

3/27	L26	<p>Stack Pointer, with all necessary state saved on the stack (but don't push any unnecessary state). We could include local variables, but I've left them out for simplicity.</p> <ul style="list-style-type: none"> Complete CS310 Stack Frame: two pointers: Stack pointer and the Frame Pointer. Once the stack frame is built, all accesses (in the function body) to the state is based on the frame pointer. 	Handout #50 P/P 14.3		
3/30	D10	<p>Discuss: Compiler Templates for method invocation:</p> <ul style="list-style-type: none"> warm-up: use registers when possible, otherwise use static locations CS310 simple stack frame ?CS310 complete stack frame 	function calls: Handout #50 P/P 14.3		
		<p>Handout #53: Homework 7 Directions Handout #54: Homework 7 LC3 code Handout #55: Supplement on a compiler template for a RISC ISA (for fun only) Handout #56: Partial Slides on HLL Variables (not online) Handout #57: Partial Slides on Stack Buffer Overflow (ignore slides 7-9 diagrams) Handout #58: Partial Slides for Next Topic: ISA Addressing Modes (not on exam2)</p> <p>Compiler: Wrap-up Compiler Templates for a Function Call HLL variables</p> <ul style="list-style-type: none"> Wrap-up Function Calls: <ul style="list-style-type: none"> Note, a compiler for an LC3 function 'foo' doesn't know what registers the caller may be using. However, RISC ISAs have so many registers that compilers divide them up into registers that must be callee-saved (if used), and others that must be caller-saved (if used). See Handout #55. Why do some compilers use two pointers (frame pointer and stack pointer); reinforce the ease of recursion Hmwk 7 overview 			
END OF EXAM 2 TOPICS					

3/31	L29	<p>Memory Layout: memory layout of a user process and the OS: each have a code section, global data section, run-time stack and the heap.</p> <p>In which 3 sections of MEMORY are all the HLL variables stored?</p> <ul style="list-style-type: none"> (Of course, some HLL variables can be stored into a "register only" i.e. no need for the variable to be assigned to a memory location. This only works for a single valued variable that fits into the register.) 1) The run-time stack for Local Variables: Locals are accessed via the "frame pointer" (or the "stack pointer"). The compiler (via its symbol table) will keep track of all local variables (or the local "reference") and assign them offsets from the "frame pointer" (R5). A C-style array is a contiguous area of memory, and the compiler knows exactly how much memory the array will require (thus the array can be on the run-time stack). 2) The global data section for Global Variables: Globals (and static locals) are accessed via the Global Data Pointer (R4). The compiler (via its symbol table) will keep track of all global variables and assign them offsets from the "global data pointer", even a C-style array. 3) HEAP: any memory that is allocated via the "new" function i.e. dynamically allocated memory at run-time. The "new" function is also called the "memory allocator", which will allocate the requested contiguous area of memory (based on type and if given, the array size). The return value from "new" is the base address of the allocation: what we call a "reference" in Java's OOP. <p>Compiler Symbol Table: note the difference between this and the Assembler's Symbol Table. The compiler needs to know the</p>	<p>Memory Layout Figure 12.7</p> <p>Where are HLL variables stored? Patt/Patel 12.5, 12.6.3, and 19.4 intro section. Handout #56</p> <p>Compiler Symbol Table (simplified) Patt/Patel 12.5 Handout #56</p>	H7
------	-----	---	--	----

