ECE 153a - Homework 2

Solutions

1. Microblaze Processor:

   (a) The branch delay slot improves code throughput because it allows the instruction immediately after the branch instruction to execute when otherwise it would have been flushed from the execution pipeline. (If there is no branch – it is the normal instruction to run – if there is a branch, it will be decoded and run anyway.) Since the instruction is not flushed, one more instruction per time cycle is executing, therefore increasing throughput.

   (b) Support for hardware divide instructions may be problematic if interrupt latency is critical to the design because hardware divide instructions typically have high latency. If a divide is executing and an interrupt occurs, many clock cycles will pass before the interrupt can be handled because the interrupt will wait for the current instruction to complete. Thus, interrupt latency increases significantly.

   (c) With a series of nested function calls that add values to the stack, the stack can accumulate data so quickly without popping off enough that it overflows into program or data portions of the memory. The MicroBlaze stack convention dictates that the amount of data on the stack will grow linearly with each successive function call. The growth is proportional to the number of passed parameters beyond the first and the return value, which are always passed, along with the new PC return address. Assume each parameter or pointer occupy 4 bytes, thus each call push 2-3 data into the stack. Thus, total allowed successive calls are about 14-22. 0xB0 = 176/(2 parameters*4 bytes) = 22 functions.

   (d) A long int declared in the parent will be allocated to the stack, and declaring it as static will allocate it to initialized memory (Most commonly, BRAM for MicroBlaze, could be SDRAM depending on configuration and code size).

   (e) The problem here is the increased latency in accessing DRAM instead of BRAM. CPU access to the DRAM takes many more clock cycles than access to the BRAM. In particular, the longer latency can substantive program waiting periods. In a normal processor, such behavior could be solved by adding a cache to the memory system so that frequently accessed parameters avoid the longer latency. However, this solution rarely solves the worst case latency issue faced by an embedded processor. For high-speed repetitive interrupts such as a sampled audio interface, one interrupt may not be able to complete before it is interrupted again – breaking the run to completion paradigm.

2. If the 'volatile' storage class keyword is used in a declaration, you are telling the compiler that one cannot predict when the value of the specified variable will be changed. Thus, the compiler cannot remove the comparison from the loop – despite it having no software mechanism to change the value. Volatile is provided for this reason in the language- it has nothing to do with 'const' which allows statics type checking of program l-value writing. In fact: extern const volatile int is a valid and meaningful storage class. 'const' means that the program cannot write the value. Thus, the code should be as follows:

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

3. (a) The results for part (a) are shown below:

Table 1

(a) Interrupt0

#	Mean latency	Max latency	Missed
1	3.814	19	62
2	3.79	20	86
3	3.85	15	69
4	3.796	17	59
5	3.877	20	74
6	3.843	20	94
7	3.789	17	75
8	3.842	15	82
9	3.836	17	64
10	3.831	19	72

(b) Interrupt1

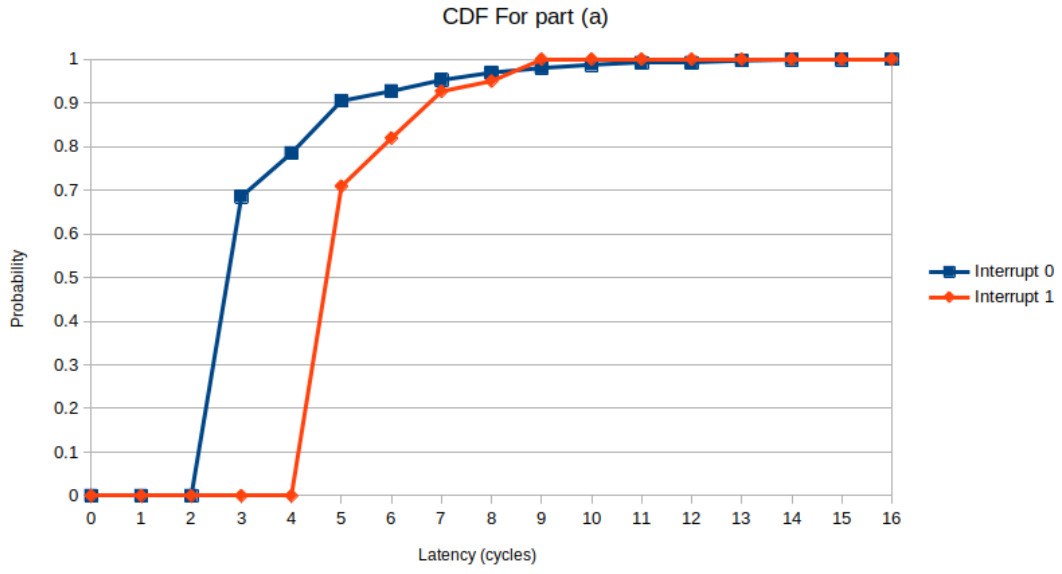
#	Mean latency	Max latency	Missed
1	5.563	9	2
2	5.575	9	8
3	5.502	9	5
4	5.61	9	5
5	5.573	9	4
6	5.509	9	7
7	5.528	9	7
8	5.597	9	5
9	5.562	9	6
10	5.58	9	4

