ECE 391 Exam 1 Review Session - Spring 2018

Brought to you by HKN

DISCLAIMER

There is A LOT (like a LOT) of information that can be tested for on the exam, and by the nature of the course you never really know what you'll be tested on. We're basing this review session off of the poll on Piazza to help you guys with the material that will most likely be on the exam, but there is a large possibility that there you will be tested on material we do not cover. Please be advised that you should still go over material on your own, and go to office hours to get TAs to help you.

x86 - Brief Overview (Reference sheet is included on the exam)

- `$LABEL` - literal value, `LABEL` - memory address
  - `leal LABEL, %edx == movl $LABEL, %edx`
- Can't have more than one memory access per instruction
  - e.g. `movl (%eax), (%ebx)`
- Memory is stored little-endian
- Comparisons
  - signed: `jl` (lower), `jg` (greater)
  - unsigned: `jb` (below), `ja` (above)
    - `cmpl %esi, %edi; jge LABEL`
    - Performs the (signed) comparison `%edi ≥ %esi`

32-bit	16-bit	8-bit high	8-bit low
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI	SI		
EDI	DI		
EBP	BP		
ESP	SP		

AX
31 | 16 15 | 8 7 | 0
AH | AL
EAX

`jb`	below	CF is set
`jbe`	below or equal	CF or ZF is set
`je`	equal	ZF is set
`jl`	less	SF = OF
`jle`	less or equal	(SF = OF) or ZF is set
`jo`	overflow	OF is set
`jp`	parity	PF is set (even parity)
`js`	sign	SF is set (negative)

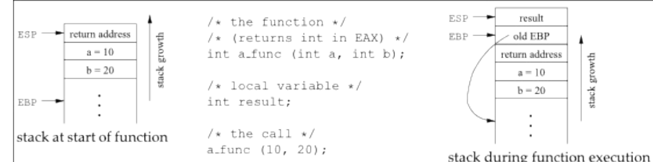
```
movb  (%ebp),%al              # AL ← M[EBP]
movb  -4(%esp),%al            # AL ← M[ESP - 4]
movb  (%ebx,%edx),%al         # AL ← M[EBX + EDX]
movb  13(%ecx,%ebp),%al       # AL ← M[ECX + EBP + 13]
movb  (,%ecx,4),%al           # AL ← M[ECX * 4]
movb  -6(,%edx,2),%al         # AL ← M[EDX * 2 - 6]
movb  (%esi,%eax,2),%al       # AL ← M[ESI + EAX * 2]
movb  24(%eax,%esi,8),%al     # AL ← M[EAX + ESI * 8 + 24]
movb  100,%al                 # AL ← M[100]
movb  label,%al               # AL ← M[label]
movb  label+10,%al            # AL ← M[label+10]
movb  10(label),%al           # NOT LEGAL!

movb  label(%eax),%al         # AL ← M[EAX + label]
movb  7*6+label(%edx),%al     # AL ← M[EDX + label + 42]

movw  $label,%eax             # EAX ← label
movw  $label+10,%eax          # EAX ← label+10
movw  $label(%eax),%eax       # NOT LEGAL!

call  printf                  # (push EIP), EIP ← printf
call  *%eax                   # (push EIP), EIP ← EAX
call  *(%eax)                 # (push EIP), EIP ← M[EAX]
call  *fptr                   # (push EIP), EIP ← M[fptr]
call  *10(%eax,%edx,2)        # (push EIP), EIP ←
                              #         M[EAX + EDX*2 + 10]
```

Conditional branch sense is inverted by inserting an "N" after initial "J," *e.g.*, JNB. Preferred forms in table below are those used by debugger in disassembly. Table use: after a comparison such as

```
cmp %ebx,%esi   # set flags based on (ESI - EBX)
```

choose the operator to place between ESI and EBX, based on the data type. For example, if ESI and EBX hold unsigned values, and the branch should be taken if ESI ≤ EBX, use either JBE or JNA. For branches other than JE/ JNE based on instructions other than CMP, check the branch conditions above instead.

