Memory and Data Structures

Arrays, Stacks, Queues

(Ch 10 & 16)
MSI Review Session:

Sunday 5:00 to 6:30 pm
ARC 202
Memory

• This is the “RAM” in a system
• We have seen labels and addresses point to pieces of memory storing:
  • words
  • bytes
  • strings
  • numbers
• Memory is just a collection of bits
• We could use it to represent integers
• Or as an arbitrary set of bits
Treat memory as a giant array

- Compiler or programmer decides what use to make of it.
- The element numbering starts at 0.
- The element number is an address.
- In “C” to allocate some memory:

```
char m[size_of_array];
```
Storage of Data

- LC-3 architecture is “word addressable” meaning that all addresses are “word” addresses.
- This means the smallest unit of memory we can allocate is 16-bits, a word.
- Use ‘LD’ (load) and ‘ST’ (store) to access this unit (or LDR & STR).
Example

mychar .BLKW 1
newline .FILL xA

LD R1, newline
GETC
ST R0, mychar
JSR Sub ; R2=R1-R0
BRz found_newline

found_newline ...
The data is placed in memory like this at start up (assuming data section starts at address 1). The “mychar” variable will change to the value of the character entered by the user once stored.
Pointers and Arrays

We've seen examples of both of these in our LC-3 programs, let's see how these work in “C”

**Pointer**
- Address of a variable in memory
- Allows us to **indirectly** access variables
  - in other words, we can talk about its **address** rather than its **value**

**Array**
- A list of values arranged sequentially in memory
- Example: a list of telephone numbers
- Expression `a[4]` refers to the 5th element of the array `a`
LEAN, FOO
Arrays

Array implementation is very important

- Most assembly languages have only basic concept of arrays (BLKW)
- From an array, any other data structure we might want can be built
Properties of arrays:

- Each element is the same size
- Elements are stored contiguously
- First element at the smallest memory address

In assembly language we must

- Allocate correct amount of space for an array
- Map array addresses to memory addresses
LC-3 declarations of arrays within memory

To allocate a portion of memory (more than a single variable’s worth)

```
variablename .BLKW numelements
```

numelements is just that, numbering starts at 0 (as in C)
Array of Integers

Calculating the address of an array element

```c
int myarray[7] /* C */
.BLKW 7 ; LC-3
```

- If base address of “myarray” is 25

- Which is base address + distance from the first element
How do you get the address of `myarray`?

- Use the “load effective address” instruction, “LEA”
- Keep clear the difference between an address and the contents of an address.
To get address of **myarray[4]** in LC-3, write the code...

```
LEA   R0, myarray
ADD   R1, R0, 4
```

Now, if we wanted to increment element number 5 by 1...

```
LDR   R4, R1, 0
ADD   R4, R4, 1
STR   R4, R1, 0
```
Address vs. Value

Sometimes we want to deal with the *address* of a memory location, rather than the *value* it contains.

Recall example from Chapter 6: adding a column of numbers.
- R2 contains address of first location.
- Read value, add to sum, and increment R2 until all numbers have been processed.

R2 is a pointer -- it contains the address of data we’re interested in.
2-Dimensional arrays are more complicated in assembly

- Memory is a 1-D array
- Must map 2-D array to 1-D array
- Arrays have rows and columns
  - $r \times c$ array
  - $r =$ rows
  - $c =$ columns
Two sensible ways to map 2-D to 1-D

Row major form: (rows are all together)

Column major form: (columns are all together)

4x2 example
How do you calculate addresses in a 2-D array?

