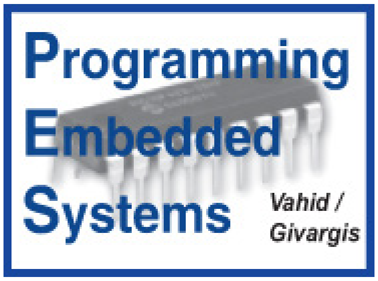


Programming Embedded Systems

An Introduction to Time-Oriented Programming

Version 4.0

Frank Vahid
University of California, Riverside

Tony Givargis
University of California, Irvine

Bailey Miller
University of California, Riverside

UniWorld Publishing, Lake Forest, California

Version 4.0, August, 2012.

Formatted for electronic publication.

ISBN 978-0-9829626-5-7 (print version of e-book)

UniWorld Publishing

www.programmingembeddedsystems.com
info@programmingembeddedsystems.com

Contents

Chapter 1: Introduction

The first computers of the 1940s and 1950s occupied entire rooms. The 1960s and 1970s saw computers shrink to the size of bookcases. Continued shrinking in the 1980s brought about the era of personal computers. Around that time, computers also became small enough to be put into other electrical devices, such as into clothes washing machines, microwave ovens, and cash registers. In the 1990s, those computers became known as embedded systems.

What is an embedded system?

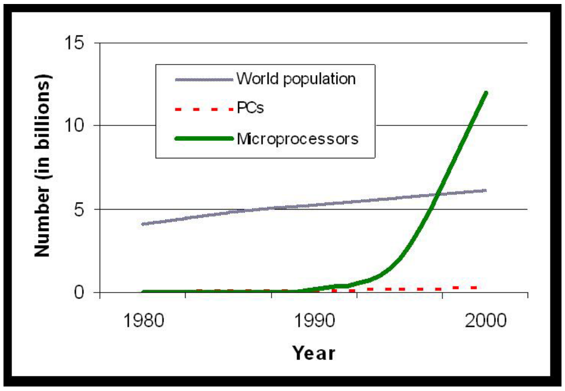
An **embedded system** is a computer embedded within another device. The embedded computer is composed of hardware and software sub-systems designed to perform one or a few dedicated functions. Embedded systems are often designed under stringent power, performance, size, and time constraints. They typically must react quickly to changing inputs and generate new outputs in response. Aside from PCs, laptops, and servers, most systems that operate on electricity and do something intelligent have embedded systems. Simple embedded system examples include the computer in a clothes washing machine, a motion-sensing lamp, or a microwave oven. More complex examples include the computer in an automobile cruise control or navigation system, a mobile phone, a cardiac pacemaker, or a factory robot. (Wikipedia: Embedded_Systems)

Examples of embedded systems include computers in simple systems like blinking tennis shoes or coffee makers, to more complex systems like mobile phones or automated teller machines.

Try: List three embedded systems that you interact with regularly.

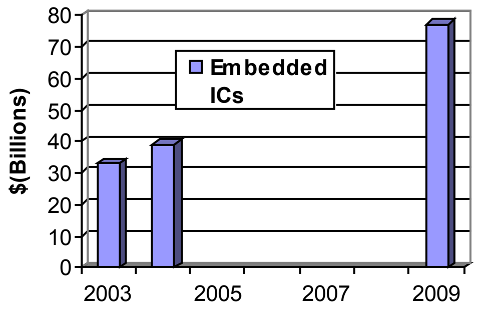
Wikipedia:iPhone	Wikipedia:IP_Phone	Wikipedia:HVAC_Control_System
Wikipedia:SetTopBox	Wikipedia:PlayStation	Wikipedia:Flight_Control_System
Wikipedia:Amazon_Kindle		Wikipedia:Engine_Control_Unit

Each year over 10 billion microprocessors are manufactured. Of these, about 98% end up as part of an embedded system.



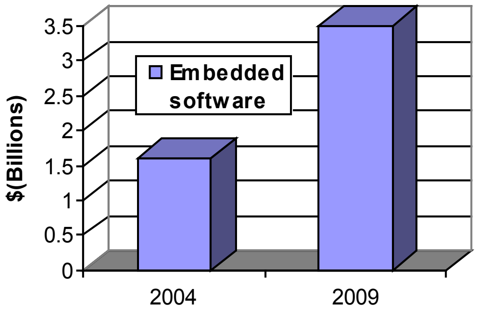
The microprocessors produced per year is growing exponentially, mostly destined for embedded systems.

