CS24 Midterm 2018 – COVER SHEET

This midterm requires you to write several programs, and to answer various questions. As with the homework assignments, a tarball is provided, **`cs24mid.tgz`**. When you start the midterm, you should download and extract this tarball and work in the directories specified by the problems.

When you have finished the midterm, you can re-archive this directory into a tarball and submit the resulting file:

> *(From the directory containing **`cs24mid`**; replace **`username`** with your username.)*
> **`tar –czvf cs24mid-`*`username`*`.tgz cs24mid`**

Important Note!

If you decide to not follow the filenames specified by the midterm problems, or your tarball is really annoying in some way (e.g. the permissions within the tarball have to be overridden by the TAs before they can grade your work!), you may lose some points. If you think you might need help creating this tarball properly, talk to Donnie or a TA; we will be happy to help.

Midterm Rules

The time limit for this midterm is 6 hours, in multiple sittings. The main rule is that if you are focusing on the midterm, the clock should be running. If you want to take the entire midterm in one sitting, or if you want to take a break (or a nap) after solving a problem, do whatever works best for you. Just make sure to track your time, and only spend 6 hours working on the exam.

If you need to go overtime, just indicate where time ran out. You receive half-credit for the first hour of overtime, no credit thereafter.

You may use any official course material on the exam. You may refer to the book, the lecture slides, your own assignments, solution sets, associated reference material, and so forth.

No collaboration with students, TAs, or others. You may not talk to another student about the contents of the midterm until both of you have completed it. You may talk to a TA about the midterm after you have completed it. You may also talk to a TA if you need help packaging up your midterm files, as mentioned above. Obviously, do not solicit help from others on the exam either.

Request any needed clarifications directly from Donnie. The TAs aren't responsible for writing the midterm. Feel free to contact Donnie if you have any questions. If you can't make any progress while you wait for an answer, stop your timer.

You may use a computer to write, compile, test, and debug your answers. In fact, this is encouraged, because we will deduct points for incorrect programs. Some problems include test code you can use to try your answers, although they may not detect all bugs (or any style issues!).

You may not use a disassembler, or the `gcc –S` feature, on the exam. The x86-64 assembly code you turn in should be entirely self-written by you.

WHEN YOU ARE READY TO TAKE THE MIDTERM, YOU MAY GO ON TO THE NEXT PAGE! GOOD LUCK!

Problem 1: Instruction Decoding *(24 points)*

(The files for this problem are in the **`cs24mid/idecode`** *directory of the archive.)*

The simple processor used with Assignment 1 used a *very* simplistic instruction encoding mechanism, where all instructions are the same bit-width, and the different components always reside at the same locations in each instruction. As we saw, this can waste a significant amount of space in the program's binary representation.

In this problem you will implement a slightly more advanced instruction format and decoding mechanism. Instructions will take either one or two bytes, and there should be relatively little wasted space in the resulting format. Some notable changes:

- The destination register is no longer explicitly specified; it is always the second source register.
- For two-argument instructions, the first argument can be a constant or a register. (The instruction encodes this detail.) The second argument is always a register, and is also the destination of the operation.
- The processor architecture has been altered so that the branching unit receives a status value from the ALU; thus, branching instructions only take an address as their argument.

The instructions along with their encoding formats are specified in the following table. Note that values in square brackets "[…]" specify the bits of each byte; there will be 8 bits (obvious), and the most significant bit is on the left. Positions with an "x" mean that the bit is unimportant and can be ignored. As a reminder, this CPU has 8 registers, so registers are denoted with values from 0 to 7.

Opcode	Assembly	Meaning
Zero-argument control instructions: always occupy 1 byte Encoding: [op3 op2 op1 op0 x x x x] • [op3 … op0] are the bits of the opcode		
0000	DONE	Stops the program execution
One-argument ALU instructions: always occupy 1 byte Encoding: [op3 op2 op1 op0 x src2 src1 src0] • [op3 … op0] are the bits of the opcode • [src2 … src0] are the bits of the register argument		
0001	INC B	B := B + 1
0010	DEC B	B := B – 1
0011	NEG B	B := -B
0100	INV B	B := ~B
0101	SHL B	B := B << 1
0110	SHR B	B := B >> 1 (logical shift right)
Two-argument ALU instructions: always occupy 2 bytes Byte 1 encoding: [op3 op2 op1 op0 a_isreg regb2 regb1 regb0] • [op3 … op0] are the bits of the opcode • a_isreg is a flag: 1 means the first argument is a register; 0 means the first arg is a constant		