	4/07	L32	<p>STI.</p> <ul style="list-style-type: none"> No need for this addressing mode, since it can be implemented in 2 regular instructions....however, for educational purposes, it's an easy way to access a far away scalar variable in one instruction Study the micro-architecture of LDI, STI In real world: defined differently than LC3 <p>• Other Addressing modes not in LC3</p> <ul style="list-style-type: none"> Looking at ADD instruction with one src operand that can access memory (i.e. this is NOT an example of a RISC ISA) Indexed (this addr mode is what most of you wanted for Hmwk6 question 2: the ability to add 2 registers together to get the effective address, etc...) Scaled and Autoincrement/decrement: these save 1 or more instructions (but it requires the size 'd' of operand, e.g: ADD.b, ADD.h, ADD.w, ADD.l) 	Addressing Modes: Handout #58.59 Supplement		H7 10am no late hw
12	4/10	L33	<p>Handout #66: Exam 2 Reference Sheet Handout #68: Solutions to Study Quiz 4</p> <p>Discuss: Yes, we will have class today.</p> <ul style="list-style-type: none"> Exam2 Review for most of class NEW TOPIC: Class discussion: <ul style="list-style-type: none"> Design the following "high-level" algorithms for input/output of integers: Convert between an integer and the string of chars representing the integer Converting a string of chars into an integer (for typical input processing of the integer data that you are keying in) converting an integer into a string of chars (for output displays: printing, displaying in a window,...) 			
	4/10	EX2	7-9p.m. TAY 2.106			
			Handout #67: Visual Depiction of <u>Integer</u> input/output Algorithms:			

	4/12	L34	<p>Handout: worksheet from Hmwk 7 Q4 stack frame Discuss: Stack Buffer Overflow and how it is exploited?</p> <ul style="list-style-type: none"> Class Discussion about the "problem" <ul style="list-style-type: none"> Definition of a Buffer Definition of a Stack (Local) Buffer Overflow Analyze what could be modified if the Local Buffer is overflowed? Thus, where will be machine language (code) will be stored? (where is the rest of your machine code?) ideally for an exploit, the input data should be what? Solutions to minimize the vulnerability: <ul style="list-style-type: none"> Static Analysis of Source code; Compiler will insert instructions to avoid/detect an attack; OS will avoid/detect an attack; HW will avoid/detect an attack 			Stack Buffer Overflow: Handout #0
	4/13	D12	<p>Discuss:</p> <ul style="list-style-type: none"> Practice Addressing modes Review Algorithms for input/output of multi-digit integers (Handout #67) 			
	4/14	L35	No CLASS in exchange for Exam2.			
			<p>Handout #69: A few TRAP slides Handout #70: Supplement: Bit Manipulation</p> <p>Discuss: "the trap instruction" ... which relates to the Operating Systems, ISA, and Programming:</p> <ul style="list-style-type: none"> Q: how does user-code call the OS? A: via a specialized instruction: TRAP Q: What's the difference between a: TRAP ## and a JSR <label>? Q: Why can't we use TRAP <label>? Q: Is there a benefit for using TRAP ##? A: yes, e.g. if the address of the OS function needs to be moved, then no user executables need to be modified (or re-compiled, assembled, and linked). Q: Is the OS code in the same process 			

13	4/17	L36	<p>as the user code? A: An OS process has access to key resources, that a user process does not have.... so each process must have a bit somewhere to indicate: the privileged mode. In LC-3, the OS process will be "privileged mode", (aka "supervisor mode", and an unprivileged mode is known as "user mode". The supervisor mode allows the OS to access ALL instructions, and all of memory, whereas user mode is restricted.</p> <ul style="list-style-type: none"> • Every OS has a Trap "Vector" Table in low memory to store the current address of the TRAP function. • TRAP Instruction Details: <ul style="list-style-type: none"> • Study it's micro-architecture • Programming with TRAPs: write the trap function as normal using one of the compiler function call templates, then at the beginning of your program, write the code to insert the address of the first executable instruction into the trap vector. <p>Bit manipulation Motivation</p>	Traps P/P 9.1 (skip figs 9.4, 9.5)		
	4/19	L37	<p>Handout #71: Partial Slides for today Discuss: Algorithms: Bit manipulation</p> <ul style="list-style-type: none"> • Motivation, terminology: • Bitwise operations and constructing masks to preserve fields, and at the same time either clear fields, set fields to 1, or "logically negate" fields • 3 types of Shift instructions (no shift instructions in LC-3): arithmetic shifts, logical shifts, and rotates • Two generic Bit Manipulation usages: <ul style="list-style-type: none"> ◦ extract a bit field from a word and right justify it <ul style="list-style-type: none"> ▪ algorithm 1: use only shifting ▪ algorithm 2: use shifting along with an AND-constructed mask. ◦ insert a value into a bit field without modifying the other field <p>Discuss:</p> <ul style="list-style-type: none"> • Traps: Review 	Bit Manipulation Handout #70, P/P 2.6, 2.7.1, 12.3.5	H8	
	4/20	D13				

