CS24: Introduction to Computing Systems

Spring 2015

Lecture 12

CS24 Midterm

- Midterm format:
  - 6 hour overall time limit, multiple sittings
    - (If you are focused on midterm, clock should be running.)
  - Open book/notes/slides/homeworks/solutions
  - Open CS24 2014 Moodle, closed people/Internet/etc.
  - Can use computer to implement and test your work (but not disassemble or generate IA32 code from C)

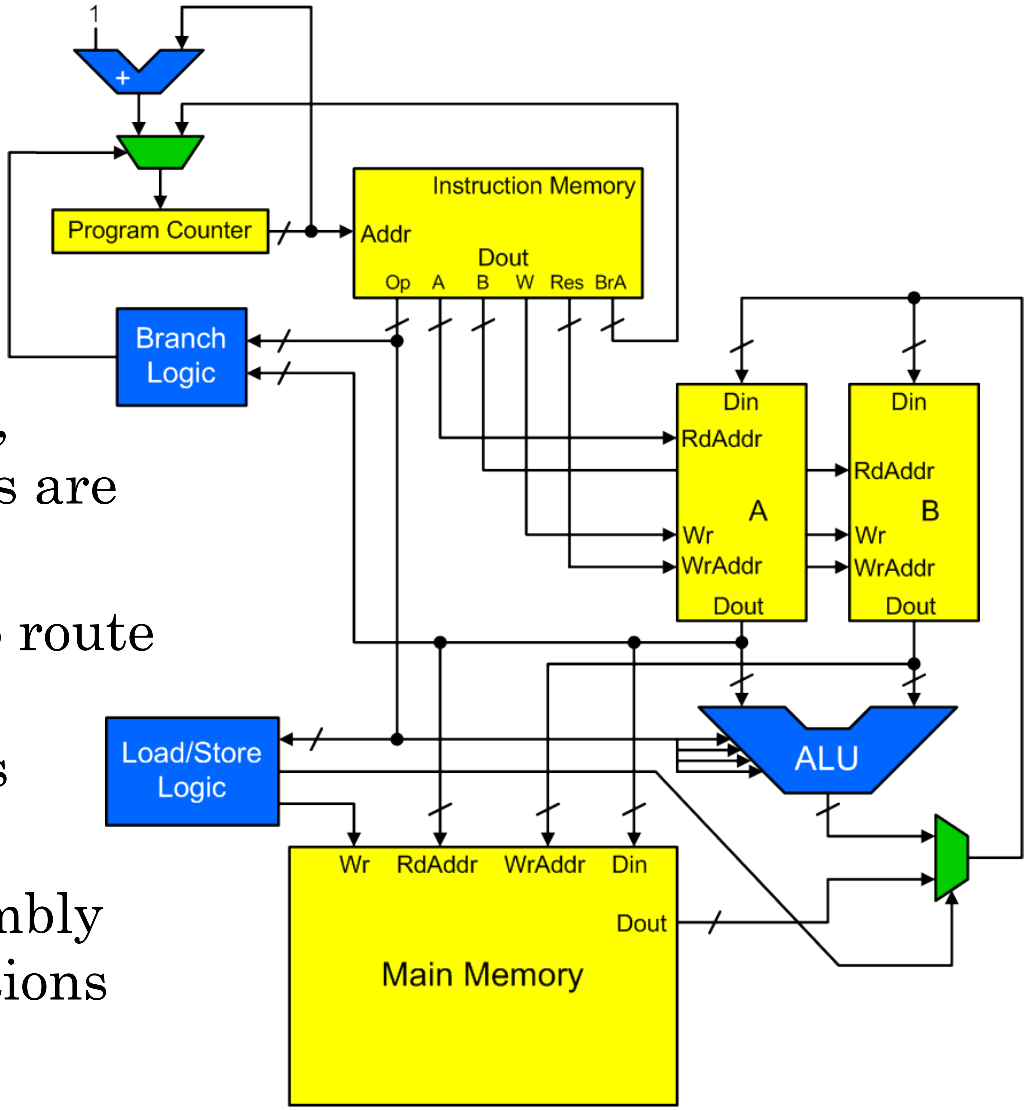
- **No collaboration!**
  - If you need clarifications, feel free to email Donnie and/or **`cs24tas`** list (although Donnie has final say)

CS24 MIDTERM (2)

