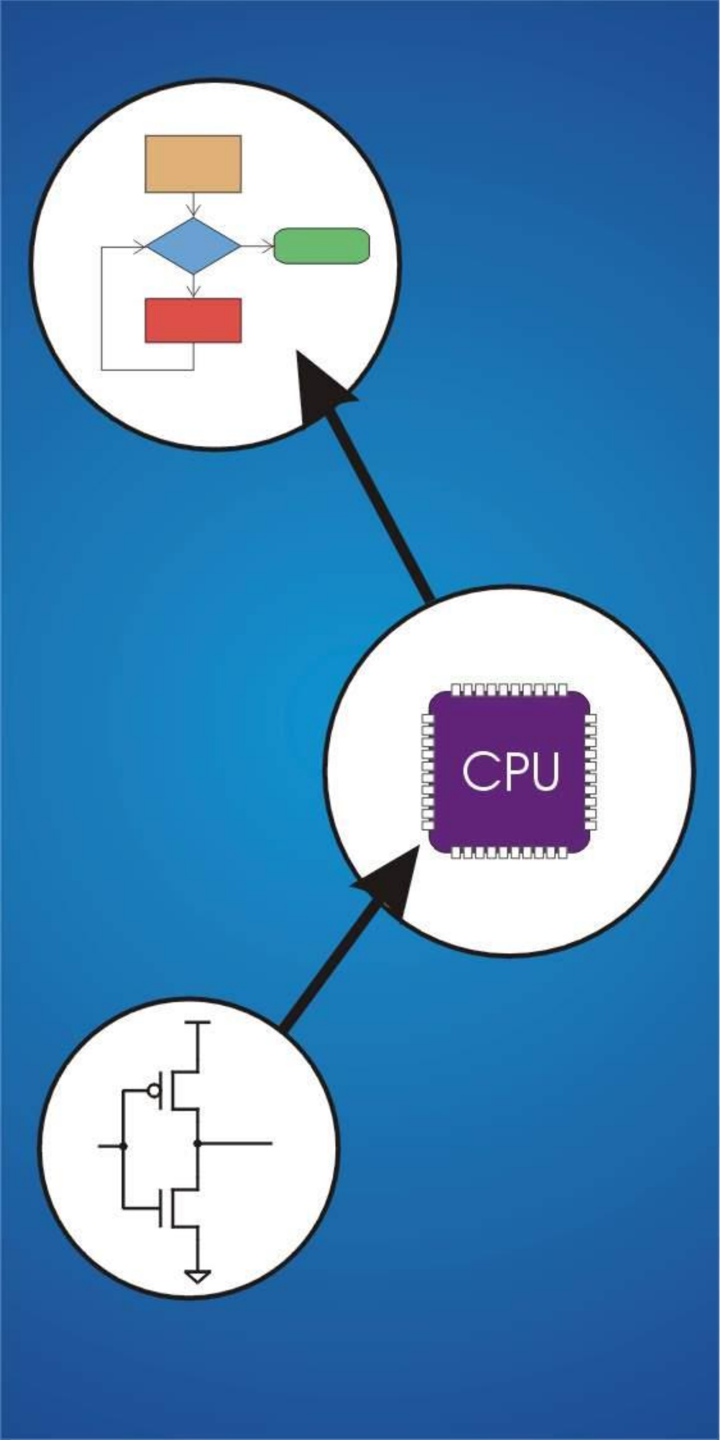


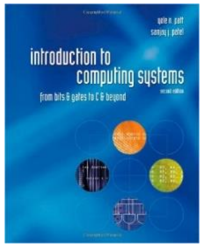
Midterm 2 Review

Chapters 4-16

LC-3

Topics

- Bit width
- Range of offsets
- Purpose of registers
- Basics of what the instructions do
- 2’s comp
- Basics of interrupts
- Stacks / stack protocol
- Hex to instruction
- Condition codes
- Instruction cycle
- Assembler directives