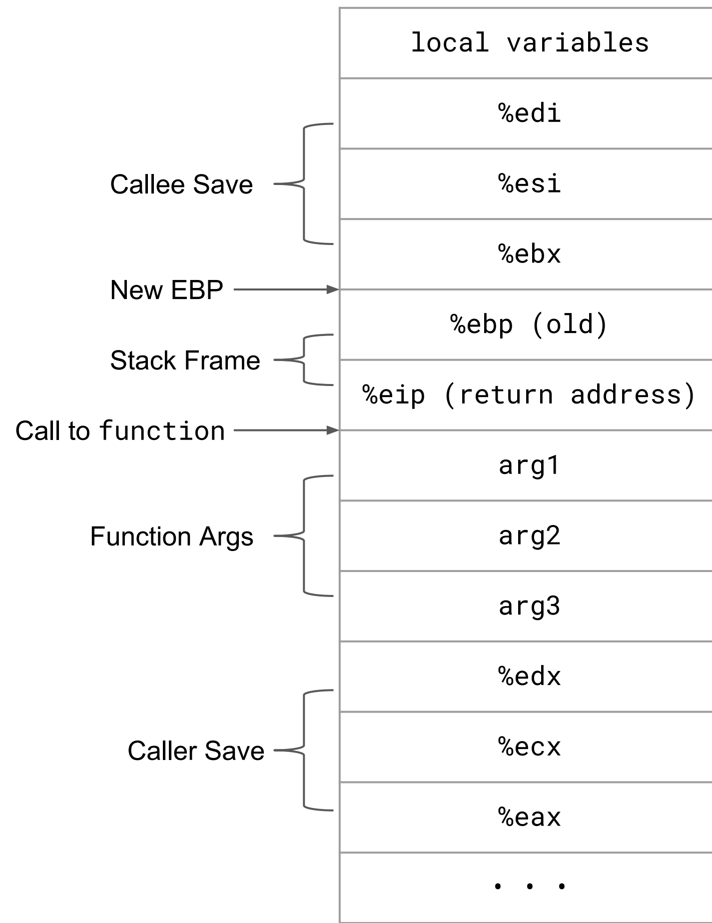
	`jnz`	`jnae`	`jna`	`jz`	`jnb`	`jnbe`	unsigned comparisons
preferred form	`jne`	`jb`	`jbe`	`je`	`jae`	`ja`	
	=	<	≤	=	≥	>	
preferred form	`jne`	`jl`	`jle`	`je`	`jge`	`jg`	signed comparisons
	`jnz`	`jnge`	`jng`	`jz`	`jnl`	`jnle`	



x86/C Calling Conventions

- Caller save registers - EAX, ECX, EDX
- Callee save registers - EBX, ESI, EDI
- call vs jump:
  - jump → `jmp LABEL`
  - call → `pushl %eip; jmp LABEL`
- enter - `pushl %ebp; movl %esp, %ebp`
  - “creates” the stack frame
- leave - `movl %ebp, %esp; popl %ebp`
  - “tears down” the stack frame
- ret - `popl %eip`
- Push arguments from right to left (why?)

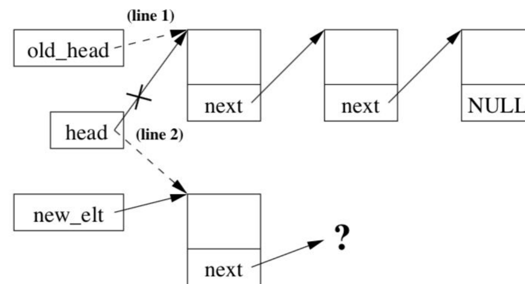
*We’ll go over a translation question later



Synchronization Part 1

- Sharing data structures between program and interrupt handlers
  - Linked list example

```
/* line 1 */ old_head = head;
/* line 2 */ head = new_elt;
             /* INTERRUPT OCCURS HERE! */
/* line 3 */ new_elt->next = old_head;
```



Synchronization Part Dos

```
volatile int ready = 0;
             while(!ready);
```

Synchronization Part III

- Critical sections → code that runs without being interrupted
  - For single processor machines, use the interrupt flag to accomplish this (drawbacks to using this)
- For multiprocessors, we need more:
- Introducing...the SPIN LOCK
  - spin_lock→ doesn’t push the flags to the stack before entering critical section
  - spin_lock_irqsave→ pushes the flags before entering critical section
  - Both however clear the IF flag (why?)

Synchronization Part IV

- Semaphores
  - Up/Down operations
  - Process goes to sleep giving other processes access to the CPU
  - Can be used to protect longer critical sections
- Reader/Writer Spinlocks
  - Can cause writer starvation
- Reader/Writer Semaphores
  - Helps prevent starvation

Synchronization Example Problem

There is a laboratory which can be occupied by either zombies or scientists. Now you are hired to design a system that satisfies the following rules:

• At all times, the lab must either be empty, contain scientists, or contain zombies. At most **10** zombies or **4** scientists can stay in the lab, but there is NO limit for the number of zombies/scientists waiting in line.

• If there are scientists occupying the lab, zombies must wait for all scientists to leave before entering. Similarly, if zombies occupy the lab, scientists must wait for them to leave before they can enter.