Source: Study of Worldwide trends and R&D programmes in Embedded Systems by FAST GmbH. ftp://ftp.cordis.europa.eu/pub/ist/docs/embedded/final-study-181105_en.pdf



Gross sales of ICs destined for embedded systems is growing each year.

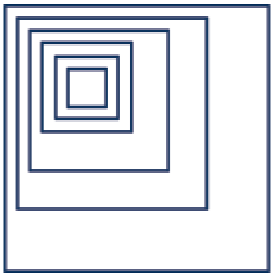
Source: BCC, Inc.



Gross sales of embedded software is also growing rapidly.

Source: BCC, Inc.

Integrated circuits (a.k.a. ICs or chips), on which microprocessors are implemented, have been doubling in transistor capacity roughly every 18 months, a trend known as **Moore's Law** (Wikipedia: Moore's Law). Such doubling means: (1) that a same-size system (e.g., a cell phone) gets more capable, and (2) that a same-capability system can be made smaller (halved every 18 months) thus enabling new inventions (e.g., computerized pills that can be ingested) (Wikipedia: Motes).



IC size shrinking in half every 18 months; note the reduction after just 5 * 18 months = 90 months or 7.5 years.

Try: Fold a sheet of paper in half as many times as you can. Each fold corresponds to IC size shrinking in 18 months. Notice how size shrinks dramatically after just a few folds.

Try: Think of a new invention that would be enabled by a microprocessor that is the size of a speck of dust, self-powers for years, has ample memory, and can communicate wirelessly.

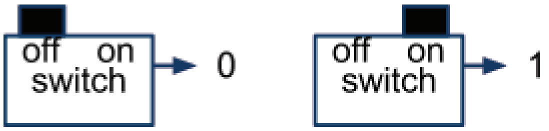
Basic components

A system with electrical components uses wires with continuous voltage signals. A useful abstraction is to consider only two voltage ranges, a “low” range (such as 0 Volts to 0.3 Volts) that is abstracted to 0, and a “high range” (such as 0.7 Volts to 1.2 Volts) that is abstracted to 1. A **bit** (short for “binary digit”) is one digit of such a two-valued item. A bit that can change over time is called a digital **signal**. “Digital” refers to the signal having discrete rather than continuous possible values. (Wikipedia: Digital signal).

Switch and push button

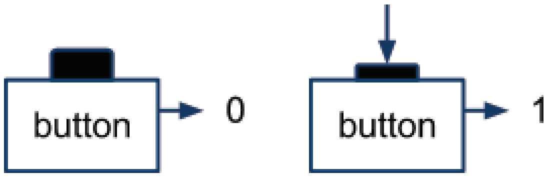
A switch is an electromechanical component with a pair of electrical contacts. The contacts are in one of two mechanically controlled states: closed or open. When closed, the contacts are electrically connected. When open, the contacts are electrically disconnected. (Wikipedia: Switch).

In digital system design, it helps to think of an abstraction of a switch. A switch is a component with a single bit output that is either a 0 or 1 depending on whether the switch is in the off or on position.



A push button operates similar to a simple switch, having a pair of electrical contacts and two mechanically controlled states: *closed* or *open*. Unlike a simple switch, the push button enters its closed state when it is being *pressed*. The moment the pressing force is removed, the push button reverts to and remains in its open state. (Wikipedia: Push_button).

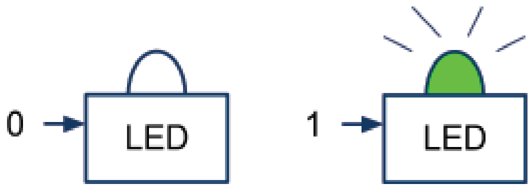
An abstraction of a push button is a component with a single bit output that is 0 when the button is not pressed, and that is 1 while the button is pressed.



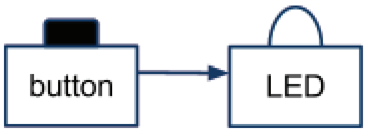
LED

A light emitting diode (LED) is a semiconductor with a pair of contacts. When a small electrical current is applied to the LED contacts, the LED illuminates. (Wikipedia: LED).

An abstraction of an LED is a component with a single bit input that can be either 0 or 1. When the input is 0, the LED does not illuminate. When the input is 1, the LED illuminates.



We can build a simple system that is composed of a push button and an LED connected as shown below. When the button is pressed, the LED will illuminate.