- Potential topics include:
  - **Anything covered in first half of class**
  - Basic processor structure, operation, low-level programming
  - Boolean logic, logical/bitwise operations in C
  - IA32 assembly language programming
  - Flow-control constructs in IA32 – **`if`**, **`for`**, **`while`**, **`do`**
  - Arrays, structs, heap allocation (**`malloc`**/**`free`**)
  - Explicit heap allocator internals
  - Garbage collection, objects, exceptions

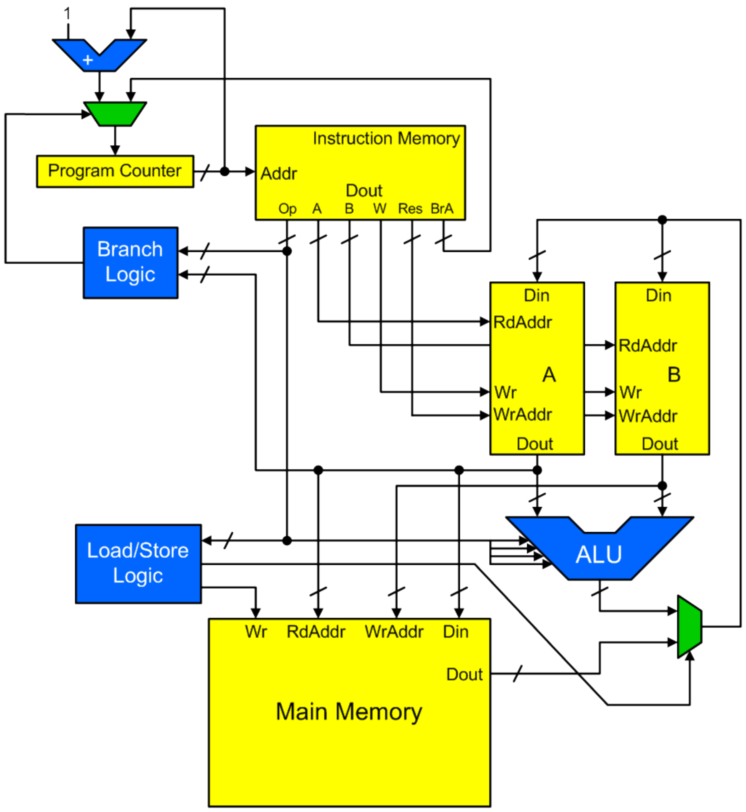
Processor Structure

- Covered basic CPU components in lectures 1-4
- General idea:
  - Based on the opcode, different components are enabled or disabled
  - Multiplexers used to route address/data signals between components
  - Feed a sequence of instructions to assembly to perform computations



Processor Structure (2)

- Midterm will present a simple variant for you to examine
- Example questions:
  - Write C code to simulate the logic that controls components
  - Write a simple machine-code program to control the CPU
  - Analyze characteristics (i.e. strengths and weaknesses) of the alternate design
- (Similar to problems on HW1)



C Logical and Bitwise Operations

- Should be familiar with all C logical and bitwise operations
- C uses integers to represent Boolean values
  - 0 = false; any nonzero value = true
- Logical Boolean operators:
  - Logical AND: `a && b`
  - Logical OR: `a || b`
  - Logical NOT: `!a`
  - Result is 1 if true, 0 if false
- `&&` and `||` are short-circuit operators
  - Evaluated left-to-right
  - For `&&`, if LHS is false then RHS is not evaluated
  - For `||`, if LHS is true then RHS is not evaluated

C Logical and Bitwise Operations (2)

- Bitwise operators manipulate individual bits
- Given a = 00010100_2 (20_{10}), b = 00110010_2 (50_{10})
  - `a & b` = 00010000 Bitwise AND
  - `a | b` = 00110110 Bitwise OR
  - `~a` = 11101011 Bitwise negation (invert)
  - `a ^ b` = 00100110 Bitwise XOR
- Also, bit-shift operators
  - `a >> 2` = 00000101 (5_{10})
  - `a << 1` = 00101000 (40_{10})
- Shift-right has two versions!
  - Arithmetic shift-right preserves sign (topmost bit)
  - Logical shift-right doesn't preserve sign
  - In C, <u>type of LHS</u> determines if arithmetic or logical

IA32 Assembly Language

- Midterm will include problems writing and/or understanding IA32 assembly programs
- Be familiar with IA32 execution environment

  Lecture 4

  - Registers: **`eax`**, **`ebx`**, **`ecx`**, **`edx`**, **`esp`**, **`ebp`**, **`esi`**, **`edi`**
  - Stack pointer in **`esp`**; stack grows downward
  - Set up stack frame using **`ebp`**
- Also, be familiar with cdecl calling convention!

