Assignment 9 - out of 30 marks

Objectives:

- Pipelined Execution Analysis – Uniform and non-uniform stage lengths
- Benchmarking “branch reduction’ optimization
- Microbenchmarking array search versus linked list search
- “Loop unrolling” optimization

Submission:

- Submit your document called Assignment_9.pdf, which must include your answers to all of the questions in Assignment 9.
  - Add your full name and student number at the top of the first page of your document Assignment_9.pdf.
- When creating your assignment, first include the question itself and its number then include your answer, keeping the questions in its original numerical order.
- If you hand-write your answers (as opposed to using a computer application to write them): When putting your assignment together, do not take photos (no .jpg) of your assignment sheets! Scan them instead! Better quality -> easier to read -> easier to mark!
- Submit your assignment Assignment_9.pdf electronically on CourSys.

Due:

- Thursday, April 9 at 3pm.
- Late assignments will receive a grade of 0, but they will be marked in order to provide the student with feedback.

Marking scheme:

This assignment will be marked as follows:

- All questions will be marked for correctness.

The amount of marks for each question is indicated as part of the question.

A solution will be posted after the due date.
1. [5 marks] Pipelined Execution Analysis – Uniform and non-uniform stage lengths

Part 1 - Following on from our Assn 8 Question 4 a. and b. ...

Imagine we built a processor. The propagation delay of its entire combinational logic circuit is 600 ps and the propagation delay to load the clocked register we used is 30 ps.

c. What is the minimum clock cycle time (in picoseconds), the latency (in picoseconds) and throughput (in GIPS) of our processor if it performs pipelined execution using 8 equal stages?

Part 2 - Now, imagine we built a second processor. For the purpose of this question, we shall call this processor the **original processor**.

The propagation delay of its entire combinational logic circuit is 600 ps and the propagation delay to load the clocked register we used is 30 ps.

However, the entire combinational logic circuit of this processor has been divided into 6 stages:

- stage F with a propagation delay of 100 ps,
- stage D with a propagation delay of 90 ps,
- stage E with a propagation delay of 200 ps,
- stage M with a propagation delay of 90 ps,
- stage W with a propagation delay of 70 ps, and
- stage P with a propagation delay of 50 ps.

So far, no clocked registers have been placed between any of these 6 stages.

a. If we were to place one clocked register between a pair of adjacent stages of the original processor in order to form a 2-stage pipeline, where would we place this clocked register? Compute the minimum clock cycle time and the maximum CPU throughput of this processor.

b. If we were to place two clocked registers between the stages of the original processor in order to form a 3-stage pipeline processor, where would we place these two clocked registers? Compute the minimum clock cycle time and the maximum CPU throughput of this processor.

c. If we were to place more and more clocked registers between the stages of the original processor in order to form greater staged-pipeline processors, which stage would become a limiting factor on the minimum clock cycle time?
Hint: Form a 4-stage pipeline processor then a 5-stage pipeline processor in order to answer this question.

d. If we were to fully pipeline our original processor (create a 6-stage pipeline processor), what would its minimum clock cycle time and its maximum CPU throughput be?

2. [10 marks] Benchmarking “branch reduction” optimization (using the benchmarking tool introduced in Lab 5)

In this question, we shall optimize the performance of the linear search algorithm, which as its name implied, has a time complexity of $O(n)$.

Download Assn9-files.zip and extract its files. Have a look at the files in the Q2 folder. We see that main.c benchmarks a pair of linear searches on a randomized array. The first of these linear search functions is found at the end of main.c. As we can see, it is a very straightforward implementation of the linear search algorithm. In this question, we are asked to implement an alternative linear search and to demonstrate that it performs better (faster) than the first implementation.

a. Even though this second implementation of the linear search will also have a time complexity of $O(n)$, it can be optimized to reduce the cost of each loop. In other words, we are expecting this second implementation of the linear search to produce a smaller CPE (clock cycles – cost - per element).

A good first attempt at optimizing this function is to replace its expensive operations: the comparisons. A comparison in our C code will generate a corresponding branch in the assembly code. And as we know, a branch has the potential to be mispredicted by the CPU (i.e., the CPU’s instruction pipeline). The standard algorithm does two comparisons per loop: the comparison between $A[i]$ and target and the comparison between $i$ and $n$. The number of comparisons can be roughly cut in half using the following algorithm (expressed in pseudocode), which makes use of target as a sentinel put in the last position of the array:

```c
search(A[n], target)
    if n <= 0 then return -1

tmp <- A[n-1]
A[n-1] <- target

i <- 0
while A[i] != target do
    i <- i+1
```
A[n-1] <- tmp

if i < n-1 then return i
else if A[n-1] == target then return n-1
else return -1

Code this algorithm in lsearch_2.c.