- [regb2 … regb0] are the bits of the *second* register argument

Byte 2 encoding (when a_isreg == 1): [x x x x x rega2 rega1 rega0]

Byte 2 encoding (when a_isreg == 0): [8-bit constant]

1000	MOV A, B	B := A
1001	ADD A, B	B := B + A
1010	SUB A, B	B := B – A
1100	AND A, B	B := B & A
1101	OR A, B	B := B | A
1110	XOR A, B	B := B ^ A

Branching instructions: always occupy 1 byte

Encoding: [op3 op2 op1 op0 addr3 addr2 addr1 addr0]

- [op3 … op0] are the bits of the opcode
- [addr3 … addr0] are the bits of the address to branch to

0111	BRA Addr	PC := Addr
1011	BRZ Addr	If ALU zero-flag == true, PC := Addr
1111	BNZ Addr	If ALU zero-flag == false, PC := Addr

Your Tasks

You will need to provide an implementation of the **`fetch_and_decode()`** function in the **`branching_decode.c`** file. A skeleton implementation is provided for you, including all of the values that instructions must be decoded into.

Note that unlike the Assignment 1 processor, all values from the Instruction Store are now bytes, not dwords. When you are decoding a multi-byte instruction, you must move the program counter forward and then retrieve the next byte from the instruction store. You will need these operations:

`ifetch(InstructionStore *)` – The Instruction Store reads the instruction at PC, and pushes it onto a bus. This value becomes visible to the instruction decoder on the **`d->input`** pin.

`incrPC(ProgramCounter *)` – When decoding multi-byte instructions, the program counter must be moved forward independent of the branching logic. This function will do exactly that. (Note that after incrementing the program counter, **`ifetch()`** must be used to push the new instruction onto the bus, and then the instruction byte must be read from the **`d->input`** pin.)

You must decode all instructions specified above. Note that you only need to implement the decoding portion; other portions of the processor are already implemented for you.

Once you have completed this implementation, you can use the **`multiply.ibits`** and **`multiply_*.rbits`** files to test your work. The multiplication routine is simply the one used in class; here is the assembly code that was hand-translated into machine code in the above file:

```
# Inputs:  R0 = A, R1 = B
# Result:  R7 = P (= A * B)

    MOV 0, R7      # Set P = 0
    AND R0, R0     # If A == 0:
    BRZ MUL_DONE   #     Finished.
```

```
MUL_LOOP:
    MOV R0, R2
    AND 1, R2      # If low bit of A == 1:
    BRZ SKIP_ADD
    ADD R1, R7     #     P := P + B

SKIP_ADD:
    SHL R1         # B := B << 1
    SHR R0         # A := A >> 1 (also sets zero-flag status value)
    BNZ MUL_LOOP   # If A still ain't 0, do more multiplyin'!  Yee Haw!

MUL_DONE:
    DONE
```

In a file **`idecode/questions.txt`**, answer the following question:

1. Given that the first instruction of the encoded program will start at PC = 0, what are the offsets (i.e. PC values) of the labels **`MUL_LOOP`**, **`SKIP_ADD`**, and **`MUL_DONE`** in the binary program?

2. Specify the assembly instructions and the encoded bit patterns for the following two operations. The encoded versions should be a sequence of 0s and 1s, with most significant bit at the left. Feel free to use spaces to separate different components of the encoded version.

   R5 := R5 – 6

   R4 := R4 ^ R1

(Scoring: `fetch_and_decode()` = 14 points; Q1 = 6 points; Q2 = 4 points.)

