CPSC 261 Midterm 1
Tuesday February 9th, 2016

[10] 1. Short Answers

[2] a. Two CPUs support the same instruction set, and use the same amount of power. CPU A's latency and throughput are both 10% lower than CPU B's. Which CPU should you choose, and why?

Solution : Throughput is the measure we really care about (how long a single instruction takes is not all that relevant), so we would choose CPU B.

[4] b. One technique used to cope with complexity is to be *tolerant of inputs* and *strict on outputs*. What does each of these terms mean?

Solution : By *tolerant of inputs* we mean that functions should do something sensible when they are given incorrect (or slightly out of range) inputs. By *strict on outputs* we mean that functions should not produce incorrect, unexpected, or not-as-specified-by-the-documentation results.

[4] c. What do we mean by *internal* and *external* fragmentation, when we are talking about dynamic memory allocation?

Solution : *Internal fragmentation* refers to space that is wasted inside blocks because of either alignment requirement, or constraints on block sizes imposed by the memory allocator. *External fragmentation* refers to the situation where free heap space is divided into many non-adjacent blocks, which means the allocator may be unable to satisfy a request even though the total amount of space available is sufficient.

[6] 2. What is the output of the following C program? Justify your answers!

```
#include <stdio.h>

int array[] = { 1, 3, 5, 7 };

void crazy_function(int *p, int *q, int **r) {
  int i;

  for (i = 0; i < 4; i++) {
      *q += array[i];
  }
  *r = array + 3;
}

int main(int argc, char *argv[]) {
```

```
    int *x = array + 1;
    int i;

    crazy_function(array, &array[2], &x);
    printf("Value x points to is %d\n", *x);

    for (i = 0; i < 4; i++) {
        printf("Element at position %d is %d\n", i, array[i]);
    }
    return 0;
}
```

Solution : Inside the call to function `crazy_function`, pointer `q` points to `array[2]`, and hence all of the elements of the array (including the value of `array[2]` at that point in the execution of the `for` loop) are added to it. The output is thus

```
Value x points to is 7
Element at position 0 is 1
Element at position 1 is 3
Element at position 2 is 25
Element at position 3 is 7
```

[11] 3. Pipelines

[2] a. A CPU uses a three stage pipeline; the minimum time needed by each stage is 150ps, 200ps and 180ps respectively (including the time needed to load the stage registers). What is the CPU's maximum theoretical throughput?

Solution : In the best case, an instruction will leave the pipeline every 200ps (remember that all stages will take as much time as the slowest one). So the throughput is $1/200\text{ps} = 1/2 \times 10^{-12} = 5 \times 10^{9}$ instructions per second.

[3] b. What is the difference between a *data hazard* and a *control hazard*?

Solution : A data hazard occurs when an instruction needs a value (usually a register value) that is not yet available because the instruction computing it has not yet finished executing. A control hazard happens when the CPU does not know for sure which instruction it should fetch next, and usually results from a conditional branch, indirect jump or call, or `ret` instruction.

[3] c. What technique do modern pipelined CPUs use to prevent an instruction i from stalling until the instruction producing a data value that i needs has stored it in a register?

Solution : Often, even though the value has not yet been stored in the machine register, it has already been computed and is available in a stage register somewhere inside the pipeline. The value can then be forwarded to the stage that needs it (typically Decode) without having to wait for the first instruction to complete.

[3] d. Describe one situation *other than a stall* where a CPU will insert `nop` instructions in its pipeline.

Solution : If it has mis-predicted the outcome of a conditional branch, then it will need to turn the instructions that it has already started executing (incorrectly) into `nop`s.

