CS2002 Exam – Student ID: 200007413

I hereby declare that the attached piece of written work is my own work and that I have not reproduced, without acknowledgement, the work of another.

1.

(a)

The key advantage CPU's obtain from pipelining is that multiple instructions split into these stages can be run simultaneously, where different processes, one at each stage at a time, can be computed. This means you can complete multiple instructions within one clock cycle, as you are working on different parts of different instructions at the same time, thus increasing the speed of the CPU when more instructions are queued.

The first reason not to increase stages past 5 would be because speed is increased as different parts of hardware can simultaneously work on different instructions. Increasing stages would mean different stages of the same instruction require the same parts of hardware, and so you would have a hazard of waiting for one stage to complete before running the next, when both could have been completed in the same clock cycle.

The next reason is that some sections cannot easily be split up any further without serious issues occurring – such as the MEM stage which is often idle to prevent branch delays. If there is a jump in the instruction, you may have to stop the pipelined code which comes after the jump from executing depending on the jump. Therefore, as all of MEM needs to be idled, it can't easily be split up. If you split other sections down and not MEM, then the stages become uneven, and the MEM stage may take longer than all of the split stages. The time taken to complete an instruction depends on the longest stage to complete, and so if the longest stage doesn't improve, then there is no performance benefit from further splitting.

(b)

4096 lines = 2^{12} , so 12 bits to store the line number = index
512 byte line size = 2^9, so 9 bits required to specify where in the line = offset
32 – 9 – 12 = 11, so 11 bits for the tag

Offset: 9 bits
Index: 12 bits
Tag: 11

Therefore, we require 512 * 4096 + 2^{9+12} = 4194304 bytes of memory to fill the cache.

00D36E64 would translate to:
Offset: 000000001
Index: 101001101101
Tag: 11001100100

(c)

The first reason is that cores don't help increase the speed past a certain number with the way programs are written today. 10,000 cores would allow for 10,000 simultaneous instructions to run, as each core can run one at a time. However, designers have aimed for larger, more complex instructions which decreases the number of instructions, rather than having smaller, but a greater number of, instructions. Therefore, transistors are more important to store the larger instructions, and cores are less important as there's fewer instructions to run simultaneously.

Another reason is that transistors have gotten smaller to a larger degree than the cores. This means that the trade-off between transistors and cores is not equal – increasing the number of cores takes away a greater number of potential transistors we could fit onto the chip. This meant that there is more potential for a speed increase when we design applications for larger, more complex tasks, as the benefit from the increasing in potential number of transistors has increased compared to the benefit of potentially more instructions running at one time.

2.

(a)

For three variable boolean functions, it would require 4 input bits as the functions take three arguments.

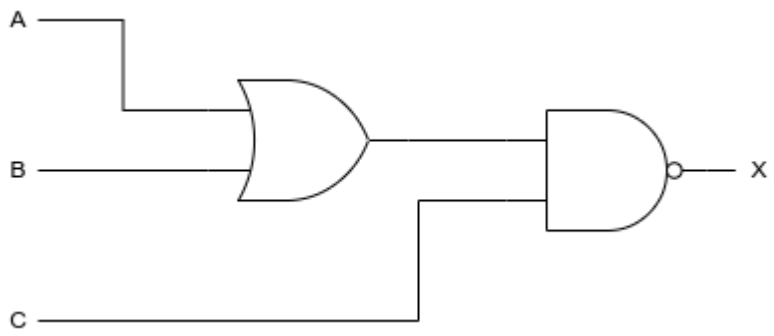
You would be able to use a multiplexer for this, as you would have the three bits for data at the top, with three NOT gates leading into the first AND gate with 4 inputs, and then for each other AND gate use a different combination of the NOT and source data inputs, with the other input control bits being directly connected. If you couldn't use and AND gate with 4 inputs, then you could chain AND gates together to reach the same affect as $(A \wedge B) \wedge C)$ is the same as $(A \wedge B \wedge C)$.

(b)

0 when C and (B or A)

A	B	C	X	-(C $\wedge$ (B $\vee$ A))
0	0	0	1	1
0	0	1	1	1
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1

1	0	1	0	0
1	1	0	1	1
1	1	1	0	0



(c)

No. Gates: (n-1) * 5 + 2
For the first bit, we only require a half-adder, as there is no carry bit required, and so 2 gates are needed. For all subsequent bits, we require a full-adder, which has 5 gates, and so we have the function of n for the number of gates being: (n-1) * 5 + 2

Gate Delay: (n-1) * 3 + 1
For the first bit, we have a half-adder, where there are two parallel gates and so the longest route to an output is 1 gate, giving a maximum 1 gate delay. For all other bits, we have a full-adder, which means the longest route to an output is to the carry bit, which goes through a XOR, AND, and OR gate giving a maximum gate delay of 3. Therefore, as a function of n, the maximum gate delay is: (n-1) * 3 + 1.

The trade-off is complexity vs. speed, as a ripple adder is very simple and understandable in it’s design and so good as a demonstration of the technology. However, ripple-adders have a very high gate-delay, and so it takes a long time for the output to become stable. Other techniques are beneficial as they decrease the maximum gate-delay, and so vastly increases the speed of adding, which is especially useful the more bits you want to add.

CS2002 Exam – Student ID: 200007413

3.