Problem 2: Big Fibs *(28 points)*

(Complete this problem in the **`cs24mid/bigfib`** *subdirectory of your submission.)*

So far in CS24 we have worked primarily with 32-bit **`int`** values, which can only represent unsigned values between 0 and 2^{32}-1 (or 4,294,967,296). This hasn't been a problem for us, but there are many situations where this is far too limiting. Take, for example, the Fibonacci sequence:

$fib(0) = 0$
$fib(1) = 1$
$fib(n) = fib(n\text{-}1) + fib(n\text{-}2) \qquad (n > 1)$

Let's say that we want to compute *fib*(100). This works out to approximately $3.54\text{x}10^{20}$, which is far too large to store in a 32-bit unsigned integer. Unfortunately, even if we use a 64-bit unsigned integer (which can store a value of around $1.8\text{x}10^{19}$), we are still out of luck.

Clearly, the standard integer types are too limiting for these kinds of computation. But, we can stitch together multiple unsigned integers into a "bigint" data type that will hold all the bits of a larger integer value. Since we are on a 64-bit architecture, and want to take advantage of our full data bus size, we will use 64-bit components.

A bigint variable *n* represents an arbitrary-size unsigned integer value, comprised of *size* 64-bit unsigned-int components, laid out as follows:

***n*[0]**	***n*[1]**	***n*[2]**	**...**	***n*[*size*-1]**
Bits 0..63 of *n*	Bits 64-127 of *n*	Bits 128-191 of *n*	...	High 64 bits of *n*

In other words, *n*[0] is the low 64 bits of *n*, *n*[1] is the next 64 bits of *n*, and so forth.

Note that for the sake of simplicity, this is an unsigned bigint.

To compute *fib*(*n*) = *fib*(*n*-1) + *fib*(*n*-2), we need to be able to add together these bigint values. A simple function like this wouldn't work:

```
/* Implements b += a... or does it? */
add_bigint(a, b, size) {
    for (int i = 0; i < size; i++)
        b[i] += a[i];
}
```

The problem is, when *a*[0] and *b*[0] are added, there may be a carry-out that should be applied to *a*[1] and *b*[1]. Correspondingly, there may be a carry-out from *a*[1] and *b*[1] that must be applied to *a*[2] and *b*[2]. And so forth.

Fortunately, the Intel x86 processor provides the **`adc`** instruction, which can be used to perform a sequence of additions where the carry-out of each addition should be used as the carry-in of the next addition. The caveat is, you must write the multi-word addition in assembly language.

Your Tasks

For this problem you will need to implement this function in x86-64 assembly language:

```
int __add_bigint(uint64_t *a, uint64_t *b, uint8_t size)
```

- Both **a** and **b** are arrays of **uint64_t** values (i.e. 8-byte quadwords that hold unsigned values) representing unsigned bigints.
- Both arrays contain **size** elements. Note that **size** is an unsigned byte.
- The function should implement **b += a** (i.e. add the value of bigint **a** into bigint **b**).
- The function should return 1 if the addition is completed successfully (no unsigned overflow), or 0 if the addition results in an unsigned overflow.
- The function should handle the case where **size** == 0 (do nothing, and return 1).

This function is called by the **add_bigint(bigint_t *a, bigint_t *b)** function in **bigint.c**, which makes **__add_bigint**'s life simpler by performing useful error-checking, and then passing only what **__add_bigint** needs to do its job.

1) *(8 points)* In the file **bigint.txt**, write pseudocode for the implementation of **__add_bigint**.

   If your pseudocode includes for-loops, if-statements, or while-loops, transform it into goto-style code, as discussed in class. Show your work; make a copy of your original pseudocode, apply some transformations, and repeat this process until your pseudocode is ready to implement.

   For full credit, make sure to comment anything subtle in your pseudocode.

   Here are a few hints that may affect the design of your algorithm:

   - It's easier to perform the entire sequence of additions with the **adc** instruction, rather than trying to use **add** on *a*[0] and *b*[0], then using **adc** for the rest.

     The **clc** instruction may be useful – it clears the Carry Flag.

   - Note that **add**, **adc**, **sub** and **cmp** all modify the Carry Flag (as well as other flags). This means that using **cmp** to check when to exit a loop will interfere with the operation of **adc**. (If you were to use both **adc** and **cmp**, you would need to save and restore the flags register with **pushf** and **popf**, which simply gets too complicated.)