Submit lsearch_2.c electronically via CourSys (make sure it is the version that compiles and executes successfully) and add a hardcopy of the same version in this assignment.

b. Benchmark lsearch_1() and lsearch_2() for \( N = \{5000000, 10000000, 15000000, 20000000\} \), and \( NTESTS = 400 \). Collect three samples for each \( N \), and present your raw data and the average times in a table format and plot the average times on a graph (with labels) as we did in Lab 5.

What conclusion do we draw from our results?

Include all datasets and graph in this assignment.

c. Open lsearch_2.s and have a look at the assembly code gcc has created on our behalf. Then do the following in the file lsearch_2.s:

- Above the line “lsearch_2:”, ...
  - add the function prototype as a comment
  - add the list of parameter-register mapping as a comment
- Remove all the directives that can be removed,
- Replace the labels used by the jump instructions with more descriptive labels, and
- Comment the assembly code so that it would help the reader understand the code. In the comments, make sure you indicate which variable is held in which register (as well as its value).

Include lsearch_2.s in this assignment. You do not have to submit it to CourSys.

d. Because compiler optimization was turned on (-O2), gcc has made some adjustments to our original algorithm (original algorithm given above). Figure out what these adjustments to our original algorithm are and write this algorithm with these new adjustments, in pseudocode, below.
3. [8 marks] Microbenchmarking array search versus linked list search (using the microbenchmarking tool introduced in Lab 6)

We will be able to fully answer this question after having done our Lab 6 on Monday April 6.

In this question, we shall investigate the performance of the *linear search* algorithm as it searches an array and a linked list. But as we saw in the previous question, not all linear time algorithms are created equal, so to identify the most efficient algorithm, we shall measure the clock cycles (cost) per element (CPE) using microbenchmarking.

a. If we have not yet done so, let’s download Assn9-files.zip and extract its files. Have a look at the files in the Q3 folder. The main.c sets up two data structures — an array int A[N] and a linked list List *L — with the values 0 . . . N – 1 in a random order. The microbenchmarking code follows.

We can choose between lsearch() for the array by entering 1 at the command line (./x 1) or LLsearch() for the linked list by entering 0 at the command line (./x 0).

The code for these linear search functions is in LL.c. Have a look. Compile the code and open LL.s. Comparing the main loops of the two functions:

```
LLsearch:
   .L42:
   cmpl %esi, (%rdx)
   je .L41
   .L35:
   movq 8(%rdx), %rdx
   addl $1, %eax
   testq %rdx, %rdx
   jne .L42
   .L57:
   addq $1, %rcx
LLsearch:
   ... 

lsearch:
   .L57:
   addq $1, %rcx
lsearch:
   ... 
```

it looks like there are the same number of instructions per element on both sides, but the cost of LLsearch() will be higher because of the von Neumann bottleneck, i.e., there are two memory references per element for LLsearch(), but only one memory reference per element for lsearch(). To even out the situation, let’s add a memory reference instruction to lsearch as follows:

```
lsearch:
   ... 
   .L57:
   addq $1, %rcx
```

Add this line and recompile -> movl -8(%rdi), %r8d
cmpl %edx, -4(%rdi,%rcx,4)
je .L54
.L56:
cmpq %rsi, %rcx
movl %ecx, %eax
jne .L57
...

Re-build the code and microbenchmark each function for $N = \{150, 200, 250, 300\}$. For each, run the code 11 times and throw out the shortest 3 and longest 3 times. Let’s tabulate our results, one quintet of measurements per row. Compute the average of each set of 5 times. Then, plot both data sets on a graph (with labels) and draw a straight line through each set as we did in Lab 6.

Compute the slope of each line. The slope = the clock cycles (cost) per element (CPE) for linearly searching each data structure (the array and the linked list).

What conclusion do we draw from our results?

Submit all datasets and graph with this assignment.

b. The main reason for the difference in the CPE is a data dependency in the linked list search that isn’t present in the array search. Explain the sequence of instructions that causes the delay. Feel free to include a diagram as part of your explanation.

4. [7 marks] “Loop unrolling” optimization

a. In a file called unrolling.c, write a version of the inner product function described in Homework Problem 5.13 (at page 570 in our textbook) that uses 6 x 1 loop unrolling and call this function inner6x1unrolling(...).

b. In the same file, write a second version of the inner product function described in Homework Problem 5.13 that uses 6 x 6 loop unrolling and call this function inner6x6unrolling(...).

Add a header comment block and comments to your functions and submit unrolling.c electronically via CourSys as well as adding a hardcopy of this file in this assignment.