This system falls short of being an embedded system because it lacks computing functionality. For example, the system can't be easily modified to toggle the LED each time the button is pressed, or to illuminate the LED when the button is pressed AND a switch is in the on position. A component executing some computing functionality is a key part of an embedded system.

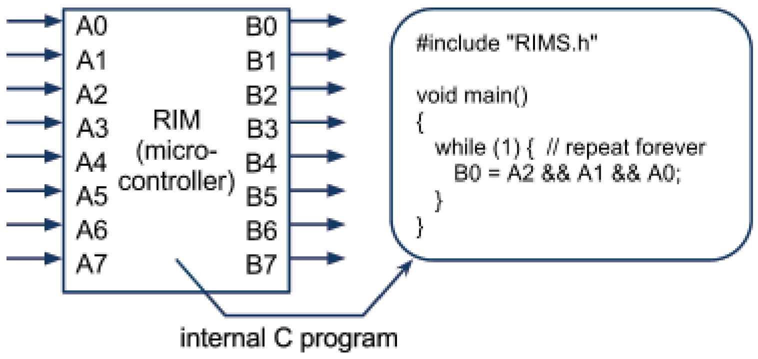
Microcontroller

A **microcontroller** is a programmable component that reads digital inputs and writes digital outputs according to some internally-stored program that computes. Hundreds of different microcontrollers are commercially available, such as the PIC, the 8051, the 68HC11, or the AVR. A microcontroller contains an internal program memory that stores machine code generated from compilers/assemblers operating on languages like C, C++, Java, or assembly language.

A "PIC" microcontroller

(Wikipedia: Microcontroller Wikipedia: Atmel AVR Wikipedia: C language)

We will use an abstraction of a microcontroller, referred to as **RIM** (Riverside-Irvine Microcontroller), consisting of eight bit-inputs A0, A1, ..., A7 and eight bit-outputs B0, B1, .., B7, and able to execute C code that can access those inputs and outputs as implicit global variables (this book assumes reader proficiency with C programming).

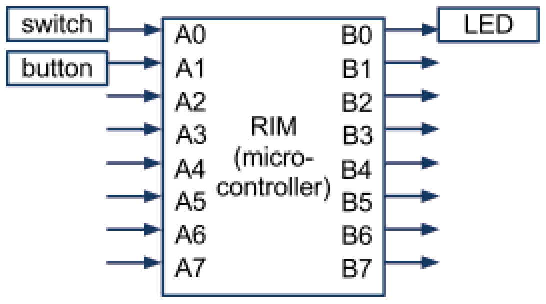


```
#include "RIMS.h"

void main()
{
  while (1) {  // repeat forever
    B0 = A2 && A1 && A0;
  }
}
```

The RIM executes C code

The example statement "B0 = A2 && A1 && A0" sets the microcontroller output B0 to 1 if inputs A2, A1, and A0 are all 1. The "`while (1) { <statements> }`" loop is a common feature of a C program for embedded systems and is called an **infinite loop**, causing the contained statements to repeat continually.