  Lecture 5

  - Caller-save registers: **`eax`**, **`ecx`**, **`edx`**
  - Callee-save registers: **`ebp`**, **`ebx`**, **`esi`**, **`edi`**
  - Arguments passed on the stack: **`ebp`** + *offset*
  - Local variables stored on the stack: **`ebp`** – *offset*
  - Return-value passed back in **`eax`**

Calling C Functions from IA32

Lecture 7

- Be familiar with how to call standard C functions from IA32, e.g. **`malloc()`**
- Important note: these functions also use cdecl ☺
  - Save any caller-save registers you want to preserve!
  - e.g. **`malloc()`** may change **`eax`**, **`ecx`**, **`edx`**
  - If you need them, save them on the stack!
  - *…before you push your arguments to malloc…*

Writing IA32 Assembly Programs

- Very helpful to follow a specific pattern when implementing IA32 functions

```
my_function:
    # TODO:  Document arguments and offsets here
    pushl   %ebp          # Save caller ebp
    movl    %esp, %ebp    # Set up stack frame ptr

    # TODO:  Save callee-save registers we alter

    # TODO:  Function body

    # TODO:  Restore callee-save registers we alter

    # Clean up stack before returning to caller
    movl    %ebp, %esp    # Remove local vars
    popl    %ebp          # Restore caller ebp
    ret
```

Writing IA32 Assembly Programs (2)

- Usually need to set up and tear down stack frame

```
my_function:
    # TODO:  Document arguments and offsets here
    pushl    %ebp          # Save caller ebp
    movl     %esp, %ebp    # Set up stack frame ptr

    # TODO:  Save callee-save registers we alter

    # TODO:  Function body

    # TODO:  Restore callee-save registers we alter

    # Clean up stack before returning to caller
    movl     %ebp, %esp    # Remove local vars
    popl     %ebp          # Restore caller ebp
    ret
```

Writing IA32 Assembly Programs (3)

- Figure out where your arguments are stored
  - cdecl specifies that args are pushed in reverse order!

```
my_function:
    # TODO:  Document arguments and offsets here
    pushl   %ebp           # Save caller ebp
    movl    %esp, %ebp     # Set up stack frame ptr
    ...
```

- Example from lecture 7: vector-add function

```
vector_add:
    # a      =  8(%ebp)              (Arg 1, pushed last)
    # b      = 12(%ebp)              (Arg 2)
    # length = 16(%ebp)              (Arg 3, pushed first)
    pushl   %ebp           # Save caller ebp
    movl    %esp, %ebp     # Set up stack frame ptr
```

Writing IA32 Assembly Programs (4)

- Easiest to write function body in pseudocode, then translate to IA32

```
my_function:
    ...
    # TODO:  Save callee-save registers we alter

    # TODO:  Function body

    # TODO:  Restore callee-save registers we alter
    ...
```

- No point in saving/restoring registers until you know what you use in the implementation…
  - Fill in those parts last
  - *(Or, just save them all, but we can do better…)*
- Return-value in **`eax`**

IA32 and C Flow-Control Constructs

Lecture 6

- Need to be familiar with how C flow control constructs are implemented in IA32 assembly
  - e.g. given a C program that uses **`if`**, **`for`**, **`while`**, **`do`**, be able to translate into IA32 assembly with labels and conditional jumps
  - At very least, can take a problem, write it in C-style pseudocode, then translate into IA32
- IA32 conditional jump operations are based on contents of **`eflags`** register
  - Most relevant flags:
    - CF = carry flag (1 indicates unsigned overflow)
    - SF = sign flag (1 = result is negative)
    - OF = overflow flag (1 indicates signed overflow)
    - ZF = zero flag (1 = result is zero)

IA32 Conditional Jumps

- Arithmetic and logical operations update **`eflags`**
  - Sometimes can use an arithmetic operation to set up for a conditional jump
- Can also update **`eflags`** with **`cmp`** and **`test`**
  - **`cmp Src2, Src1`**
    - Updates flags as for Src1 – Src2 (i.e. **`sub Src2, Src1`**)
  - **`test Src2, Src1`**
    - Updates flags as for Src1 & Src2 (i.e. **`and Src2, Src1`**)
  - Src1, Src2 unchanged by comparison/test operation
  - Specify size prefixes, as usual: **`cmpl %ecx, $0`**
- **Most important detail:** AT&T syntax imposes confusing order on compare operands!