- **Row Major:**

\[
\text{Address } (r_i, c_i) = \text{Base Address} + \left( ((r_i \times \text{Number of Cols}) + c_i) \times \text{Element size} \right)
\]

- **Column Major:**

\[
\text{Address } (r_i, c_i) = \text{Base Address} + \left( ((c_i \times \text{Number of Rows}) + r_i) \times \text{Element size} \right)
\]
Summary of 2D arrays

- Row/Column major (storage order)
- Base address
- Size of elements
- Dimensions of the array

How about 3-D arrays?
Bounds Checking

- Many HLL’s have bounds checking (not C!!!)
- Assembly languages have no implied bounds checking
- Your program is in total control of memory
- With a 5 x 3 array, what does the following address?

```
array .BLKW 15
LEA R1, array
ADD R1, R1, 15
LDR R0, R1, 0
```

- Bounds checking is often a good idea!!
- Most C development environments include optional bounds checking.
HeartBleed

Heartbeat

bob, 3 →

bob ←

bob, 65k

bob, ...

"<" Line

Line OpenSSL
Stacks

A LIFO (last-in first-out) storage structure.

- The first thing you put in is the last thing you take out.
- The last thing you put in is the first thing you take out.

This means of access is what defines a stack, not the specific implementation.

Two main operations:

- **PUSH**: add an item to the stack
- **POP**: remove an item from the stack
A Physical Stack

Coin rest in the arm of an automobile

Initial State  
After One Push  
After Three More Pushes  
After One Pop

First quarter out is the last quarter in.
A Hardware Implementation

Data items move between registers

Initial State

Empty: Yes

After One Push

Empty: No

After Three More Pushes

Empty: No

After Two Pops

TOP

#18

#12

#31

#18

TOP

TOP

TOP

#5

#31

#18

#31

#18

#12
A Software Implementation

Data items don't move in memory, just our idea about where the TOP of the stack is.

Initial State

After One Push

After Three More Pushes

After Two Pops

By convention, R6 holds the Top of Stack (TOS) pointer.
Stack overflow

Basic Push and Pop Code

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

Push

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

Pop

LDR R0, R6, #0 ; load data from TOS
ADD R6, R6, #1 ; increment stack ptr
Pop with Underflow Detection

If we try to pop too many items off the stack, an underflow condition occurs.

- Check for underflow by checking TOS before removing data.
- Return status code in R5 (0 for success, 1 for underflow)

```
POP  LD R1, EMPTY ; EMPTY = -x4000
ADD R2, R6, R1 ; Compare stack pointer
BRz FAIL ; to x4000 to see if empty

LDR R0, R6, #0
ADD R6, R6, #1
AND R5, R5, #0 ; SUCCESS: R5 = 0
RET
FAIL AND R5, R5, #0 ; FAIL: R5 = 1
ADD R5, R5, #1
RET
EMPTY .FILL xC000 ; 2SC rep of -x4000
```

x3389 - x4000

number

\[-\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]
Push with Overflow Detection

If we try to push too many items onto the stack, an overflow condition occurs. This example assumes stack has room for 5 items.

- Check for overflow by checking TOS before adding data.
- Return status code in R5 (0 for success, 1 for overflow)

```
PUSH   LD R1, MAX   ; MAX = -x3FFB
     ADD  R2, R6, R1  ; Compare stack pointer
     BRz  FAIL       ; top address to see if full
     ADD  R6, R6, #1
     STR   R0, R6, #0
     AND  R5, R5, #0  ; SUCCESS: R5 = 0
     RET
     FAIL AND R5, R5, #0  ; FAIL: R5 = 1
     ADD  R5, R5, #1
     RET
     MAX .FILL xC005  ; 2SC of -x3FFB
```
Stack Example

- Printing out a positive integer, character by character
- Push LSB to MSB
- Pop MSB to LSB (LIFO)

\[
\text{integer} = 1024
\]

if integer == 0 then
    push '0'
else
    while integer != 0
        digit \leftarrow \text{integer mod base}
        char \leftarrow \text{digit + 48}
        push char onto stack
        integer \leftarrow \text{integer div base}
    \]

while stack is not empty
    pop char
    print char
Arithmetic Using a Stack

Instead of registers, some ISA's use a stack for source and destination operations: a zero-address machine.

- Example:
  ADD instruction pops two numbers from the stack, adds them, and pushes the result to the stack.