• While waiting in line, scientists have higher priority. This means that if there are scientists waiting to enter, no zombie can enter the lab (even if the lab is currently occupied by zombies). You can think of this as scientists always being moved to the front of the line. Note that when a scientist joins the line and the lab is occupied by zombies, the scientist still must wait for the zombies in the lab to leave; scientists cannot force zombies out of the lab.

• There is no need to enforce priority among zombies or among scientists (ie. if A arrives before B, A does not necessarily enter the lab before B)

• When zombies occupy the lab, they introduce contamination. To avoid contaminating the scientists' work, the lab must be sanitized before scientists can enter a lab previously occupied by zombies. No sanitization need be done when switching from scientists to zombies.

Sanitization must be done as infrequently as possible, and must only be done when the lab is empty. You are to implement a thread safe synchronization library to enforce the lab occupancy policy described above.

Example Problem Continued

- A set of enter functions will be called by scientists and zombies to enter the lab. These functions should behave like a “lock,” and do not return until the lab has been successfully entered by the caller. If any errors occur, return -1. Otherwise, return 0.

- When a scientist or zombie wants to leave the lab, they call their leave function to let your library know they have left.

- You may use only one spinlock in your design, and no other synchronization primitives may be used.

- As these functions will be part of a thread safe library, they may be called simultaneously

- Write the code for enter and exit of scientist/zombie. Assume that these functions may be called simultaneously from multiple threads.

Interrupts, Exceptions, System Calls, and Tables!

type	generated by	example	asynchronous	unexpected
interrupt	external device	packet arrived at network card	yes	yes
exception	invalid opcode or operand	divide by zero	no	yes
system call/trap	deliberate, via INT instruction	print character to console	no	no

These guys interrupt the regular flow of a program, and are used to:

- Deal with something that requires urgent attention now (interrupt). This can usually be masked if we don't wanna (or can't) deal with that stuff.
- Tell us what to do when unexpected bad stuff happens (exception)
- Let the kernel, higher-privileged code, execute some instruction or carry out some task for us that we can't do ourselves (system call/trap)

Every time we get an interrupt, exception, or system call, we jump to a 256-entry vector table called the Interrupt Descriptor Table (IDT) to handle them. You can find the table in lecture slides/notes.

- Entries in the table from 0x00-0x1F are reserved and defined by Intel, more later in course
- Single entry (0x80) for all system calls. Privilege and stuff matters here, more later in course

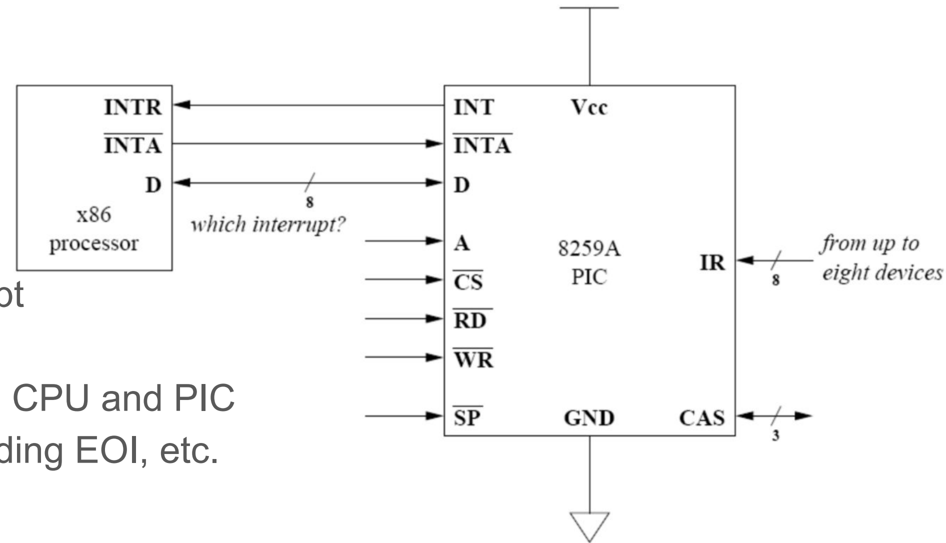
Programmable Interrupt Controller

Useful for handling multiple interrupts from different devices, can't just stick all the interrupts onto a single bus and expect that to work. The PIC allows us to prioritize, mask, and individually address different interrupts that get raised.