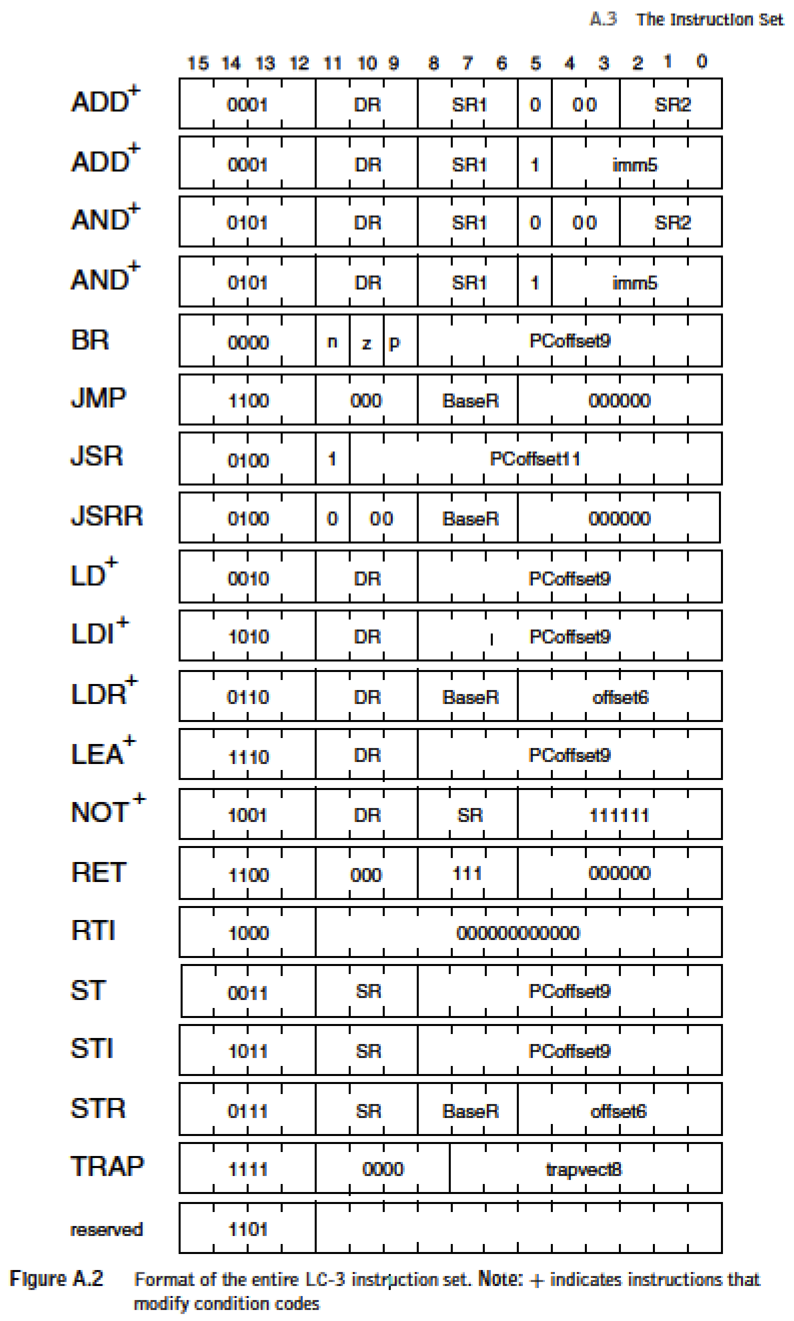

introduction to computing systems
from bits & gates to C & beyond

How many bits are in the IR?

A. 4
B. 3
C. 16
D. 2^{16}
E. None of the above

ISA

You will be allowed to use the one page instruction summary.



15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

ADD$^+$	0001	DR	SR1	0	00	SR2
ADD$^+$	0001	DR	SR1	1	imm5	
AND$^+$	0101	DR	SR1	0	00	SR2
AND$^+$	0101	DR	SR1	1	imm5	
BR	0000	n	z	p	PCoffset9	
JMP	1100	000	BaseR	000000		
JSR	0100	1	PCoffset11			
JSRR	0100	0	00	BaseR	000000	
LD$^+$	0010	DR	PCoffset9			
LDI$^+$	1010	DR	PCoffset9			
LDR$^+$	0110	DR	BaseR	offset6		
LEA$^+$	1110	DR	PCoffset9			
NOT$^+$	1001	DR	SR	111111		
RET	1100	000	111	000000		
RTI	1000	000000000000				
ST	0011	SR	PCoffset9			
STI	1011	SR	PCoffset9			
STR	0111	SR	BaseR	offset6		
TRAP	1111	0000	trapvect8			
reserved	1101					

Figure A.2 Format of the entire LC-3 instruction set. **Note:** + indicates instructions that modify condition codes

LC-3 Overview: Instruction Set

Opcodes

- **15 opcodes**
- *Operate* **instructions: ADD, AND, NOT**
- *Data movement* **instructions: LD, LDI, LDR, LEA, ST, STR, STI**
- *Control* **instructions: BR, JSR/JSRR, JMP, RTI, TRAP**
- **some opcodes set/clear** ***condition codes*****, based on result:**
  - **N = negative, Z = zero, P = positive (> 0)**

Data Types

- **16-bit 2’s complement integer**

Addressing Modes

- **How is the location of an operand specified?**
- **non-memory addresses:** ***immediate*****,** ***register***
- **memory addresses:** ***PC-relative*****,** ***indirect*****,** ***base+offset***

ADD/AND (Immediate)

this one means "immediate mode"

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD	0	0	0	1	Dst			Src1			1	Imm5				

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AND	0	1	0	1	Dst			Src1			1	Imm5				

Assembly Ex:
Add R3, R3, #1

Note: Immediate field is
***sign-extended*.**

Load and Store instructions

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LD	0	0	1	0	Dst			PCoffset9								

Example: LD R1, Label1
R1 is loaded from memory location labelled Label1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LDI	1	0	1	0	Dst			PCoffset9								

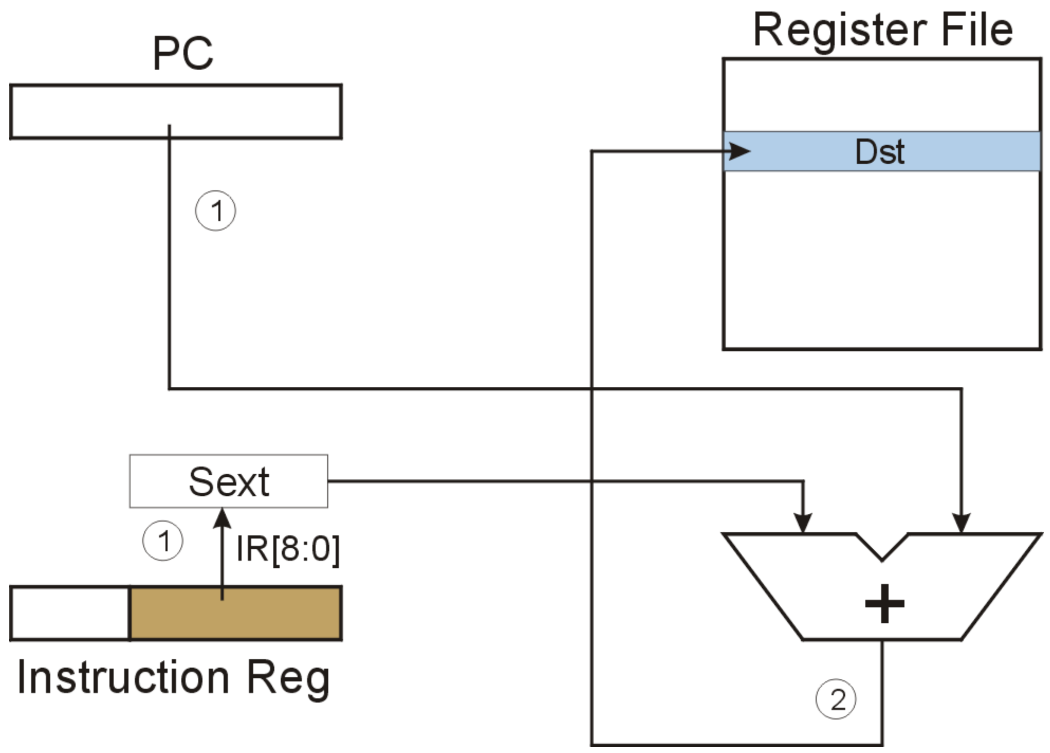
Example: LDI R1, Label1
R1 is loaded from address found at location Label1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LDR	0	1	1	0	Dst			Base			offset6					

