CS24 Midterm 2015 – 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 IA32 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 *(20 points)*

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

Processors generally perform these steps when executing instructions:

- Instruction fetch
- Instruction decode
- Execute
- Memory access
- Register write-back

The simple processor used with Assignment 1 also performs these steps in its controller (although there was no memory access phase since the processor doesn't access memory). This processor also 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. And, 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 be at PC = 0, what are the offsets 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 can 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()` = 12 points; Q1 = 4 points; Q2 = 4 points.)

Problem 2: Drawing Pixels *(30 points)*

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

In the screen directory you will find a very simple little library for drawing pixels, lines and circles in a simple virtual screen. The types for representing the screen's dimensions and state are declared in **screen.h**, and are as follows:

```
typedef struct Pixel {
    unsigned char value;
    unsigned char depth;
} Pixel;

typedef struct Screen {
    int width;
    int height;

    Pixel pixels[];
} Screen;
```

You will see in **screen.c** how this type is allocated and initialized.

All you have to do for this problem is to implement the **draw_pixel()** function in IA32 assembly code. The function signature is specified in the **pixel.h** header file, and your implementation should go in a file **pixel.s** that you create. You can see how this function is called by looking at the **smain.c** file and the **drawing.c** file. Some other specifications are as follows:

- **In C, multidimensional arrays are stored in *row-major order*.** This means that the elements in a given row are all adjacent to each other. To access a particular screen pixel (x, y), you would need to map this to a 1D index value with the computation y * width + x.
- **Your function should properly ignore out-of-bounds pixel writes.** You can treat pixel coordinates as unsigned values in your function if you like; this will keep you from having to check if they are negative. But you do need to check if they are beyond the width and height of the screen, and skip the write if it is out of bounds.
- Valid pixel coordinates (x, y) satisfy the following condition: $0 \le x < width$ and $0 \le y < height$
- **The depth value is used to support drawing some objects "in front of" or "behind" other objects.** If a pixel being drawn is "closer" than the existing screen-pixel value (i.e. the screen pixel's value is larger than the new pixel value) then the new pixel replaces the old pixel. If the pixel being drawn is "further away" than the existing screen-pixel value then the new pixel is simply not drawn.

  In the case where the incoming pixel has the same depth as the existing screen-pixel depth, the incoming pixel should replace the screen pixel's value.

As you will note, the **draw_pixel()** function takes a pointer to a **Screen** struct. As discussed in class, struct members are positioned at various offsets from the start of the struct, which means you should be able to use the IA32 memory addressing modes to access these values very easily. There is one caveat, since this particular type ends with an undimensioned array. In this case, *the array*

itself starts after **`height`** in the struct; **`pixels[0]`** will immediately follow **`height`** in the struct's memory region.

You should also note that since the **`Pixel`** type only consists of two bytes, the compiler doesn't pad it! In other words, **`sizeof(Pixel)`** is 2.

Your Tasks

a) In the file **`draw.txt`**, write pseudocode for the implementation of **`draw_pixel()`**. *(10 points)*

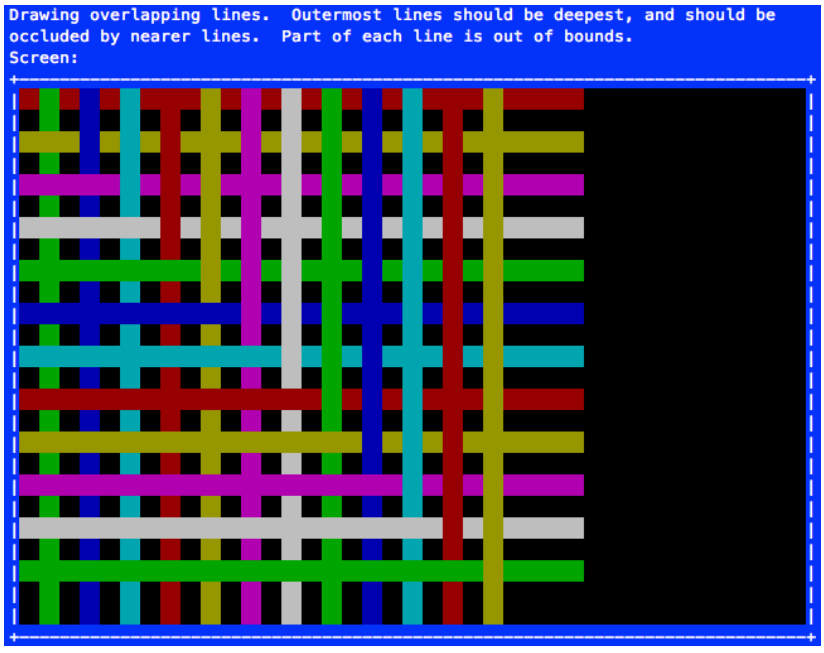
If your pseudocode includes for-loops, if-statements, or while-loops, make sure to transform your code following the guidelines in class (e.g. for-loop becomes a while-loop with initialization; while-loop becomes an if and a do-while). Show your work; make a copy of your original pseudocode, apply some transformations, and repeat this process until your pseudocode is ready to implement.

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

b) In the file **`pixel.s`**, implement the **`draw_pixel()`** function. You must provide detailed comments (including documentation of the function, arguments and their positions on the stack, and description of the implementation's operation) to receive full credit. *(20 points)*

Hint: You can always use portions of your pseudocode for some of this documentation.