Evaluating \((A+B) \cdot (C+D)\) using a stack:

1. push A
2. push B
3. ADD
4. push C
5. push D
6. ADD
7. MULTIPLY
8. pop result

Why use a stack?

- Limited registers.
- Convenient calling convention for subroutines.
- Algorithm naturally expressed using LIFO data structure.
Example: OpAdd

POP two values, ADD, then PUSH result.

START

POP

OK?

Yes

No

Put back first

ADD

Range OK?

Yes

No

Put back both

PUSH

RETURN
Example: OpAdd

OpAdd

JSR POP ; Get first operand.
ADD R5,R5,#0 ; Check for POP success.
BRp Exit ; If error, bail.
ADD R1,R0,#0 ; Make room for second.
JSR POP ; Get second operand.
ADD R5,R5,#0 ; Check for POP success.
BRp Restore1 ; If err, restore & bail.
ADD R0,R0,R1 ; Compute sum.
JSR RangeCheck ; Check size.
BRp Restore2 ; If err, restore & bail.
JSR PUSH ; Push sum onto stack.
RET

Restore2
ADD R6,R6,#-1 ; Decr stack ptr (undo POP)
RET

Restore1
ADD R6,R6,#-1 ; Decr stack ptr
RET
Queues

A **queue** is a **FIFO** (First In, First Out).

- The classic analogy of a queue is a line.
  - Person gets on the end of the line (the **Tail**),
  - Waits,
  - Gets off at the front of the line (the **Head**).
- Getting into the queue is an operation called **enqueue**
- Taking something off the queue is an operation called **dequeue**.
- It takes 2 pointers to keep track of the data structure,
  - Head (let’s use R5)
  - Tail always points to empty element (R6)
Initial state:

After 1 enqueue operation:

After another enqueue operation:
After a dequeue operation:

| X | Y |     |

Head (R5)  

Tail (R6)

Like stacks, when an item is removed from the data structure, it is physically still present, but correct use of the structure cannot access it.
Implementation of a queue

Storage:

```assembly
.queue .BLKW infinity ; assume infinite for now
LEA R5, queue ; head
LEA R6, queue ; tail
```

Enqueue (item):

```assembly
STR R0, R6, #0 ; R0 has data to store
ADD R6, R6, #1
```

Dequeue (item):

```assembly
JSR SUB ; R0 = R5-R6
BRz queue_empty ; put data in R1
LDR R1, R5, #0
ADD R5, R5, #1
```
Circular Queues

- To avoid infinite array, wrap around from end to beginning.
- Head == Tail means empty
- Head points to first item (for next dequeue)
- Tail point to empty location (for next enqueue)

Example of an 8 element circular queue
After “enqueue’ing” one element

After “enqueue’ing” another element
After “dequeuing” an element
5086 bytes

22
21
Storage and initialization:

queue .BLKW queue_size
queue_end .BLKW 1
LEA R5, queue ; head
LEA R6, queue ; tail

Enqueue (item)

STR R0, R6, #0 ; data to enqueue is in R0
ADD R6, R6, #1
LEA R1, queue_end
JSR SUB ; R1 = R1 - R6
BRp continue1
LEA R6, queue ; wrap around
continue1

modulo
Dequeue (item):

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSR</td>
<td>SUB</td>
</tr>
<tr>
<td>BRz</td>
<td>queue_empty</td>
</tr>
<tr>
<td>LDR</td>
<td>R0, R5, #0</td>
</tr>
<tr>
<td>ADD</td>
<td>R5, R5, #1</td>
</tr>
<tr>
<td>LEA</td>
<td>R1, queue_end</td>
</tr>
<tr>
<td>JSR</td>
<td>SUB</td>
</tr>
<tr>
<td>BRn</td>
<td>continue2</td>
</tr>
<tr>
<td>LEA</td>
<td>R5, queue</td>
</tr>
<tr>
<td></td>
<td>continue2</td>
</tr>
<tr>
<td></td>
<td>; R1 = R5 - R6</td>
</tr>
<tr>
<td></td>
<td>; R1 = R5 - R1</td>
</tr>
<tr>
<td></td>
<td>; wrap around</td>
</tr>
</tbody>
</table>
Summary of data structures

- All data structures are based on the simple array.
- 2D Arrays, Stacks, Queues.
- It is all about the implementation.
- Bounds checking is important.
- If not documented can become confusing.
Microcontrollers and Embedded Systems

Taking over the world
What is a microcontroller?