			<ul style="list-style-type: none"> • Bit Manipulation Practice: write code to extract and insert bit fields 			
	4/21	L38	<p>Handout #72: Hmwk8 Q1 & Q2 Handout #73: Hmwk8 Q3 & Q4 Handout #75: Partial Slides on I/O Hardware and Software Discuss: I/O Hardware: How do a CPU and I/O processor interact with each other?</p> <ul style="list-style-type: none"> • I/O processor (controller) + Device electronics • Note, an I/O processor has all of the same characteristics as the CPU (PC, IR, instruction cycle, registers, ALU, ...) • Design Issue: how should the CPU name the I/O processor's "interface registers", i.e. "ports": two strategies: memory-mapped vs special IO instructions (for the Intel ISA) • I/O processors registers: generic description for a character oriented input/output device: control register, status register, input/receive data register, output/transmit data register • LC3 details of registers for the LC3 input processor for a keyboard. 	I/O Hardware P/P Chapter 8.1-8.4	H8	
14	4/24	L39	<p>Handout #74: Hmwk8 Q5 & Q6 Discuss: OS Algorithm: simple polling (busy-wait loop) vs interrupts</p> <ul style="list-style-type: none"> • Design issue: is the timing of input/output devices such that the CPU can expect it a certain time (synchronous) or variable (asynchronous) • CPU I/O algorithm #1: "polling" aka busy-wait loop • Note the use of memory indirect addressing modes is perfect for accessing specific memory-mapped locations • Interrupt motivation, focusing on the inefficiency of the "busy-wait" loop, i.e. polling the ready bit • Visual depiction of how an interrupt would work 	I/O Software (polling) P/P 8.2-8.4	Interrupt Motivation P/P p211	
			<p>Handout #76: Summary slides from discussion on Lec39 Handout #77: Slides on P/P Hardware</p>			

			<p>Additions to support Interrupts Discuss: Interrupt Hardware (mostly at the Micro-architecture)</p> <ul style="list-style-type: none"> Review how an interrupt allows the CPU to be more efficient by NOT having to poll for each I/O processor's "READY bits" Interrupts from a "Hardware" (Microarchitecture) perspective: General Interrupt ADDITIONAL concepts for all H/W <ul style="list-style-type: none"> I/O processor needs a Control Reg bit for turning on/off Interrupts (IEN bit) ANDing the IEN bit to the Status Reg bit generates an "interrupt signal" BUS needs 1 or more lines in the control bus to specify which interrupt occurred (could be the priority...) P/P Interrupt Hardware specifics: <ul style="list-style-type: none"> Priority of the interrupt (generated by Priority Encoder) Priority of the currently running program (in PSR) Priority Status Register: 3 distinct bitfields: Privilege, Priority, Condition codes Need a way to distinguish between the run-time stack for the user code, vs the supervisor code: one SP, but 2 memory locations with the address of the other stack. Add a new phase to the Instruction Cycle: Interrupt phase, as it sets the PC to the address of the ISR (Interrupt Service Routine), and saves the old PSR and the Return Address Requires a new privileged instruction (RTI) to "undo" the Interrupt phase after the ISR completes <p>Study Quiz 5: Bit Manipulation</p> <p>Discuss: Interrupt "hardware":</p> <ul style="list-style-type: none"> Finish slides Handout #77 	<p><i>I/O Interrupts, focusing on H/W P/P 8.5, 10.2, Hdt #85</i></p>				<p>4/27 D14</p>	<ul style="list-style-type: none"> Walk through 10.2.3 Interrupt H/W example What ISA change is required? (RTI) <p>Handout #78: Slides: General description of the SW architecture of a device driver! Handout #79: Time-space diagrams of the interaction between the CPU and the IO Processor using interrupts Discuss:</p> <ul style="list-style-type: none"> Review Interrupt ISA/Microarchitecture additions Exceptions/Traps/Interrupts: i.e. all events that will require a vector table entry: such as interrupts, traps, errors, along with page faults, bus error, reset, etc... Patt/Patel define the term "exception" for an exceptional event that is caused by your program (while executing). Some are fatal errors, and some are recoverable errors, but they all require a vector table entry. <p>OS Algorithm: Interrupt SOFTWARE Architecture for device drivers: using the producer-consumer model.</p> <ul style="list-style-type: none"> Split the "device driver code" into 2 halves: a trap routine that the application calls (e.g. OUT_new that doesn't poll but enqueues the characters into), and the ISR that runs whenever an interrupt occurs. The producer & consumer have to "talk to each other" via SHARED STATE: i.e. a circular queue OUT_new(char) : doesn't poll but enqueues the char into the Circular Queue OUTPUT_ISR: whenever the print char IO processor sends an interrupt, that means the IO processor is <idle> and is ready for the next character. Therefore, the ISR will dequeue a char from the Circular Queue, to be sent to the IO processor's "data register". Time-space diagram depicts this nicely. 	<p><i>Exceptions: P/P A.4</i></p> <p><i>I/O Interrupts, focusing on S/W architecture Hdts #78, 79, 85</i></p>	<p>H8</p>
									<p>Handout #80: Linking Supplement</p>		