Example: LDR R1, R4, #1
R1 is loaded from address pointed by R4 with offset 1.
Store instructions use the same addressing modes, except the register contents are written to a memory location.

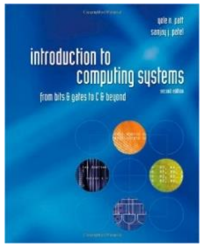
LEA (Immediate)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LEA	1	1	1	0	Dst			PCoffset9								



Assembly Ex:
LEA R1, Lab1

Used to initialize a pointer.



What instructions would achieve the same result as the LDI instruction below

```
      .ORIG x3000
MAIN  ADD R2, R2, #3
      LDI R2, FAR
FAR   .FILL xFFFC
```

A. LEA R2, FAR
   LD R2, R2, #0
B. LEA R2, FAR
   LDR R2, R2, #0
C. LD R2, FAR
   LDR R2, R2, #0
D. LD R2, MAIN
   LDR R2, R2, #2
E. C and D

Condition Codes

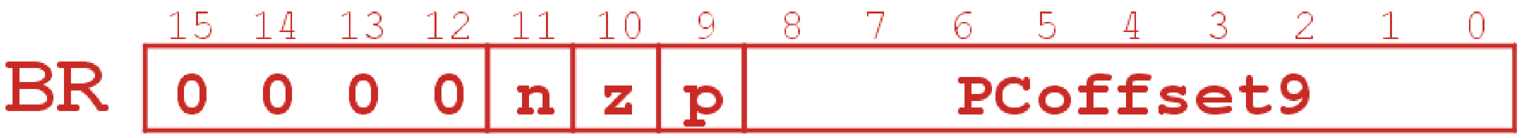
LC-3 has three condition code registers:

- **N -- negative**
- **Z -- zero**
- **P -- positive (greater than zero)**

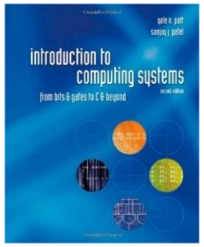
- **Set by any instruction that writes a value to a register (ADD, AND, NOT, LD, LDR, LDI, LEA)**

Exactly <u>one</u> will be set at all times

- **Based on the last instruction that altered a register**



Assembly Ex: BRz, Label



What Condition Code is set when the Branch instruction is reached

```
        .ORIG x3000
Main    LD R1,Twelve
        LEA R0, Twelve
        NOT R1,R1
        ADD R1,R1,1
        ADD R0,R0,R1
        BRnzp Main
Twelve  .FILL x000C
```

A. N
B. Z
C. P
D. Can’t be determined

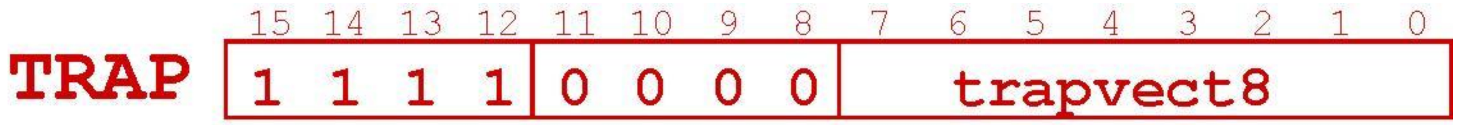
Assembler Directives

Pseudo-operations

- do not refer to operations executed by program
- used by assembler
- look like instruction, but “opcode” starts with dot

Opcode	*Operand*	*Meaning*
`.ORIG`	address	starting address of program
`.END`		end of program
`.BLKW`	n	allocate n words of storage
`.FILL`	n	allocate one word, initialize with value n
`.STRINGZ`	n-character string	allocate n+1 locations, initialize w/characters and null terminator

TRAP Instruction



Trap vector

- identifies which system call to invoke
- 8-bit index into table of service routine addresses
  - ➢ in LC-3, this table is stored in memory at 0x0000 – 0x00FF
  - ➢ 8-bit trap vector is zero-extended into 16-bit memory address