The 99% confidence intervals of mean latencies are:
Interrupt0: 3.75 - 3.9 clock cycles
Interrupt1: 5.46 - 5.65 clock cycles

For an 11th run to gather raw data, the 99% confidence interval of the individual service times are:
Interrupt 0: 3-14 cycles
Interrupt 1: 5-9 cycles

The following CDF for the interrupt service times were obtained:

Figure 1



(b) The results for part (b) are shown below:

Table 2

(a) Interrupt0

#	Mean latency	Max latency	Missed
1	4.042	15	342
2	4.198	19	377
3	4.066	18	340
4	4.07	16	343
5	4.117	24	391
6	4.133	25	323
7	4.129	27	370
8	4.056	20	328
9	4.106	25	375
10	4.106	20	364

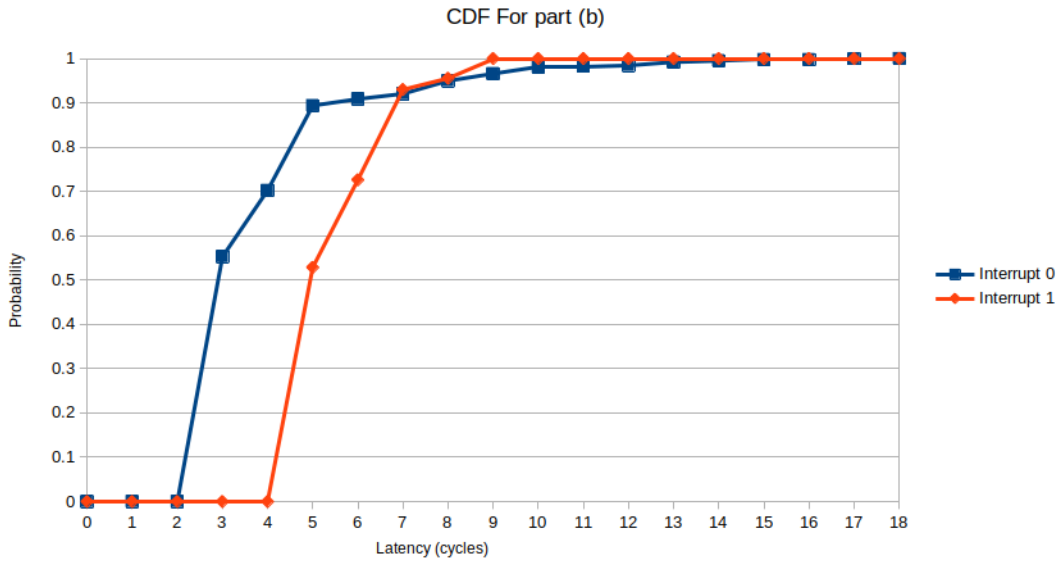
(b) Interrupt1

#	Mean latency	Max latency	Missed
1	5.72	9	8
2	5.89	9	11
3	5.688	9	7
4	5.754	9	7
5	5.688	9	9
6	5.797	9	4
7	5.819	9	5
8	5.746	9	9
9	5.793	9	4
10	5.846	9	10

The 99% confidence intervals of mean latencies are:
Interrupt0: 3.98 - 4.22 clock cycles
Interrupt1: 5.6 - 5.95 clock cycles
Doubling the probability of occurrence of Interrupt0 leads to a slight increase in the mean latencies for both Interrupt0 and Interrupt1. This is expected because if Interrupt0 occurs more often, interrupts will be "pending" more often, which increases the mean latency. Worst-case latency for Interrupt0 also increases slightly because of the same reason, but worst-case latency of Interrupt1 remains the same. Worst-case latency of Interrupt1 is unaffected because Interrupt1 has a higher priority than Interrupt0 and more frequent occurrences of a lower priority interrupt doesn't impact the worst-case latency of a higher priority interrupt. Interestingly, the number of missed Interrupt0's increases by 3-6 times, which is expected due to more frequent occurrences of Interrupt0 (and increased probability of Interrupt0 being "pending" already when Interrupt0 is fired again). The number of missed Interrupt1's increases only slightly, because of its priority being higher than Interrupt0's priority.

For an 11th run to gather raw data, the 99% confidence interval of the individual service times are:
Interrupt 0: 3-13 cycles
Interrupt 1: 5-9 cycles

Figure 2


CDF For part (b)
Probability
Latency (cycles)
Interrupt 0
Interrupt 1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18