[7] 4. Given the following C function,

```
/* Insert element "value" at the right position in sorted */
/* array "ptr" with "n" elements.                          */
void insert(long *ptr, long n, long value) {
    while (--n >= 0 && value < ptr[n]) {
        ptr[n+1] = ptr[n];
    }
    ptr[n+1] = value;
}
```

the `gcc` compiler generates the following x86-64 assembly language code:

```
1   insert: jmp     .L2
2   .L4:    movq    %rax, 8(%rdi,%rsi,8)
3   .L2:    subq    $1, %rsi
4           js      .L3
5           movq    (%rdi,%rsi,8), %rax
6           cmpq    %rdx, %rax
7           jg      .L4
8   .L3:    movq    %rdx, 8(%rdi,%rsi,8)
9           ret
```

Using the C version as a guide, and your knowledge of the x86-64 calling conventions, comment the assembly language version of the function. Hint: `js` means "jump if negative".

Solution : Here are the comments (we did not grade the comments for lines 1 and 9, as it's hard to comment these sensibly).

(1) Jump to the real beginning of the function.

(2) Assign `ptr[n]` (stored in `%rax` by instruction 5) to `ptr[n+1]`.

(3) Decrement `n`.

(4) If `n` is negative, exit the loop.

(5) Read `ptr[n]` into register `%rax`.

(6) Compare `ptr[n]` to the parameter `value`.

(7) If `ptr[n]` is smaller than `value`, then go to the loop body.

(8) Store `value` into `ptr[n+1]`.

(9) Done: return from the function.

[7] 5. Here is a longer function (insertion sort) that incorporates function `insert` from the previous question inside a loop.

```
1  isort:  movq    $1, %r8
2          jmp     .L6
3  .L10:   movq    (%rdi,%r8,8), %rcx
4          movq    %r8, %rax
5          jmp     .L7
6  .L9:    movq    %rdx, 8(%rdi,%rax,8)
7  .L7:    subq    $1, %rax
8          js      .L8
9          movq    (%rdi,%rax,8), %rdx
10         cmpq    %rdx, %rcx
11         jl      .L9
12 .L8:    movq    %rcx, 8(%rdi,%rax,8)
13         addq    $1, %r8
14 .L6:    cmpq    %rsi, %r8
15         jl      .L10
16         rep ret
```

List at least five hazards that are present in this function. For each hazard:

- If it is a data hazard, write `D`, and both the number of the instruction that uses the data value, and the number of the instruction that computes it. If an instruction uses several values computed by earlier instructions, list only the number of the most recent such instruction.
- If it is a control hazard, write `C` and the number of the instruction where the hazard occurs.

Solution : There were many more than five hazards. They are:

	Type	Instruction 1	Instruction 2	Register?
1.	C	8		
2.	C	11		
3.	C	15		
4.	C	16		
5.	D	1	14	%r8
6.	D	4	7	%rax
7.	D	7	9	%rax
8.	D	9	10	%rdx
9.	D	13	14	%r8

[9] 6. The standard C library function `realloc` takes in a pointer `ptr` and an integer `user_size` as parameters, and changes the size of the memory block that `ptr` points to so it becomes `user_size`. The contents of the payload are preserved (up to the smaller of the old and new sizes). For efficiency reasons, `realloc` will **extend the current block** (make it bigger) if there is enough available space after it. Here is a partial implementation of the `realloc` function within the context of lab 3: Fill in code for the case where `curr_size < real_size` (other than the else clause). The next page contains function headers for some of the functions from lab 3. Hint: our solution contains 6 lines of actual code (plus a few lines with only curly braces).

Solution :

```
void *heap_realloc(heap *h, void *ptr, long user_size)
{
    long real_size = get_size_to_allocate(user_size);
    char *block_start = get_block_start(payload);
    long curr_size = get_block_size(block_start);
    long curr_block = block_start;

    if (curr_size > real_size)
    {
        /* Ignore this case; we will shrink the block (maybe). */
    }
    else if (curr_size < real_size)
    {
```

```
        long max_size = curr_size;
        if (!is_last_block(h, block_start) &&
            !block_is_in_use(get_next_block(block_start)))
        {
            max_size += get_block_size(get_next_block(block_start));
        }
        if (max_size >= real_size)
        {
            set_block_header(block_start, max_size, 1);
            prepare_block_for_use(block_start, real_size);
        }
        else
        {
          /* Case where we could not extend the block enough. */
        }
    }
    return get_payload(block_start);
}
```