Where to go

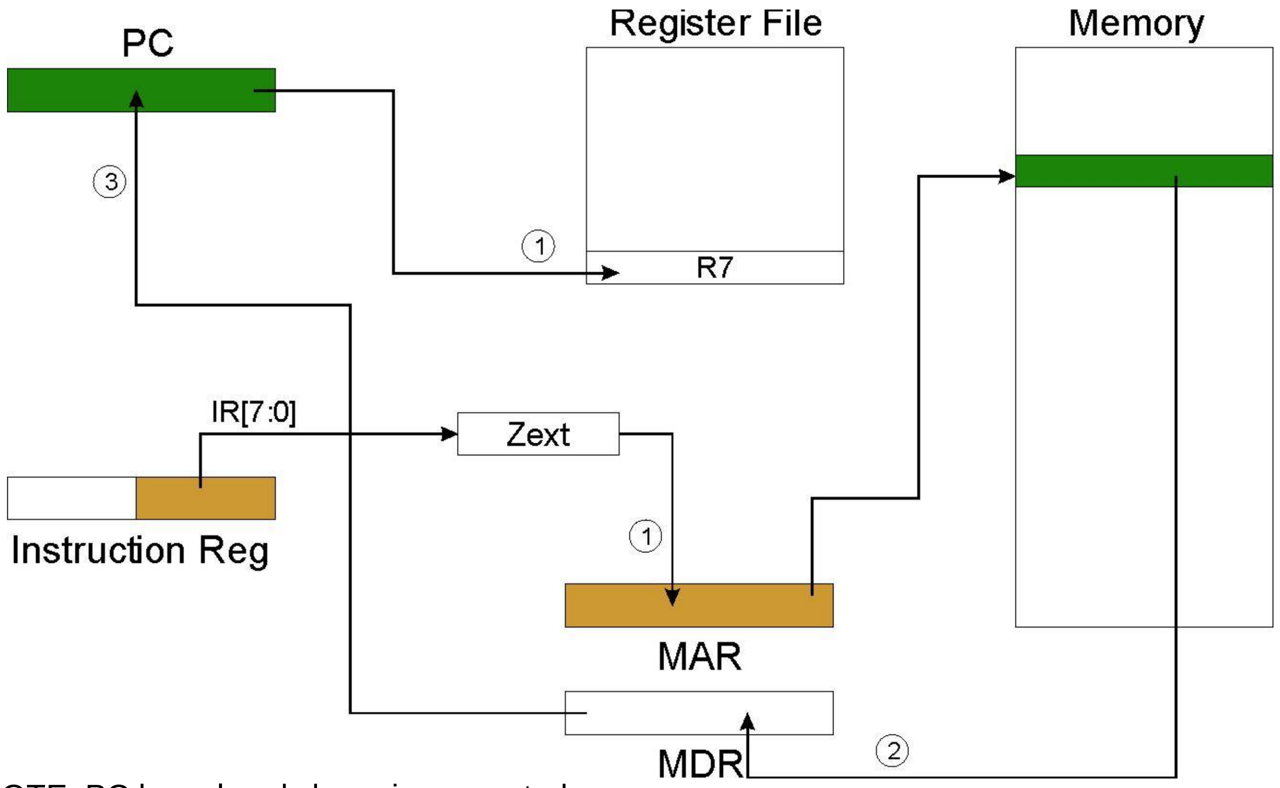
- lookup starting address from table; place in PC

How to get back

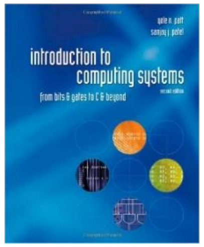
- save address of next instruction (current PC) in R7

TRAP

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TRAP	1	1	1	1	0	0	0	0	trapvect8							



NOTE: PC has already been incremented during instruction fetch stage.



Given the following segments of LC3 memory what will the PC be loaded with after this instruction has executed?
TRAP x21

location	data
x0020	x0420
x0021	x0231
x0022	x046C
x0023	x0326
x0024	x0324

location	data
x0323	xFADC
x0324	x32AC
x0325	xAE1F
x0326	x330F
x0327	x98A1

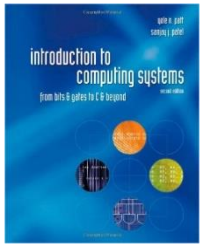
location	data
x0230	x2A43
x0231	x32AC
x0232	x5E1F
x0233	x8FB2
x0234	xE8A1

A. x0021
B. x0231
C. x32AC
D. xFADC
E. None of the above

Trap Codes

LC-3 assembler provides “pseudo-instructions” for each trap code, so you don’t have to remember them.

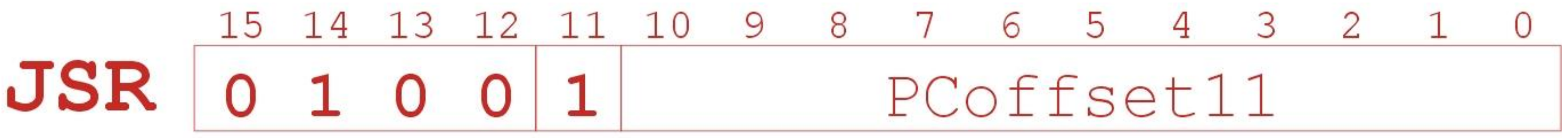
Code	*Equivalent*	*Description*
`HALT`	`TRAP x25`	Halt execution and print message to console.
`IN`	`TRAP x23`	Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].
`OUT`	`TRAP x21`	Write one character (in R0[7:0]) to console.
`GETC`	`TRAP x20`	Read one character from keyboard. Character stored in R0[7:0].
`PUTS`	`TRAP x22`	Write null-terminated string to console. Address of string is in R0.



What is the OUT Trap expecting when it is called

A. R5 to have the address of a character
B. R0 to have a characters ASCII value
C. R5 to have a characters ASCII value
D. R0 to have the address of a character
E. None of the above

JSR Instruction



Jumps to a location (like a branch but unconditional), and saves current PC (addr of next instruction) in R7.

- **saving the return address is called "linking"**
- **target address is PC-relative** (PC + Sext(IR[10:0]))
- **bit 11 specifies addressing mode**
  - **if =1, PC-relative: target address = PC + Sext(IR[10:0])**
  - **if =0, register: target address = contents of register IR[8:6]**

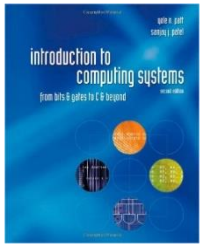
Example: Negate the value in R0

```
2sComp      NOT   R0, R0          ;  flip bits
            ADD   R0, R0, #1      ;  add one
            RET                   ;  return to caller
```

To call from a program (within 1024 instructions):

```
; need to compute R4 = R1 - R3
            ADD   R0, R3, #0      ;  copy R3 to R0
            JSR   2sComp          ;  negate
            ADD   R4, R1, R0      ;  add to R1
            ...
```

Note: Caller should save R0 if we'll need it later!