IA32 Conditional Jumps (2)

- A sometimes-useful trick: **`test Src, Src`**
  - Comparing a value against itself
- Easy way to tell if a value is positive, negative, or zero
  - Causes the Sign Flag and Zero Flag to be set based on Src
  - SF = 1 Value is negative
  - SF = 0, ZF = 1 Value is zero
  - SF = 0, ZF = 0 Value is positive

FOR-LOOP IN IA32

- Remember: **`for`** loop is a **`while`** loop with extra features

```
for (i = 0; i < length; i++)
    result[i] = a[i] + b[i];
```

  - Identical to:

```
i = 0;
while (i < length) {
     result[i] = a[i] + b[i];
     i++;
}
```

- If you start with C code or pseudocode:
  - Can manually perform these transformations to get to IA32 assembly language

For-Loop in IA32 (2)

- Also turn **`while`** loops into **`do`** loops to simplify coding

```
i = 0;
while (i < length) {
     result[i] = a[i] + b[i];
     i++;
}
```

  - Identical to:

```
    i = 0;
    if (!(i < length))  // Factor out first test
        goto end_for;
start_for:
    result[i] = a[i] + b[i];
    i++;
    if (i < length)     // Subsequent tests at end
        goto start_for;
end_for:
    ...
```

For-Loop in IA32 (3)

- …and then the tricky part: figuring out **`cmp`**
- C code:

```
if (!(i < length))    /* Or i >= length */
        goto end_for;
```

- **`cmpl Src2, Src1`** sets flags as for Src1 – Src2
  - Specify **`length`** as Src2, **`i`** as Src1 so that appropriate conditional jump instruction mirrors the C code
- IA32 code: (**`i`** in **`%esi`**, **`length`** at **`16(%ebp)`**)

```
cmp 16(%ebp), %esi    # if (i >= length)
jge end_for           #     goto end_for;
```

- First arg in compare is Src1, second arg is Src2
  - Args will appear in reverse order, but the jump will more closely match your C code or pseudocode

Signed and Unsigned Comparisons

- IA32 includes different instructions for signed and unsigned comparisons!
- Signed: jump if greater/less
  - **`jg`**, **`jge`** – jump if greater, jump if greater or equal
  - **`jl`**, **`jle`** – jump if less, jump if less or equal
  - Instructions examine sign flag, overflow flag, and zero flag to determine signed comparison result
- Unsigned: jump if above/below
  - **`ja`**, **`jae`** – jump if above, jump if above or equal
  - **`jb`**, **`jbe`** – jump if below, jump if below or equal
  - Instructions examine carry flag and zero flag to determine unsigned comparison result

IA32 Arithmetic/Logical Operations

- IA32 provides a variety of arithmetic and logical operations
  - Addition, subtraction, multiplication, division, increment, decrement
  - AND, OR, NOT, XOR, shift left, arithmetic shift right, logical shift right
- Can specify a variety of operand types
  - Source argument can be Immediate, Register, Memory (direct or indirect)
- Destination argument can be:
  - Register, Memory (direct or indirect)
- Both arguments cannot be a memory reference

IA32 Multiply Operations

- IA32 has a *wide variety* of multiply operations!
  - **`imul`** = signed multiply
  - **`mul`** = unsigned multiply
- **`mul`** only has a 1-operand form
  - Multiply **`al`**, **`ax`**, or **`eax`** by the operand
  - Store result in **`ax`**, **`eax`**, or **`edx:eax`**, respectively
    - **`edx`** contains high dword, **`eax`** contains low dword
- **`imul`** has 1-, 2-, and 3-operand forms!
  - One-operand form is identical to **`mul`**
  - 2- and 3-operand forms don't implicitly use **`al/ax/eax`**
    - Result also truncated to same bit-width as arguments