- A microprocessor
- Usually not cutting edge
- Dependable
  - All major bugs well known
- Predictable
  - Critical for real-time processing
- On-chip peripherals and memory
- Parallel and serial digital I/O
- Analog I/O
- Counters and timers
- Internal ROM and/or EPROM

Referred to as a “System On a Chip” (SoC)
What are microcontrollers used in?

- EVERYTHING

Some products that you might know:

- NASA’s Sojourner Rover – 8-bit Intel 80C85
- Palm Vx handheld – 32-bit Motorola Dragonball EZ
- Sonicare toothbrush – 8-bit Zilog Z8
- The Vendo V-MAX 720 Soda Machine – Motorola HC11
- Miele dishwasher – 8-bit Motorola 68HC05
- Hunter 44550 Programmable Thermostat – (4-bit cpu)
What word size are they? 32-bit is starting to dominate.
Microcontroller unit sales are much higher than microprocessors.

... and are MUCH, MUCH cheaper.
Dynamics of TOTAL CPU unit sales is changing
So what languages are they being programmed in?

<table>
<thead>
<tr>
<th>Language</th>
<th>'98-'99</th>
<th>'99-'00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly</td>
<td>~ 21%</td>
<td>~ 10%</td>
</tr>
<tr>
<td>C</td>
<td>~ 69%</td>
<td>~ 80%</td>
</tr>
<tr>
<td>C++</td>
<td>~ 5%</td>
<td>~ 6%</td>
</tr>
<tr>
<td>Java</td>
<td>~ 1%</td>
<td>~ 2%</td>
</tr>
<tr>
<td>Other</td>
<td>~ 3%</td>
<td>~ 2%</td>
</tr>
</tbody>
</table>

Global programming language usage

Most Popular Coding Languages of 2014

- Python: 30.3%
- Ruby: 10.6%
- Java: 22.2%
- C++: 13%
- Javascript: 5.2%
- C#: 5%
- C: 4.1%
- PHP: 3.3%
- Perl: 1.6%
- Go: 1.5%
- Bash: 0.1%
- TCL: 0.03%
- Lua: 0.04%
- Clojure: 0.2%
- Objective C: 0.4%
- Scala: 1%
Summary

- Microcontrollers/SoC are now the most common devices programmed
- The consist of a CPU plus a lot of other stuff
  - Built in IO
  - Built in audio/video
  - Built in sensors
  - etc
Instructions:

- This practice exam is based off of last quarter's midterm. Due to the rearrangement of class certain sections were removed. This means the length is not necessarily representative of the given midterm.

- This exam is closed book and closed notes. You may NOT use a calculator.

- Do not remove the staple.

- The last page is the LC3 instruction and ASCII reference. You may tear it off, but leave the staple.

- Always show your work in the space provided. If you do not show your work, you will not be given full credit for that problem.

- Do not use extra paper.
\(-1 + -2\)
1) [pts] Boolean Logic:

a) (pts) Create a logic circuit (gates) for the following truth table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

b) (pts) Write the truth table for the following boolean circuit:

Assume level of difficulty from the homework.
2) [pts] Binary Conversion

a) (pts) Fill in the following table by converting the given number to the other bases. Assume that each number is 8 bits. If number is un-representable by given representation indicate this. **Show your work!**