(a)

```
typedef struct Collection {
        void** data;
        int max_size;
        int size;

} Collection;
```

(b)

```
Collection* new_Collection(int max_size) {

        // Allocate space for the collection
        Collection* collection = malloc(sizeof(Collection));

        // Check that the collection created successfully
        if (collection == NULL) {
                perror("Fatal Malloc Error:");
                abort();
        }


        // Allocate space for a pointer array of a given size
        collection→data = malloc(sizeof(void*) * max_size);

        // Check the data array created successfully
        if (collection→data == NULL) {
                perror("Fatal Malloc Error:");
                abort()
        }

        // Set the initial collection values
        collection→max_size = max_size;
        collection→size = 0;

        return collection;

}
```

(c)

```
bool Collection_add(Collection* this, void* element) {
	// Check the collection is not null
	if (this == NULL) {
		printf(stderr, “Error: Cannot add to a null collection);
		return false;
	}

	// Check the element is not null
	if (element == NULL) {
		printf(stderr, “Error: Cannot add a NULL element to the collection);
		return false;
	}

	// Check the collection is not full
	if (this→size >= this→max_size) {
		printf(stderr, “Error: Cannot add element – collection is full”);
		return false;
	}

	// Add the element to the collection array, incrementing the size afterwards
	*(this→data + (this→size++ * sizeof(void*)) = element;

	return true;
}



int Collection_size(Collection* this) {
	// Check the collection is not null
	if (this == NULL) {
		printf(stderr, “Error: Cannot get size of a null collection);
		return -1;
	}
	return this→size;

}

void Collection_free(Collection* this) {
	// Check the collection is not null
	if (this == NULL) {
		printf(stderr, “Error: Cannot free a null collection);
		return;
	}

	// Free the collection data array
```

```
        free(this→data);

        // Free the collection
        free(this);

}
```

For the Collection_add method, I have assumed that the element added is a real generic pointer. If someone was to cast an item to a void* pointer and add that then data would be lost when casting which wouldn't break the collection but obviously would be useless information when retrieved.

I have also assumed that a failed malloc should be treated as fatal and abort the entire program.

For the Collection_size method, I have assumed that -1 is the error return code.

For the Collection_free method, I am assuming that the given collection data array or collection itself hasn't already been freed otherwise undefined behaviour would occur.

I have also assumed that any malloced data in the collection is going to be managed and freed elsewhere.

For all functions, I have also assumed that a not NULL collection* is a real collection* type as I don't think it is possible to verify. If someone casted a pointer to a collection* type then undefined behaviour would occur.

I have also assumed that a reference data structure for knowing where an added element is stored in the data array is not required.

I have also assumed that the pointers are going to be retrieved within the scope they were added, or the data the pointers point to is already dynamically allocated, as otherwise the pointers would have decayed and be meaningless.

(d)

// NOT COMPLETED

4. Concurrent Computation

Please see the file q4a.c and file q4b.c for a better formatted version of the answers given below.

a)

```
// Make global variables accessible to the threads
int sum;
```

```
int *thread_array;
pthread_mutex_t sumMutex;

// Create a structure for thread parameters containing the start and end of
// the sum
typedef struct ThreadOpts {
  int low;
  int high;
} ThreadOpts;


/** GIVEN AS ASSUMED **/
int computePartialSum(int *array, int low, int high) {
  int sum = 0;
  for (int i = low; i < high; i++) {
    sum += array[i];
  }
  return sum;
}

// Thread method which takes arguments via a ThreadOpts struct holding high and
// low indexes for summing in an array
void *thread_computePartialSum(void *args) {
  // Cast the argument to the options structure
  ThreadOpts *opts = (ThreadOpts *)args;

  // Do the hard work in parallel
  int partial_sum = computePartialSum(thread_array, opts->low, opts->high);

  // Lock the sum mutex for accessing the global sum
  pthread_mutex_lock(&sumMutex);

  // Add the calculated partial_sum for this section of the array
  sum += partial_sum;

  // Relock the mutex for other threads
  pthread_mutex_unlock(&sumMutex);

  return NULL;
}

int computeSum(int array[], int size) {
  pthread_t thread_bottom, thread_top;

  // Initialize the mutex for incrementing the sum
  pthread_mutex_init(&sumMutex, NULL);
```

```
  // Initialize the values, making sum 0, and the array accessible to the
  // threads
  sum = 0;
  thread_array = array;

  // Create the option parameters for each thread
  // Odd numbers will truncate fractions so this will work with one thread
  // doing an extra calculation
  ThreadOpts bottom_opts = {0, size / 2};
  ThreadOpts top_opts = {size / 2, size};

  // Create two threads, one which totals from the top, and one which totals
  // from the bottom
  pthread_create(&thread_bottom, NULL, thread_computePartialSum, &bottom_opts);
  pthread_create(&thread_top, NULL, thread_computePartialSum, &top_opts);

  // Join the threads together to calculate the total sum
  pthread_join(thread_bottom, NULL);
  pthread_join(thread_top, NULL);

  // Return the calculated total sum
  return sum;
}
```

b.

```
// Code not complete, but calculates each separate part in a process,
// I just had trouble merging the two back together

int computeSum(int array[], int size) {

  int top_cpid = 0;
  int bottom_cpid = 0;

  // Get the parent pid
  int ppid = getpid();

  // Set the value of sum to 0
  int sum = 0;

  // Fork the program to calculate the top half
  top_cpid = fork();

  // Fork to calculate the bottom half
  if (top_cpid != 0) {
    bottom_cpid = fork();
  }
```

```
  // The bottom half process will know the top_cpid
  if (top_cpid != 0 && bottom_cpid == 0) {
    // Bottom calculation
    sum = computePartialSum(array, 0, size / 2);
    exit(sum);
  }

  // The top half process will not know either child cpids
  if (top_cpid == 0 && bottom_cpid == 0) {
    // Top Calculation
    sum = computePartialSum(array, size / 2, size);
    exit(sum);
  }

  int child1_sum;
  int child2_sum;

  // Wait for both the top and bottom proceses to complete
  wait(&child1_sum);
  wait(&child2_sum);

  return child1_sum + child2_sum;
}
```