IA32 Divide Operations

- IA32 has two division instructions
  - **`idiv`** = signed division
  - **`div`** = unsigned division
  - Only one form for each: takes one operand
- Instruction divides **`ax`**, **`dx:ax`**, or **`edx:eax`** by the argument
  - Produces both quotient and remainder
- Example: dividing **`edx:eax`** by some value
  - Quotient is stored in **`eax`**
  - Remainder is stored in **`edx`**

C, IA32, and Arrays

Lecture 6

- Need to be familiar with arrays in C and IA32
- For an array declaration: ***T* A[*N*]**
  - ***T*** is the data type
  - **A** is the array's variable name
  - ***N*** is the number of elements
- C allocates a contiguous region of memory for array
  - Allocates ***N*** × **sizeof(*T*)** bytes for the array
- Variable **A** holds a <u>pointer</u> to start of array
  - Can use **A** to access various elements of the array
- Can access elements of **A** using pointer arithmetic
  - e.g. **A[i]** is equivalent to ***(A + i)**
  - **A + i** moves pointer forward **i** elements, not **i** bytes
  - C figures out how large each element is, from type of **A**

C, IA32, and Arrays (2)

- IA32 memory addressing modes make it *easy* to access arrays from assembly language routines
- Indexed memory access
  - **`(RegB, RegI)`** accesses M[RegB + RegI]
    - RegB is the base (i.e. starting) address of a memory array
    - RegI is an index into the memory array
  - **`Imm(RegB, RegI)`** accesses M[Imm + RegB + RegI]
  - Assumes that array elements are bytes
- Scaled indexed memory access
  - With scale factor s = 1, 2, 4, 8 (i.e. size of array element):
  - **`(, Reg, s)`** M[Reg × s]
  - **`Imm(, Reg, s)`** M[Imm + Reg × s]
  - **`(RegB, RegI, s)`** M[RegB + RegI × s]
  - **`Imm(RegB, RegI, s)`** M[Imm + RegB + RegI × s]

IA32 Memory-Reference Summary

- Summary chart, from IA32 manual:

Base	+	Index	×	Scale	+	Displacement
`eax` **`ebx`** **`ecx`** **`edx`** **`esp`** **`ebp`** **`esi`** **`edi`**	+	**`eax`** **`ebx`** **`ecx`** **`edx`** *(not **`esp`**!)* **`ebp`** **`esi`** **`edi`**	×	1 2 4 8	+	None 8-bit 16-bit 32-bit

  - Base, Index, Displacement all optional
  - Scale only allowed when Index is specified
  - Note that **`esp`** can only be used as a base value

C, IA32, and Arrays (3)

- Again, our vector-add function loop-body:

```
result[i] = a[i] + b[i];
```

- IA32 assembly code:

```
vadd_loop:
    movl (%ebx, %esi, 4), %edx    # edx  = a[i]
    addl (%ecx, %esi, 4), %edx    # edx += b[i]
    movl %edx, (%eax, %esi, 4)    # r[i] = edx
```

- Our arrays contain **`int`** elements
  - **`sizeof(int)`** = 4 bytes, so scale = 4
  - **`ebx`** = starting address of array **`a`**
  - **`ecx`** = starting address of array **`b`**
  - **`eax`** = starting address of array **`result`**
  - **`esi`** = array index **`i`**

C, IA32, and Structs

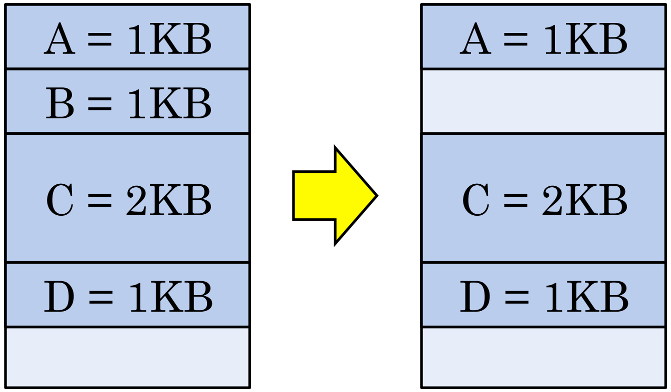
- IA32 addressing modes also make it easy to work with structs
- From lecture 8:

```
struct s1 {
    int i;
    char ch;
    int j;
};
```

- If **s1 *s** is stored in **%ebx**
  - **s->i** **movl (%ebx), %eax**
  - **s->ch** **movb 4(%ebx), %cl**
  - **s->j** **movl 8(%ebx), %edx**