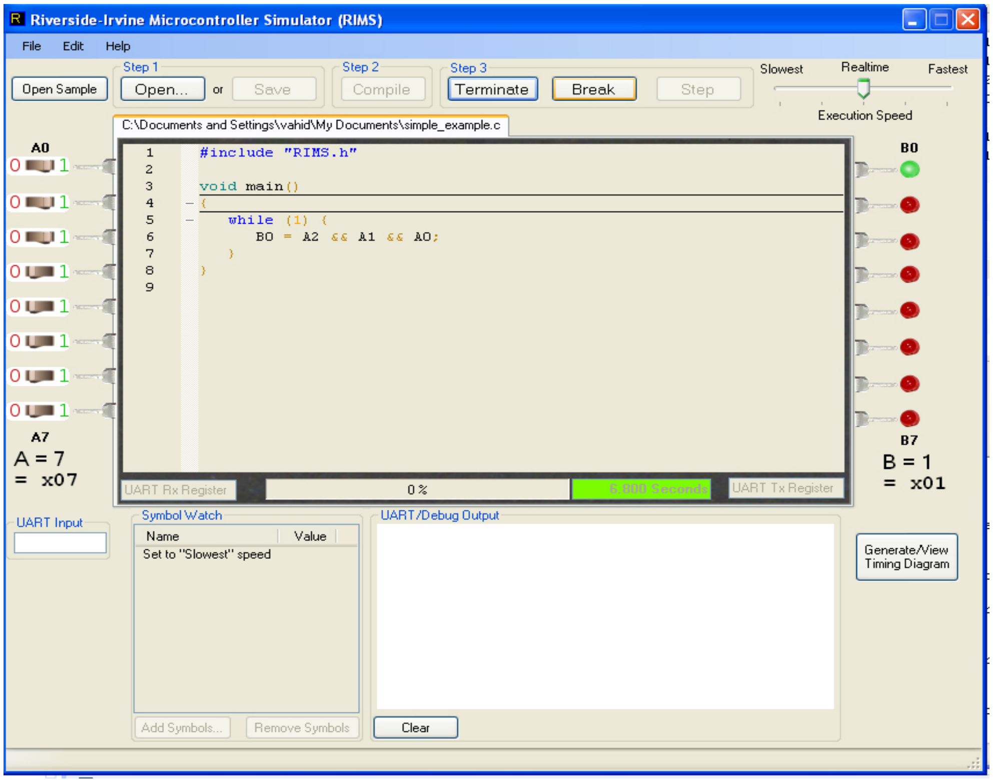
We can use a microcontroller to add functionality to the earlier simple system to create an embedded system. The term embedded system, however, commonly refers just to the compute component. The switch and buttons are examples of **sensors**, which convert physical phenomena into digital inputs to the embedded system. The LED is an example of an **actuator**, which converts digital outputs from the embedded system into physical phenomena. (Wikipedia: Sensor) (Wikipedia: Actuator)



The RIM connected to switches, buttons, and LEDs.

RIMS

RIMS (RIM simulator) is a graphical PC-based tools that supports C programming and simulated execution of RIM. RIMS is useful for learning to program embedded systems. A screenshot of RIMS is shown below. The eight inputs A0-A7 are connected to eight switches, each of which can be set to 0 or 1 by clicking on the switch. The eight outputs B0-B7 are connected to eight LEDs, each of which is red when the corresponding output is 0 and green when 1.



C:\Documents and Settings\vahid\My Documents\simple_example.c

```
#include "RIMS.h"

void main()
{
    while (1) {
        B0 = A2 && A1 && A0;
    }
}
```

Name	Value
Set to "Slowest" speed	

C code can be written in the center text box. `#include "RIMS.h"` *is required atop all C files for RIMS*. The user can first press the “Save” button to save the C code, then press “Compile” (to translate the C code to executable machine code, which is hidden from the user), then press “Run” (after which the button name changes to “Terminate” as in the above figure).

While the program is running, the user can click on the switches on the left to change each of RIM’s eight input values to 0 or 1. RIM’s eight output values, written by the running C code, set each LED on the right to green (for 1) or red (for 0). When done, the user should press “Terminate”.

Try: Download, install, and execute RIMS (see www.programmingembeddedsystems.com). Note that a default C program appears in the center text box. Replace the default C program by the program below. Press “Save” and name the file “example1.c”, then press “Compile”, press “Run”, and then click switches for A1 and A0 to 1, noting that B0 becomes 1 and its LED turns green. Press “Terminate” when done.

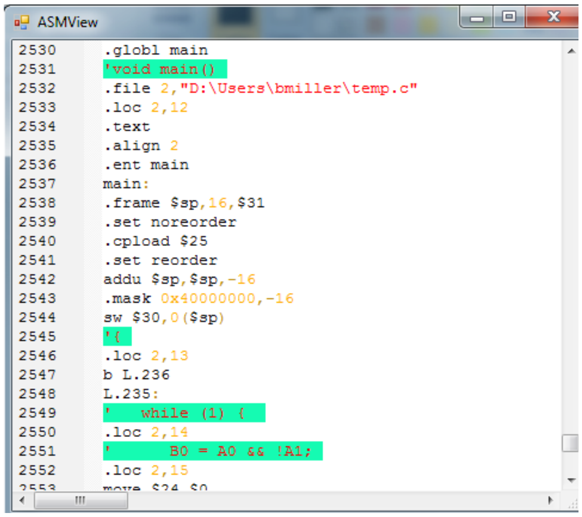
```
#include “RIMS.h”

void main()
{
   while (1) {
      B0 = A1 && A0;
```

```
        }
}
```

Try: Write a C program for RIM that sets B0=1 whenever the numbers of 1s on A2, A1, and A0 is two or more (i.e., when A2A1A0 are 011, 110, 101, or 111). Hint: Use logical OR (||). Run the program in RIMS to test the program.