     However, **inc** and **dec** do not modify the Carry Flag. As stated in the Intel x86-64 manual, **inc** and **dec** can be used to update loop-variables in loops that must preserve the Carry Flag from iteration to iteration. They still update other flags, such as the Zero Flag.

2) *(14 points)* In the file **bigint_asm.s**, implement the **__add_bigint** function. To receive full credit, you must also provide detailed comments in your implementation, including a comment header at the top of the function describing its purpose, arguments, and return value.

Testing Bigint Addition

A simple program **bigfib** is provided for testing your code. The program takes two command-line arguments, which you can see by running **bigfib** without arguments:

```
usage: ./bigfib n size [-v]
      Prints the n-th value in the Fibonacci sequence, using
      a size-quadword bigint representation for computations.
```

```
fib(0) = 0, fib(1) = 1, fib(n) = fib(n-2) + fib(n-1).

-v causes the program to print out all Fibonacci numbers
between fib(0) and fib(n).
```

You can use this program to check that your addition is working correctly. For example, "**`./bigfib 20 1 -v`**" should print out the first 20 Fibonacci numbers, using a single-quadword bigint. You can scale up the number of quadwords used, and the *n* to compute, as you gain confidence that your code is working correctly.

Note: Feel free to use WolframAlpha to verify the larger Fibonacci numbers. You can ask it to compute any Fibonacci number like this: http://www.wolframalpha.com/input/?i=fib(100)

Questions

Once your unsigned bigint addition is working correctly, answer these questions in **`bigint.txt`** (including the relevant portions of **`bigfib`**'s output that demonstrate the answers).

NOTE: Since it is also possible to answer these questions without a working **`bigfib`** program, you will only get partial credit for correct answers if your **`bigfib`** program doesn't also work.

3) Given a bigint comprised of four 64-bit quadwords, what is the *n* of the largest Fibonacci number *fib*(*n*) we can compute in this space without an unsigned overflow? *(3 points)*

4) How many 64-bit quadwords are required to compute *fib*(2000)? What is the value of *fib*(2000)? *(3 points)*

Problem 3: Buddy Allocators *(32 points)*

(Answers for this problem should go in the file **`buddy.txt`** provided for you.)

Buddy allocators are a category of allocators that impose specific constraints on the size of blocks that can be allocated, and also what blocks can be coalesced with each other. Because these allocators tend to be simple to implement, and reasonably fast, they are often used in operating systems for when the kernel needs to dynamically allocate memory.

The simplest kind of buddy allocator is the *binary buddy allocator*. Blocks have a minimum size MIN_BLOCK_SIZE, and they also have a *size order*; a block with a given order has this size: MIN_BLOCK_SIZE * 2^{order}. A block of order 0 is the smallest block that the allocator can provide, and as the order is increased, blocks double in size. Therefore, given a block with an order *i*, it should be obvious that it contains exactly two blocks of order *i* – 1.

Every block (except the block with the largest order) has exactly one unique buddy (hence the term *binary* buddy allocator). When a block of order *i* is divided into two smaller blocks of order *i* – 1, the two blocks are buddies of each other. A block can only be coalesced with its own buddy, and only if the two blocks have the same size order.

Initially, the buddy allocator contains a single free block of the maximum order that can fit within the heap. When an allocation request is made, the allocator determines the smallest order that will still contain the requested size, and then it attempts to find and return a block of the required order. It does this by finding the smallest-order free block currently in the heap that is at least the order required for the allocation request. If the block has a larger order than the required order, the allocator repeatedly subdivides this block until it has a block of the appropriate size order for the request. (When dividing a free block, the allocation request goes in the left buddy; this makes the allocation operation very clean to implement.)

Deallocation is straightforward; when a block is freed, the allocator looks for the block's buddy, which will either be before or after the block. If the buddy is also a free block of the same order *i*, the two blocks are coalesced into a new free block of order *i* + 1. This process may need to be repeated multiple times if the new block also has a free buddy that can be coalesced.

It is often of immense value for the binary buddy allocator to maintain an array of free-lists for each size order that it can provide. The free-list at index 0 is a list of available blocks of size order 0, the list at index 1 is a list of available blocks of order 1, and so forth. This makes it very easy for the buddy allocator to determine what free blocks are available when a given request comes in, and to identify a free block with an order closest to the order required for the allocation request. As with the explicit free-list allocator described in class, unused blocks are chained together with explicit pointers stored within the unused blocks.

