 – Mean Service Time: 3.85 cycles
Interrupt 0 – Max Service Time: 34 cycles
```

```
Interrupt 1 – Mean Service Time: 5.57 cycles
Interrupt 1 – Max Service Time: 9 cycles
```

99% Confidence Interval for the Individual Latency of Interrupt 0: **(3.8315, 3.8625)**
99% Confidence Interval for the Individual Latency of Interrupt 1: **(5.5467, 5.5768)**

Statistical analysis and graphs from my Python code of 10 trials:





Trial	Interrupt 0 Mean	Interrupt 1 Mean	Interrupt 0 Max	Interrupt 1 Max
1	3.851606	5.571048	30	9
2	3.848651	5.572735	30	9
3	3.849829	5.578477	29	9
4	3.853324	5.570305	33	9
5	3.852282	5.567372	28	9
6	3.855888	5.570538	30	9
7	3.849559	5.567328	28	9
8	3.861846	5.570364	28	9
9	3.858920	5.567775	32	9
10	3.847021	5.561756	38	9

99% Confidence Interval for the Mean Latency of Interrupt 0: **(3.8480, 3.8577)**

99% Confidence Interval for the Mean Latency of Interrupt 1: **(5.5653, 5.5742)**

Part B

When the probability of Interrupt0 increases to 0.2, the number of interrupt requests for this type doubles. Theoretically, we can predict that in turn, Interrupt0 will be serviced more frequently, which can lead to more missed interrupts for both Interrupt0 (if it arrives before the previous one is handled) and Interrupt1 (due to lower priority). The worst-case latency for Interrupt1 will increase, as it has to wait longer while multiple Interrupt0s are serviced.

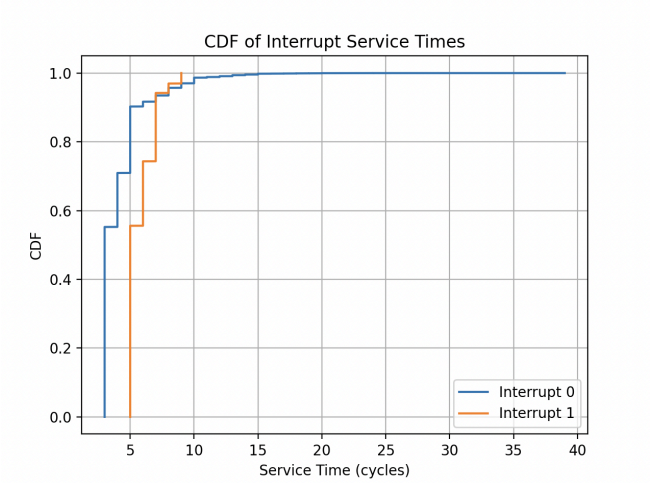
Output from main.c:

```
Number of missed interrupt 0's: 36173
Max latency for interrupt 0: 39
```

```
Number of missed interrupt 1's: 645
Max latency for interrupt 1: 9
```

Our predictions are confirmed: there are more missed interrupts (5x for interrupt 0, ~1.5x for interrupt 1). Interrupt0's worst-case latency increases, which makes sense, because more Interrupt0's likely occur while a previous Interrupt0 is being serviced. However, Interrupt1's max latency does not increase, since the priority is lower and its probability is too low for the latency to be seen.

Statistical analysis and graphs for *individual* trials:

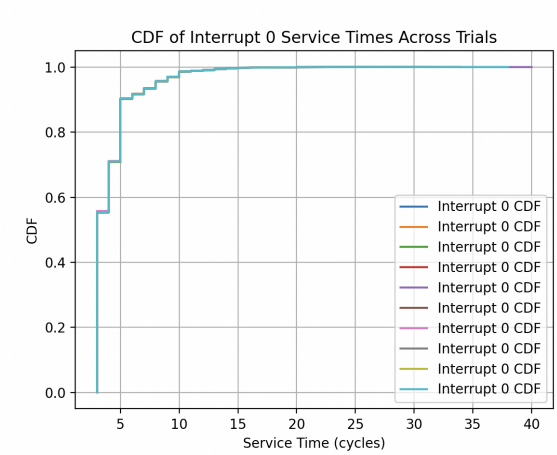


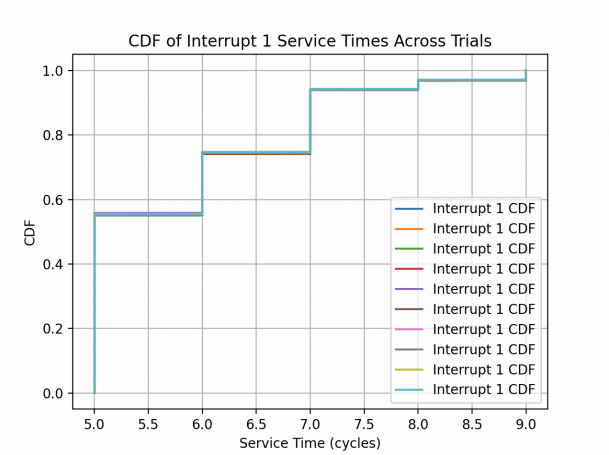

```
Interrupt 0 – Mean Service Time: 4.11 cycles
Interrupt 0 – Max Service Time: 39 cycles
```

```
Interrupt 1 – Mean Service Time: 5.79 cycles
Interrupt 1 – Max Service Time: 9 cycles
```

99% Confidence Interval for the Individual Latency of Interrupt 0: **(4.0947, 4.1189)**
99% Confidence Interval for the Individual Latency of Interrupt 1: **(5.7712, 5.8026)**

Statistical analysis and graphs from my Python code of *10 trials*:





Trial	Interrupt 0 Mean	Interrupt 1 Mean	Interrupt 0 Max	Interrupt 1 Max
1	4.112524	5.795165	32	9
2	4.109389	5.794472	30	9
3	4.108349	5.797255	31	9
4	4.106993	5.797348	34	9
5	4.107254	5.783723	40	9
6	4.114240	5.796500	30	9
7	4.100512	5.790783	30	9
8	4.117481	5.787938	30	9
9	4.111142	5.786094	34	9
10	4.106812	5.786884	38	9

99% Confidence Interval for the Mean Latency of Interrupt 0 across 10 trials: **(4.1046, 4.1143)**
99% Confidence Interval for the Mean Latency of Interrupt 1 across 10 trials: **(5.7863, 5.7969)**

One thing to note is that the confidence interval for Interval 0 increases, as expected due to higher latency, but for Interrupt 1, it doesn't change at all, due to its low probability of occurrence and mainly its lower priority.