Pressing "Break" temporarily stops the running (and the button changes to "Continue") and shows an arrow next to the current C statement, and then each press of "Step" executes one C statement. Pressing "Continue" resumes running. Pressing "Terminate" ends the program, and re-enables editing of the C code. A box under the C code shows how many seconds the C program has run.

Try: For the above program named "example1.c", set A1A0 to 00, run the program, and press "Break". Press "Step" 5 times and observe the arrow pointing to the current statement after each press. Now change A1A0 to 11, and then press "Step" several times until B0 changes. Press "Continue" to resume running. Press "Terminate" to end the running.



```
.globl main
'void main()
.file 2,"D:\Users\bmiller\temp.c"
.loc 2,12
.text
.align 2
.ent main
main:
.frame $sp,16,$31
.set noreorder
.cpload $25
.set reorder
addu $sp,$sp,-16
.mask 0x40000000,-16
sw $30,0($sp)
'{
.loc 2,13
b L.236
L.235:
'    while (1) {
.loc 2,14
'        B0 = A0 && !A1;
.loc 2,15
```

During the compilation process, the C code is translated to a low level assembly language known as MIPS (Wikipedia:MIPS). The assembly code contains the actual instructions to be executed by the simulator. Assembly can be viewed once the "Compile" button has been pressed by selecting "View" in the menu bar of RIMS, and then selecting the "View Assembly" option. The assembly code is annotated with highlighted C code from the compiled program, as shown in the screen capture. If the assembly view is open, than pressing the "Step" button will execute a single assembly instruction.

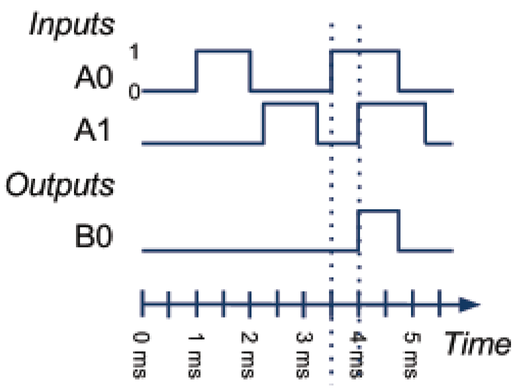
RIMS' Execution Speed slider (upper right) can be moved left to slow running speed; the "Slowest" setting causes an arrow to appear next to each C statement as it executes.

The user can click "Add Symbols" (at bottom of RIMS) to see the current value of any input, output, or global variable in the C code.

Numerous samples that introduce features can be found by pressing "Open Sample" (upper left). Other features will be described later.

Timing diagrams

An embedded system operates continually over time. A common representation of how an embedded system operates (or should operate) is a timing diagram. A **timing diagram** (Wikipedia: Digital Timing Diagram) shows time proceeding to the right, and plots the value of bit signals as either 1 (high) or 0 (low). The figure below shows sample input values for the above example.c program that continually computes B0 = A1 && A0. A0 is 0 from time 0 ms to 1 ms, when it changes to 1. A0 stays 1 until 2 ms, when it changes to 0. And so on. The 1 and 0 values are labeled for signal A0, but are usually implicit as for A1.



The timing diagram shows that B0 is 1 during the time interval when both A0 and A1 are 1, namely between 4 ms and 5 ms.

Vertical dotted lines are sometimes added to help show how items line up (as done above) or to create distinct timing diagram regions.

Try: Draw a timing diagram showing all possible combinations of three single-bit input signals A0, A1, and A2. Use vertical dotted lines to delineate each combination.

Try: A program should set B0 to 1 if exactly two of A0, A1, and A2 are 1. Draw a timing diagram illustrating this behavior.

A change from 0 to 1 or from 1 to 0 on a bit signal is called an **event.** The above figure has 10 events. If a signal changes from 0 to 1, the event is called **rising**. 1 to 0 is called **falling**.