The file **`smain.c`** will perform some basic drawing with your implementation. You should be in good shape if the first screen looks something like this:



Note: If your console is unable to display colors for some reason, change the **`USE_COLOR`** constant at the top of **`screen.c`** to 0 and recompile; letters will be used instead of colors.

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 hold the request. Write pseudocode (or use C if you prefer) for a 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. *(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 it 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 <u>not</u> 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: Garbage Collection *(18 points)*

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

The garbage collection algorithms we explored in class are designed for imperative programming languages that include a notion of *destructive assignment*, where an old value (e.g. a pointer or reference) can be replaced with a new value. The result is that a program's reachability graph may contain cycles, which must be properly handled by the garbage collection algorithm.

Some functional languages only allow *single assignment*, where no value may ever be replaced; once a value is set, this value remains unchanged until it is no longer in use by the program. (Some examples of such languages are Haskell, Erlang and OCaml; all worth exploring.) This has a significant impact on the way that garbage can be reclaimed in such systems, because it is actually impossible to produce a cycle in the reachability graph without destructive assignment.

The simple CS24 Scheme interpreter supports destructive assignment, specifically via operations like **`set!`**, **`set-car!`** and **`set-cdr!`**, among a few others. If we remove these operations that perform destructive assignment, we can greatly simplify the garbage collector.

Imagine that the Scheme interpreter's memory heap is structured as a sequence of cells, each containing these components:

type	*value*	*marked*	*prev_alloc*
An enumeration indicating the type of value: • integer • float • atom • cons-pair	A union of various types of values; the *type* member indicates which union-member to use.	A Boolean value indicating whether this value is reachable or not.	A pointer to the cell that was allocated immediately before this cell.

(Recall that a cons-pair *c* is simply a pair of values; the first value is retrieved with the expression **`(car c)`**, and the second value is retrieved with **`(cdr c)`**. As shown in Assignment 4, many kinds of data structures can be built up with this simple primitive, including lists, trees, etc.)

Whenever a new variable is created or a new expression is evaluated, one or more of these cells may be allocated from the memory pool, and the *type* and *value* are both set appropriately; these values may not change for as long as the cells are in use.

You will also notice the *prev_alloc* value; this allows the runtime to start at any given cell, and traverse all allocations that occurred before that specific cell was allocated, in the order that they were allocated. Among other things, the runtime also records the first and last memory cells in this sequence, so that it is easy to traverse all allocated cells. (Note that when a memory cell is reclaimed, it is removed from this list of cells; thus, a memory cell's *prev_alloc* actually points to the most recently allocated cell that has not yet been freed. A cell's *prev_alloc* value will change if the previous cell is reclaimed by the garbage collector.)

The runtime has these operations available to it:

- is_atomic(*cell*) – Returns *true* if the cell holds anything other than a cons-pair; *false* otherwise.
- is_cons(*cell*) – Returns *true* if the cell holds a cons-pair; *false* otherwise.
- car(*cell*) – If the cell holds a cons-pair, this returns a pointer to the memory cell that holds the value of **`(car cell.value)`**. If the cell doesn't hold a cons-pair, this is an error.
- cdr(*cell*) – If the cell holds a cons-pair, this returns a pointer to the memory cell that holds the value of **`(cdr cell.value)`**. If the cell doesn't hold a cons-pair, this is an error.
- is_marked(*cell*) – Returns *cell.marked*
- set_marked(*cell*, *mark*) – Sets *cell.marked* to the Boolean value *mark*; this function can be used to either mark or unmark a memory cell.
- free(*cell*) – Releases a memory cell back into the pool. You can assume this function also properly updates the linked-list of cells managed via the *prev_alloc* pointers.
- oldest() – Returns the oldest memory cell that is still in use (i.e. the one allocated furthest in the past, that has not yet been reclaimed).
- newest() – Returns the newest memory cell that is still in use (i.e. the one allocated the most recently, that has not yet been reclaimed).

Questions:

a) In this system, all allocated memory cells are joined together in a single large linked list, via the *prev_alloc* pointers. The only memory cells that can refer to other memory cells are ones containing cons-pairs.

What constraints do we have about what memory cells a given cons-pair cell can reference? Can it reference any allocated cell, or are there any constraints on which cells it may reference? Justify your answer. *(4 points)*

b) Design a simple algorithm for performing mark-and-sweep-type garbage collection in this system. You may assume that the program does not run while garbage collection is performed. Specify your answer at the level of pseudocode, but make sure to include comments that describe why your algorithm works. *(10 points)*

You may also assume the runtime has a function is_root(*cell*) that returns true if the memory cell is a "root" in the reachability graph; i.e. it is a function argument or local variable in a currently-running function, or it is a global value that was initialized at startup.

Hint: An optimal solution should be able to make only a single pass through the memory, performing both marking and reclamation at the same time.

c) Since this algorithm still performs "stop the world" garbage collection, we would like such "stop the world" pauses to be bounded in length. Does your algorithm from part (b) have the ability to be interrupted and resumed? (By this we mean that the algorithm can pick up where it left off, and it will not reclaim memory cells that are actually reachable. It is fine if garbage is not reclaimed immediately.) If so, what will be the consequences of pausing and resuming the algorithm? If not, why is your algorithm unable to handle this? *(4 points)*