Why do we need the JSRR instruction

A. To save the return address in a specific register
B. To load the PC with a value greater than 256 locations away from the current PC
C. We don't it is the same as JMP R7
D. To return from an interrupt service routine
E. None of the above

RET (JMP R7)

How do we transfer control back to instruction following the TRAP or service/sub routine?

We saved old PC in R7.

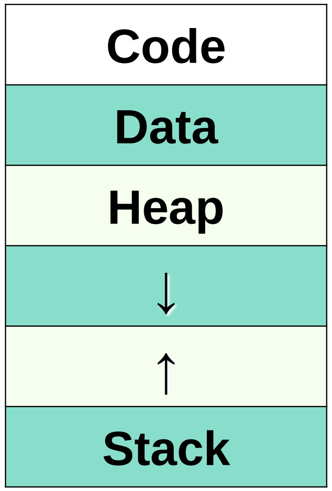
- **JMP R7 gets us back to the user program at the right spot.**
- **LC-3 assembly language lets us use RET (return) in place of “JMP R7”.**

Must make sure that service routine does not change R7, or we won’t know where to return.

Stack

Memory Usage

- Instructions are stored in code segment
- Global data is stored in data segment
- Local variables, including arrays, uses stack
- Dynamically allocated memory uses heap



- Code segment is write protected
- Initialized and uninitialized globals
- Stack size is usually limited
- Stack generally grows from higher to lower addresses.

Basic Push and Pop Code

For our implementation, stack grows downward (when item added, TOS moves closer to 0)

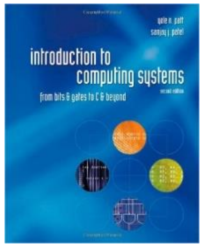
Push R0

```
        ADD   R6, R6, #-1 ; decrement stack ptr
        STR   R0, R6, #0  ; store data (R0)
```

Pop R0

```
        LDR   R0, R6, #0  ; load data from TOS
        ADD   R6, R6, #1  ; decrement stack ptr
```

- **Sometimes a Pop only adjusts the SP.**
- **Arguments pushed onto the stack last to first**

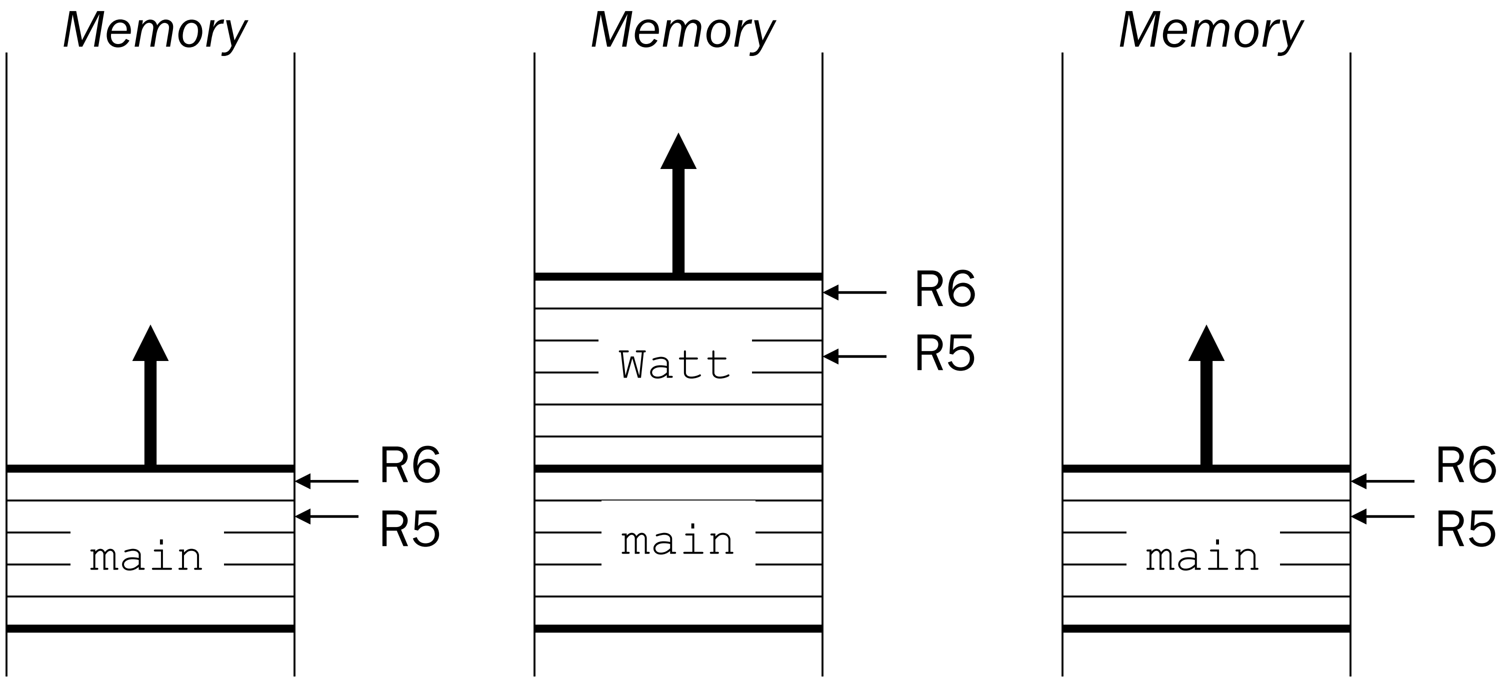


What location will the stack pointer point to after this code executes

```
        .ORIG x3000
Start   JSR Main
Main    LD R6 Stack
        LD R3, Start
        PUSH R1
        ADD R6, R6, #-2
        PUSH R3
        POP R6
Stack   .FILL x4800
```