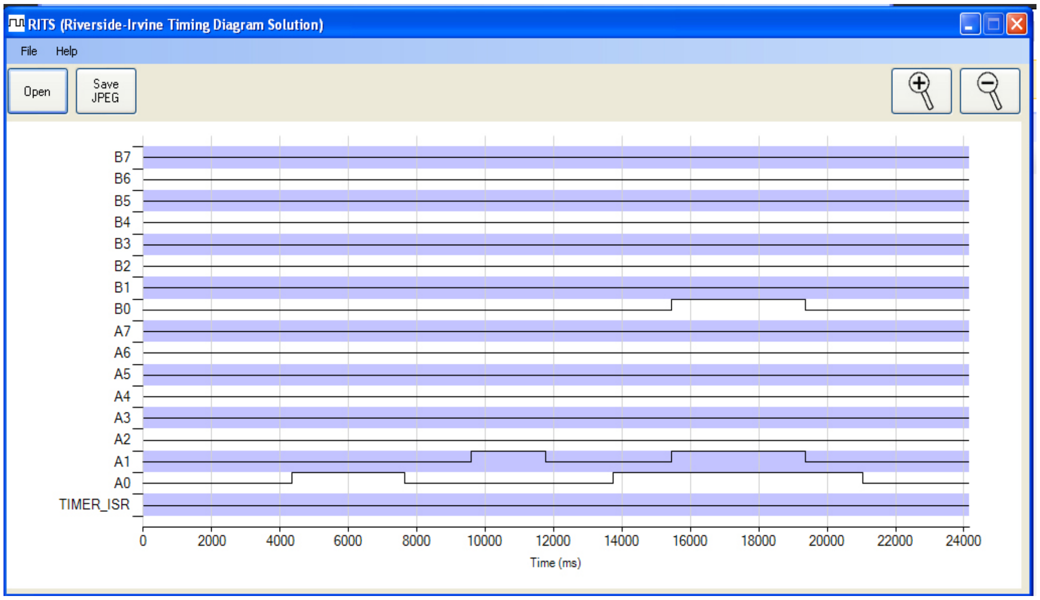
Try: Circle all ten events in the above timing diagram.

Testing

Written code should be tested for correctness. One method is to generate different input values and then observe if output values are correct. To test code implementing "B0 = A0 && !A1", *all possible input value combinations* of A1 and A0 can be generated: 00, 01, 10, and 11. Using RIMS, switches can be clicked to generate each desired input value. First switches for A1 and A0 can both be set to, then A0 can be set to 1, then A0 can be set to 0 and A1 to 1, and finally both A1 and A0 can be set to 1. B0 should only output 1, and hence B0's LED should only turn green, in the second case.

For most code, there are too many possible input combinations to test all of them. Testing should cover **border cases** such as all inputs being 0s and all inputs being 1s, and then several *sample normal cases*. For example, completely testing "B0 = A0 && A1 && A2 && A3 && A4 && A5 && A6 && A7" would require 256 unique input value combinations. Border and sample testing might instead test two borders, A7..A0 set to 00000000 (output should be 0) and to 11111111 (output should be 1), and then a few (perhaps a dozen) sample normal cases like 00110101 or 10101110. If code has branches, then good testing also ensures that every statement in the code is executed at least once, known as *100% code coverage*. (Wikipedia: Software Testing) (Wikipedia: Software Debugging)

RIMS records all input/output values textually over time. That text can be analyzed for correct code behavior, rather than observing RIMS LEDs. Pressing the "Generate/View Timing Diagram" button while a program is running (or in a "break" status) automatically saves those textual input/output values in a file and then runs the timing-diagram viewing tool called **RITS** (Riverside-Irvine Timing-diagram Solution).



RITS timing diagram from running the "B0 = A0 && !A1" program.The user can zoom in or out using the "+" and "-" buttons on the top right, and scroll using the scrollbar at the bottom. The user can save the currently shown portion of the timing diagram using the "Save JPEG" button.

The saved text file is an industry standard vcd (value change dump) file (Wikipedia: Value Change Dump file). A vcd file can also be read by many other timing diagram tools. RITS can also be run on its own, and can open vcd files generated by other tools.

Ideally, input value combinations, known as **test vectors**, could be captured in a file and then input to a tool rather than each input value combination being generated by clicking on switches. For tool simplicity and ease of use, RIMS does not presently support such file input.

Try: Save, compile, and run example1.c from above. Click the input switches to achieve the following values: A1A0=00, then 01, then 00, then 10, then 00, then 01, then 11, then 00. Press "Break", then press the "Generate/View Timing Diagram" button, causing a timing diagram window to appear, and observe how the timing diagram corresponds to the output values you observed just prior (and should closely match the above RITS figure). Press RITS' "Save JPEG" button to save the timing diagram to a JPEG file. Open the saved JPEG file using a picture viewing tool (not included with the RI tools). Finally, back on RIMS, press the "Terminate" button.