For this problem, assume that MIN_BLOCK_SIZE is 32 bytes. Also, all sizes given in KiB/MiB/GiB are kibibytes/mebibytes/gibibytes; i.e. 1KiB = 1024 bytes, 1MiB = 1048576 bytes, etc.

a) If the maximum memory heap size is 2GiB (i.e. 2^{31} bytes), what is the maximum size order that the allocator can provide? How large will the free-list array be in this situation, and why? *(3 points)*

b) Given a specific allocation request size, the buddy allocator must determine the minimal size order that can still hold the request. Write the implementation of a C function **`int get_order_of_size(int size)`** that returns the smallest size order that can still hold

the specified size. Feel free to use the MIN_BLOCK_SIZE constant in your implementation. (We are only going to visually inspect your implementation, so you don't need to compile and test your work.)
(4 points)

c) It is possible that the allocator cannot satisfy an allocation request and must return **NULL**. Describe how the allocator should detect whether a request cannot be satisfied; be specific in your answer. *(4 points)*

d) Assume the overall heap size is 16KiB. This gives a maximum size order of 9 ($32 * 2^9 = 32 * 512 = 16384$). Here is a sequence of allocations and deallocations:

1. A = allocate(1400);
2. B = allocate(5500);
3. C = allocate(800);
4. D = allocate(3200);
5. free(A);
6. E = allocate(700);
7. free(B);
8. free(C);
9. free(E);
10. free(D);

Describe the heap's state before the first allocation, and after each step has been completed, making sure to indicate every block currently managed in the heap. You must also indicate the order of each block, and whether the block is free or allocated. You do not need to describe the array of free-lists in your answer.

After the completion of step 4 in the sequence, how many bytes are still available to satisfy allocation requests? How many bytes are unused by the program, but not available to satisfy allocation requests?

(10 points)

e) As stated before, buddy allocators are frequently used within operating system kernels, but as you should notice from the previous problem, they have a significant limitation that impacts their effectiveness. Describe this limitation. *(4 points)*

f) Of the three placement strategies discussed in class, first-fit, next-fit, and best-fit, which of these is closest to the placement strategy that the buddy allocator uses? Explain your answer.
(3 points)

g) Buddy allocators frequently benefit from a technique called *deferred coalescing*, in which they do not immediately coalesce a freed block with its buddy.

Describe a program behavior in which deferred coalescing would be preferable over immediate coalescing. Make sure to identify the specific benefits achieved by deferred coalescing.
(4 points)

Problem 4: Explicit Allocator Policies *(16 pts)*

(Answer these questions in the files **`prob4?.txt`**.)

Consider a system with a fixed amount of heap memory. The system has an explicit allocator that uses a simple implicit free-list implementation.

- Initially, all heap memory is contained in a single free block.
- When an allocation request is received, a free block is selected according to some policy (e.g. first-fit or best-fit), the chosen free block is split (if necessary), and the newly allocated block is taken from the start of the free block. (Any remainder of course becomes a new free block.)
- When a block is freed, the allocator coalesces the newly freed block with any existing free-blocks on either sides of the block.

An allocation failure occurs if a request is made that cannot be satisfied; i.e. no memory fragment is large enough to hold the requested block.

(a) Construct a simple scenario where an allocation failure would occur under a first-fit policy, but not a best-fit policy. *(8 pts)*

For your scenario, make sure to specify the total heap size, and the size and order of allocations and releases. For simplicity and consistency, specify all sizes in KB.

Diagram the resulting state of memory at the end of your scenario, indicating which blocks are allocated and which are freed, along with the size of each block. (A list of blocks and their info is fine, or ASCII art, or something more sophisticated. Just make sure to indicate the complete memory layout at the end of your scenario.)

Put your answer in the file **`prob4a.txt`**.

(b) Construct another simple scenario where an allocation failure would occur under a best-fit policy, but not a first-fit policy. *(8 pts)*

Again, make sure you completely describe your scenario by specifying the same details requested in part (a).

Put your answer in the file **`prob4b.txt`**.

END OF MIDTERM! YAY!