<table>
<thead>
<tr>
<th>Decimal</th>
<th>2's complement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10110110</td>
</tr>
<tr>
<td></td>
<td>11101010</td>
</tr>
<tr>
<td></td>
<td>11111011</td>
</tr>
<tr>
<td>37</td>
<td>10100111</td>
</tr>
<tr>
<td>93</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

b) (pts) convert 431₅ to base 3.
c) (pts) convert 431₆ to base 4.
d) (pts) convert 321₄ to base 2.
3) [pts] Binary Arithmetic

a) (pts) Perform the following arithmetic operations on the unsigned integers. Do not convert them to decimal first and show your “carries” between digits. Assume variable size is same as digits given. Indicate whether there is overflow or no overflow.

\[
\begin{array}{ll}
\text{Ob} & 1 0 0 1 1 0 1 1 \\
\text{+ Ob} & 0 0 1 1 0 1 1 0 \\
\text{Ob} & \\
\text{Ob} & 1 0 1 D 2 \\
\text{+ Ob} & 2 8 0 0 \\
\text{Ob} & 0 x
\end{array}
\]

b) (pts) Perform the following arithmetic operations on the 2's complement integers. Do not convert them to decimal first and show your “carries” between digits. Indicate whether there is overflow or no overflow.

\[
\begin{array}{ll}
\text{Ob} & 1 0 1 0 0 1 0 0 \\
\text{+ Ob} & 1 1 0 1 1 0 0 1 \\
\text{Ob} & \\
\text{Ob} & 0 1 0 1 0 1 1 0 \\
\text{+ Ob} & 1 0 0 0 1 1 1 1 \\
\text{Ob} & 0 x
\end{array}
\]
\[
\frac{1}{AB^3} + 1 < q
\]
\[< 7 <
\]
4) [pts] LC-3 Architecture

a) (pts) How many memory locations can the LC-3 address?

b) (pts) What is the word size of the LC-3?

c) (pts) How many general purpose registers does the LC-3 have and what are they named?

d) (pts) What two things happen during the FETCH phase of the instruction cycle?

e) (4 pts) List what the PC, IR, MDR, and MAR stand for in the LC-3 architecture and briefly what they do.
5) \([pts]\) Binary Multiplication

Do not convert the numbers to unsigned form and flip the sign back after multiplication.

a) \([pts]\) perform \(-3\times7\) in 4 bit 2's complement

a) \([pts]\) perform \(2\times4\) in 4 bit 2's complement

a) \([pts]\) perform \(-1\times6\) in 4 bit 2's complement

a) \([pts]\) perform \(3\times5\) in 4 bit 2's complement
6) [pts] Digital Logic

a) (pts) Draw the gate level diagram for a 4 decoder, be sure to label your circuit.

\[ \overline{A + B} \]

\[ \overline{A \cdot \overline{B}} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]

\[ \overline{A + B} \]

\[ \overline{A \cdot B} \]

\[ \overline{B + \overline{C}} \]
7) [ pts ] LC-3 Datapath

Indicate which data lines are being used and indicate the values on those lines for the instruction
ADD R0, R1, R2
Assume R1 contains the value x1234 and R2 contains the value x4
8) [pts] LC-3 Coding

a) Write LC-3 assembly code that will NAND the values in R1 with R3 and store the result in R0.
b) Write a subroutine for subtraction. Be sure to use safe register calling conventions. Use R0 for the output and R1,R2 for the input.
8) [pts] LC-3 Code Running

a) After the following LC-3 code executes, what are the ending contents of the registers and memory? Assume some registers/memories have starting values as indicated. If blank, the content is unknown. Remember that both registers and memory locations are 16-bits wide. The memory portion starts at address 0x3200.

```
LEA R1, label10
LDR R2, R1, #0
STR R0, R1, #4
LEA R0, label12
ADD R5, R0, R1
LEA R0, label11
AND R7, R2, R5
MUL R3, R0
STR R7, R6, #2
STR R2, R1, #1
```

---

0 x DEAD

1 x 1234

2 x 3200

3 1234

4 x 3200

5 x DEAD

6
8) [pts] LC-3 Instructions
Decode each instruction and describe the operation it is performing.

a) (pts) 0001 0010 1111 1100

b) (pts) 0111 0000 1001 1111

c) (pts) 0000 1011 1111 1001