15	5/01	L42	<p>Handout #81: Partial Slides for linking Discuss: OS components: "the linker" and "loader"</p> <ul style="list-style-type: none"> Review: Handout #3 with the picture of the IDE steps to go through from source file to your code in memory Motivation: multiple source files are translated into multiple object files, but that doesn't make a complete program HLL syntax is required in each source file to specify "exports" and "imports", i.e. external definitions available to other code (translated at distinct times) and external references Object file: Given a short source code file, what are the components of the object file (particularly focusing on the list of external references, and the list of external definitions. Linking: is also a 2-pass activity. Inputs are multiple object files and libraries. Output is an executable file. Be able to define the details of Pass1 and Pass2 (or demonstrate the details in an example) 	Linking, Loading Handout #3, #80	<u>H9</u>
	5/03	L43	<p>Handout #82: Paper copy of Hmwk 9 (extra credit) Handout #83: Exam 2 Solutions (not online) Handout #84: Linking worksheet Handout #85: Interrupt Supplement Discuss: OS components: "the linker and loader"</p> <ul style="list-style-type: none"> Linking Pass 1: what are two of the 3 key activities to build 2 data structures that will help Pass1 and Pass2, and create a single "object file" Pass 2: what action is needed to complete the missing components in the executable file? Library: definition. Distinction between a static library and a shared library. Dynamic Linking overview using a "jump table" Loading: <ul style="list-style-type: none"> 3 strategies for copying the machine code and initialized data into memory Initialize a variety of values: SP, 	Linking, Loading Handout #80	

			Heap Ptr, FP, Global Data Pointer, and finally the last value to be set: the PC = "Start Address"		
	5/04	D15	Discuss: Linker	Handout #84: Linking worksheet	
	5/05	L44	<p>Handout #86: Hmwk 8 partial solutions (not online) Handout #87: A few more practice questions Handout #88A: Final Exam Overview Handout #88B: Final Exam Detailed Objectives/Reading</p> <p>Discuss: 0) Review of Memory Layout and In which 3 sections of MEMORY are all the HLL variables stored? (see Handout #56 and the L29 description and reading) 1) Exam Objectives 2) Study Quiz 6: 3) Class Evaluation</p>	Handout #56 (and see L29 description and the reading)	H9
	5/11	F1	THURSDAY 9a.m. - noon: comprehensive final, TAY 2.106		
	5/13	F2	SATURDAY 7p.m. - 10p.m: comprehensive final, TAY 2.006		