HEAP ALLOCATORS

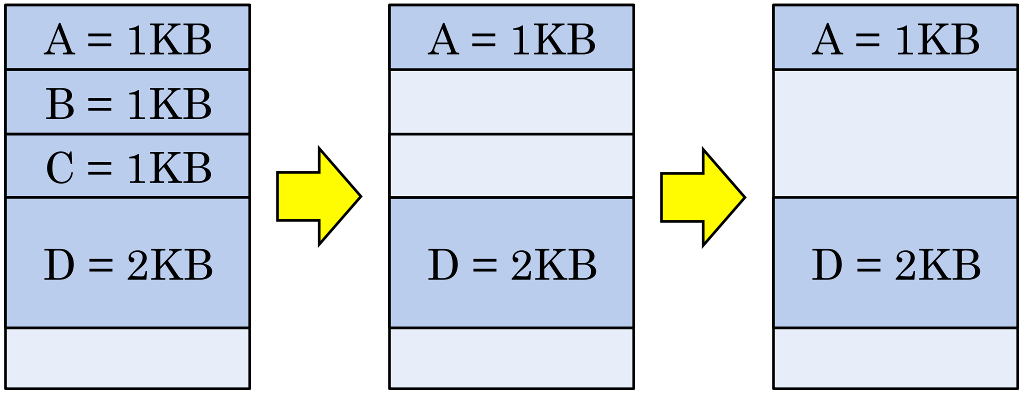
- Explicit allocators:
  - Program is responsible for allocating/freeing memory
- Memory fragmentation:
  - Requested amount of memory is available, but no block is actually large enough to satisfy the request
- Example: heap with 6KB total space
  - A = allocate 1KB
  - B = allocate 1KB
  - C = allocate 2KB
  - D = allocate 1KB
  - Free B



- Try to allocate 2KB?
  - 2KB of memory is available, but it's not contiguous!

Coalescing Free Blocks

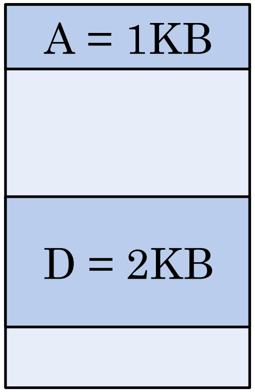
- Explicit allocators coalesce adjacent free blocks
  - A = allocate 1KB
  - B = allocate 1KB
  - C = allocate 1KB
  - D = allocate 2KB
  - Free C
  - Free B
  - Try to allocate 2KB?



- Very important for avoiding false fragmentation!
  - In above example, allocator will coalesce B with C when B is freed

Placement Policies

- Allocators employ a placement policy to choose which free block to use for a request
- Example: want to allocate 1KB
  - Two different free blocks to choose from
- First-fit policy: choose the first free block that satisfies the request
- Next-fit policy: like first-fit, but start searching where previous search ended
- Best-fit policy: examine all free blocks; choose the smallest free block that satisfies the request
- For our example:
  - First-fit policy would cause memory fragmentation
  - Best-fit policy avoids fragmentation



Implicit Allocators

- C language is very powerful, but also allows us to do many dangerous things
  - Buffer overflows, exploits, memory leaks!
- Introduce higher-level abstractions:
  - Eliminate pointers and pointer arithmetic
  - An opaque "reference" to allocated objects
  - Use an implicit allocator to manage memory
- Implicit allocators use garbage collection to find unreachable blocks
  - Mark-and-sweep GC vs. stop-and-copy GC
  - (also reference-counting, but is vulnerable to cycles)

Objects, Exception Handling

- Additionally, can provide more powerful programming abstractions
  - Object-oriented programming
  - Exception handling
- Examined **`setjmp()`** and **`longjmp()`**
  - Still works within cdecl, but enables *non-local jumps*
  - Implements an alternate flow-control mechanism by manipulating the stack in well-defined ways
- Midterm may also include topics related to these mechanisms
  - **Familiarity with cdecl, stack frames, etc. will be very important.** *(But, you need that anyway.)*

Final Midterm Notes

- Midterm will be available towards end of week
  - Due Thursday, May 7 at 2AM, as usual
- Solution sets for all assignments will be available before midterm is released
  - **Please review the solutions before taking the exam!** I frequently give more background detail in the solution sets.
- Good luck! ☺