A. X4801
B. X4000
C. x47FD
D. x47FF
E. A or D

Run-Time Stack



Before call

During call

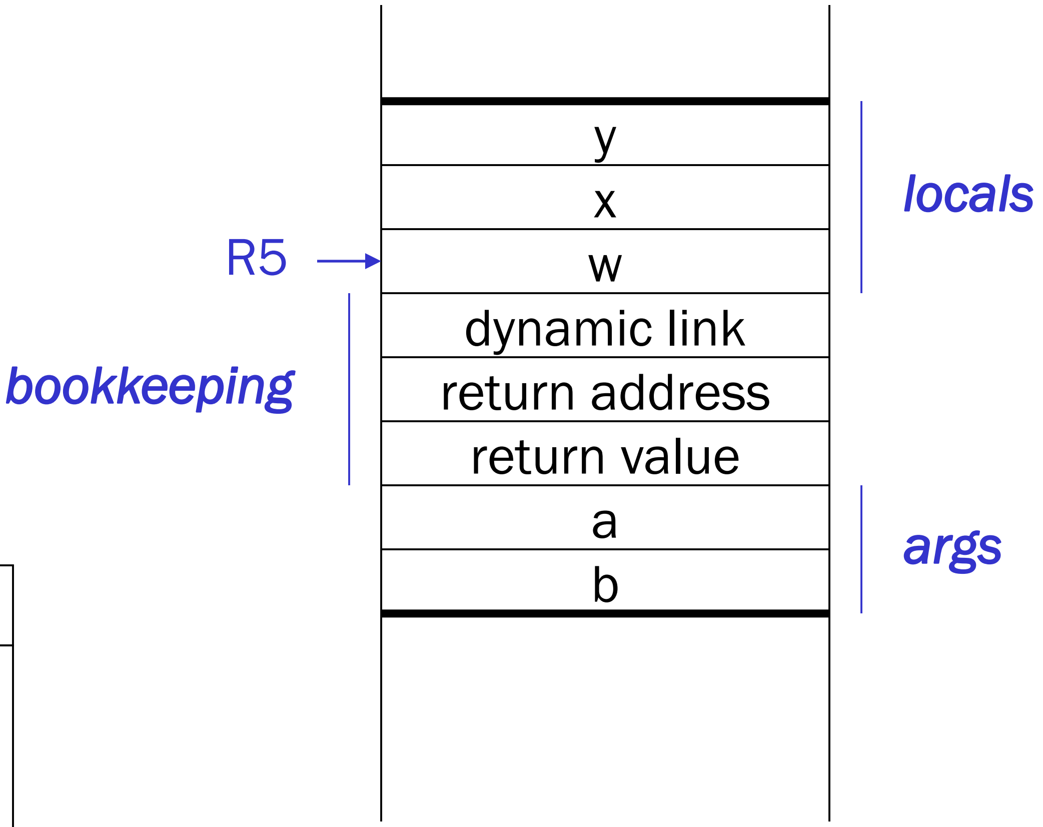
After call

Activation Record

```
int NoName(int a, int b)
{
   int w, x, y;
   .
   .
   .
   return y;
}
```

Name	Type	Offset	Scope
a	int	4	NoName
b	int	5	NoName
w	int	0	NoName
x	int	-1	NoName
y	int	-2	NoName

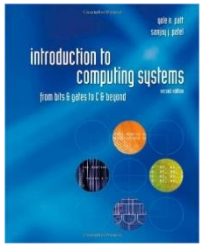
Lower addresses ⇧



Compiler generated Symbol table.
Offset relative to FP R5

Summary of LC-3 Function Call Implementation

1. Caller pushes arguments (last to first).
2. Caller invokes subroutine (JSR).
3. Callee allocates return value, pushes R7 and R5.
4. Callee allocates space for local variables.
5. Callee executes function code.
6. Callee stores result into return value slot.
7. Callee pops local vars, pops R5, pops R7.
8. Callee returns (JMP R7).
9. Caller loads return value and pops arguments.
10. Caller resumes computation…



What is not a benefit of using a stack for memory management

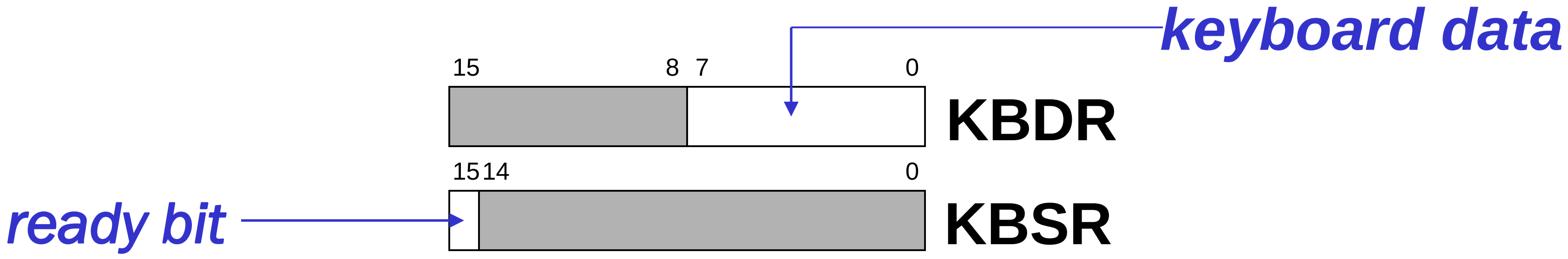
A. Functions only uses memory when they are active
B. Regions of memory can be marked read only
C. Data related to one function can be accessed by another function by altering the Frame pointer offset
D. Implementing of recursion is possible
E. None of the above

Input/Output

Input from Keyboard

When a character is typed:

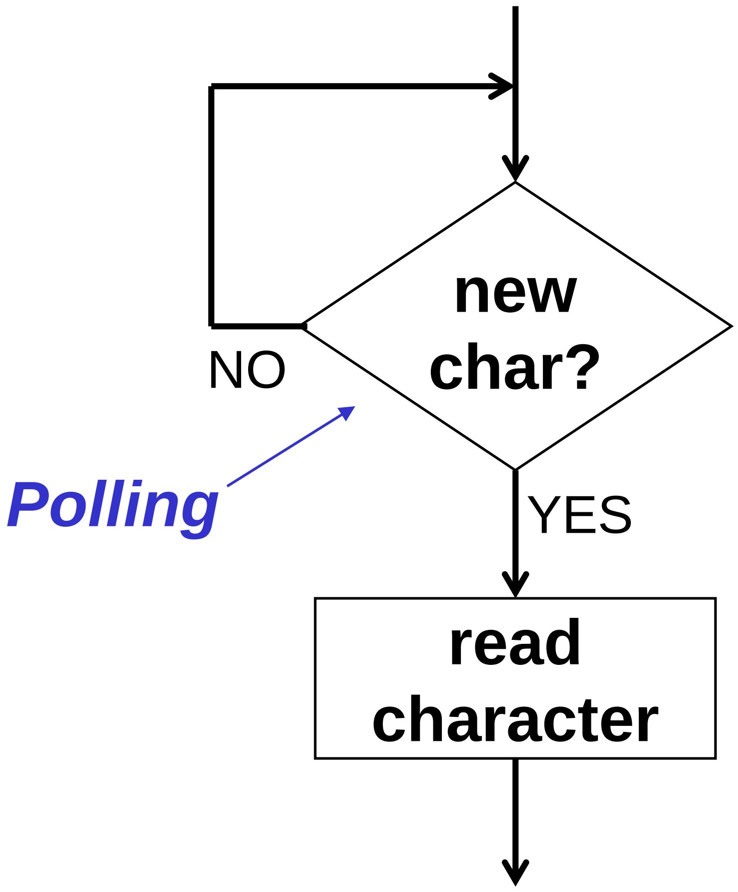
- its ASCII code is placed in bits [7:0] of KBDR (bits [15:8] are always zero)
- the "ready bit" (KBSR[15]) is set to one
- keyboard is disabled -- any typed characters will be ignored



When KBDR is read:

- KBSR[15] is set to zero
- keyboard is enabled

Basic Input Routine

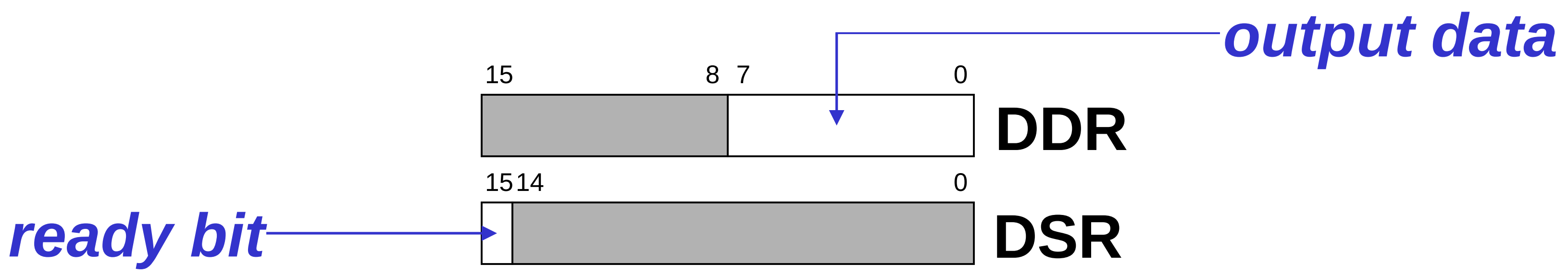


```
POLL     LDI   R0, KBSRPtr
         BRzp  POLL
         LDI   R0, KBDRPtr

         ...

KBSRPtr .FILL xFE00
KBDRPtr .FILL xFE02
```

Output to Monitor

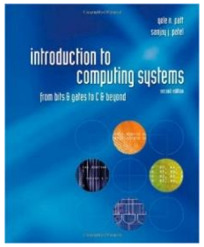
When Monitor is ready to display another character:

- **the "ready bit" (DSR[15]) is set to one**



When data is written to Display Data Register:

- **DSR[15] is set to zero**
- **character in DDR[7:0] is displayed**
- **any other character data written to DDR is ignored (while DSR[15] is zero)**


yale n. patt
sanjay j. patel
introduction to computing systems
from bits & gates to C & beyond
second edition

To implement Memory Mapped IO extra hardware will be needed in the

A. Processor
B. Memory Controller
C. Register File
D. B and C
E. None of the above

Interrupt-Driven I/O

External device can:

(1) Force currently executing program to stop;

(2) Have the processor satisfy the device's needs; and

(3) Resume the stopped program as if nothing happened.

Interrupt is an unscripted subroutine call, triggered by an external event.

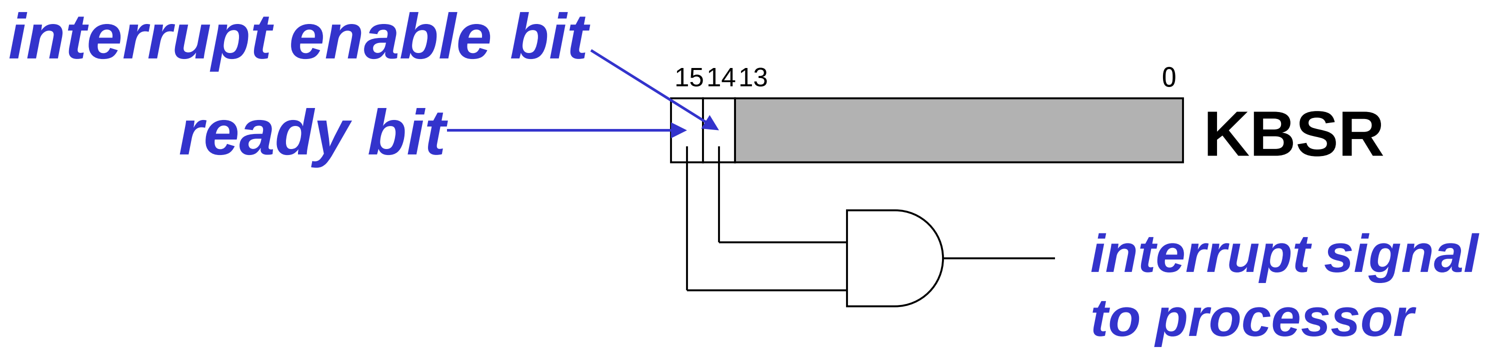
Interrupt-Driven I/O

To implement an interrupt mechanism, we need:

- **A way for the I/O device to signal the CPU that an interesting event has occurred.**
- **A way for the CPU to test whether the interrupt signal is set and whether its priority is higher than the current program.**

Generating Signal

- **Software sets "interrupt enable" bit in device register.**
- **When ready bit is set and IE bit is set, interrupt is signaled.**

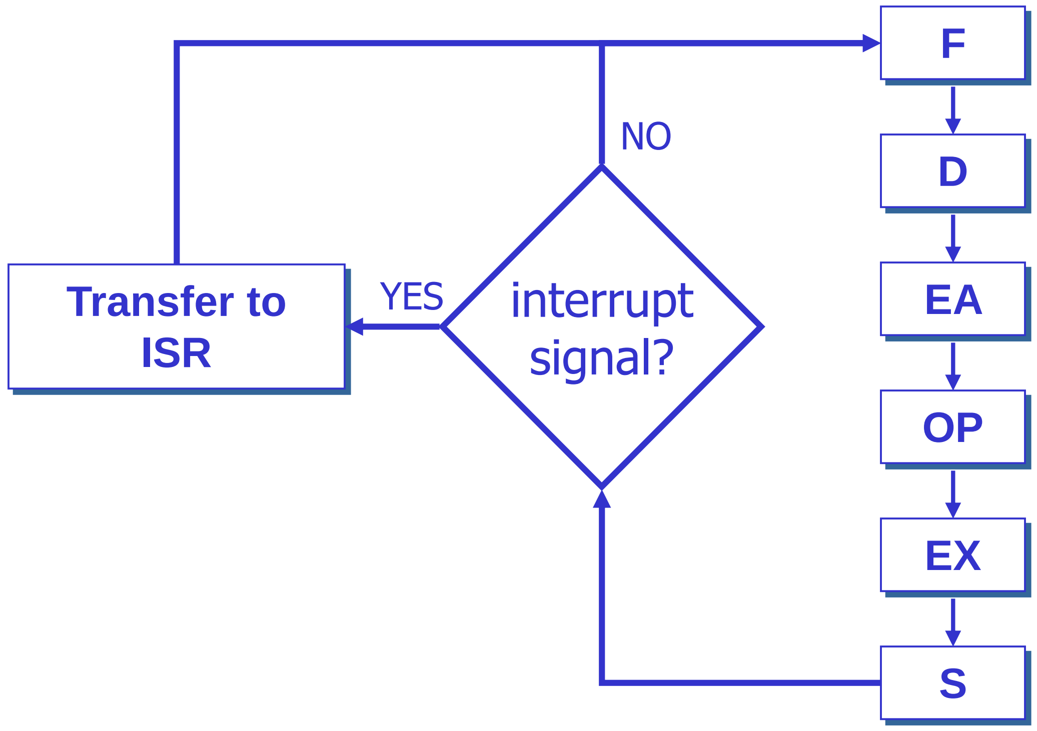


Testing for Interrupt Signal

CPU looks at signal between STORE and FETCH phases.

If not set, continues with next instruction.

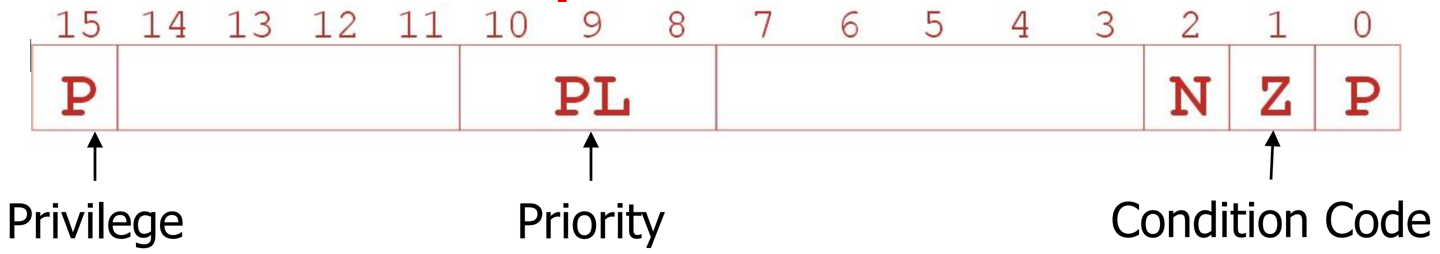
If set, transfers control to interrupt service routine.



Processor State

- **Must be saved before servicing an interrupt.**
- **What state is needed to completely capture the state of a running process?**

Processor Status Register

- 

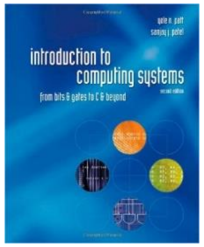
LC-3: 8 priority levels (PL0-PL7)

Program Counter

- **Pointer to next instruction to be executed.**

Registers

- **All temporary state of the process that’s not stored in memory.**



How does the RTI instruction know if it needs to switch from the supervisor stack to the user stack?

A. It checks if the priority level of the PSR it pops off the supervisor stack is higher than the current priority level
B. The RTI instruction does not need to switch between stacks this is the RET instructions responsibility
C. It checks if the privilege level of the PSR it pops off the supervisor stack is 1
D. None of the above