- Each PIC handles 8 interrupts, but they can be configured in a master-slave configuration (with one master, and up to 8 other slaves) to handle up to 64 different interrupts. Next slides will go more in-depth
- After the processor (our point of view) receives an interrupt from the PIC, it calls ack(nowledge) function to acknowledge receipt and masks all lower-priority interrupts, immediately sends an End-Of-Interrupt (EOI), then calls end function when done handling the interrupt to unmask lower-priority interrupts.
    - This lets interrupts continue to build up in a queue so we can handle them later.

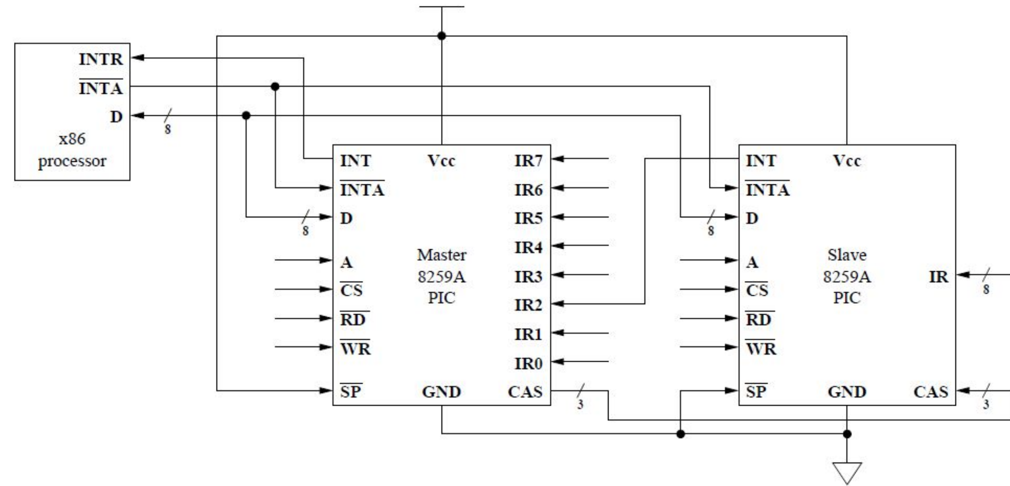
PIC Functionality

- Memory mapped to two ports
  - Command port (e.g. 0x20)
  - Data port (MUST be Command Port + 1)
- CPU - PIC Signals
  - INTR - Activated by the PIC upon interrupt
  - INTA - Pulsed by the CPU whenever
  - D - Bidirectional communication between CPU and PIC
    - Used in programming the PIC, sending EOI, etc.
- PIC Specific Signals
  - A - Distinguishes Command/Data port on PIC
    - Can be directly mapped to ADDR[0] (why?)
  - CS - Determines whether the given PIC should be active
    - Checks if ADDR[31:1] == PORT[31:1]
- Priority: IR0, IR1, IR2, … , IR7



PIC Cascading (i.e. Master/Slave configuration)

- Two Level Hierarchy
- SP - Decides whether a PIC is master or slave
  - 1 = Master, 0 = Slave
- CAS - How master PIC decides the slave
  - Width is determined by number of slaves the PIC can support (basically the number of IR lines)



- Given this info, why must it be a two-level hierarchy?
- If a slave is attached, put all its priorities on the level where it's connected
  - e.g. slave on IR3, the total hierarchy is M0, M1, M2, **S0, S1, S2, … , S7**, M4, … , M7

PIC Initialization (1/2)

5 Key steps

Lock and save flags (context) so you can initialize properly

Mask interrupts to the PIC so you don’t get disturbed while initializing

Initialize PIC

Unmask interrupts

Unlock and restore flags

PIC Initialization (2/2)

How to actually initialize PIC? All you need to do is send 4 control words!

But what do they mean?

CONTROL WORD 1: Put PIC in initialization MODE (after this it expects the next 3 control words to come to it on a particular port)

CONTROL WORD 2: Tell PIC the start of IDT mapping

CONTROL WORD 3: Master: bitmap of slaves; Slave: input pin on master

CONTROL WORD 4: Some EOI stuff

More About Interrupts!

Interrupt Chaining:

- Don’t you wish you could handle multiple things when you get an interrupt, well now you can!
    - With interrupt chaining, multiple handlers can be triggered by one interrupt
    - Several ways to do this, but generally doesn’t happen to often in practice.

Soft Interrupts (tasklets):

- It’s not good to take too much time in a hardware interrupt handler- other interrupts may need to do things too! That’s what software interrupts are for
    - Software interrupts operate at priority level between regular programs and hardware interrupts, so hardware interrupts can generate a software interrupt to handle more time-intensive tasks, allowing other hardware interrupts to interrupt the software interrupts

Anything else??