A helpful debugging practice is the use of **trace statements** to track program execution. For example, a message "Entering foo()" could be used to help identify the currently executing subroutine as foo(). RIMS supports the standard printf() function to support trace statements. The basic method for printing a trace statement is to use 'printf("Hello");'. printf() supports more complicated arguments for printing the values of different data types, like char, int, etc (Wikipedia: printf). Printed items appear in the terminal output window of RIMS.

Try: Open RIMS and edit the default program by adding the following line inside the while(1) loop: 'printf("B=%d\n",B);'. Press "Compile" and "Run" to begin the program. Note that the terminal window will output the value of B while the program is running.

Exercises

1. Write RIM C code that sets B0 to 1 only if A0-A3 are all 1s or if A4-A7 are all 1s (or if both situations are true). Using border and sample input value combinations, test the written code with RIMS, and generate a timing diagram showing the test results.

2. A car has a sensor connected to A0 (1 means the car is on), another sensor connected to A1 (1 means a person is in the driver's seat), and a sensor connected to A2 (1 means the seat belt is fastened). Write RIM C code for a "fasten seat belt" system that illuminates a warning light (by setting B0=1) when the car is on, a driver is seated, and the seat belt is *not* fastened. Test the written code with RIMS for all possible input combinations of A2, A1, A0, and generate a timing diagram showing the test results.

Chapter 2: Bit-Level Manipulation in C

C is a popular programming language in embedded systems due to its simplicity and efficiency, but C was not originally created for embedded systems. C was created in 1972 for mainframe and desktop computers, which typically manipulate data in files such as integer or character data. In contrast, embedded systems commonly manipulate bits, in addition to other data types. This chapter describes C's built-in data types and discusses how to manipulate bits in C (Wikipedia: C language). Most of the discussion applies equally to the C++ language.

Language	% of embedded systems programmers whose current project is mostly in this language (2011)
C	62%
C++	22%
Assembly	5%
Java	2%
UML/Matlab/similar	2%
.NET	1%
Basic	1%
XML	1%
Other	3%

Source: www.embedded.com, 2011 State of Embedded Market Survey.

C data types

Several C data types are commonly used to represent integers in embedded system programs:

Type	Width	Range	Notes
`signed char`	8	-128 to +127	
`unsigned char`	8	0 to 255	
`signed short`	16	-2^{15} to $+2^{15}-1$	2^{16} is 65,536 (aka 64K)
`unsigned short`	16	0 to $+2^{16}-1$	
`signed long`	32	-2^{31} to $+2^{31}-1$	2^{32} is about 4 billion (aka 4 Gig)
`unsigned long`	32	0 to $+2^{32}-1$	
~~**`signed int`**~~	N	-2^{N-1} to $+2^{N-1}-1$	*Though commonly used, we avoid these due to*
~~**`unsigned int`**~~	N	0 to $2^{N}-1$	*undefined width*

Thus, a variable whose value may only range from 0 to 100 might be best declared as "unsigned char". A variable whose value may only range from -999 to 999 might be best declared as "signed short". Using the smallest possible data type ensures that the limited space in typical embedded system designs is conserved. Note that "unsigned" data types represent positive integers, while "signed" types represent negative and positive integers. If a variable is used to represent a series of bits (rather than a number), then an unsigned type should be used.

Embedded systems commonly deal with **1-bit** data items. C does not have a 1-bit data type, which is unfortunate. Thus, 1-bit items are typically represented using an unsigned char, e.g., "`unsigned char myBitVar;`". The programmer only assigns the variable with either 0 or 1, e.g., "`myBitVar = 1;`", even though the variable could be assigned integers up to 255.

Checking whether such a variable is 1 or 0 is typically done without explicit comparison to 1 or 0, and is instead done as "`if (myBitVar)`" or "`if (!myBitVar)`".

Below are some example variable declarations:

```
unsigned char  ucI1;
unsigned short usI2;
signed   long  slI3;
unsigned char  bMyBitVar;
```

A common practice is to name variables with a lower-case prefix indicating the data type, as above -- uc, us, and ul for unsigned char, short, and long; sc, ss, and sl for signed char, short, and long; or b for bit -- to help ensure that larger constants or variables aren't assigned to smaller variables. For clarity of short examples, this book often skips that practice, but programmers of larger projects should consider it.

char is called such because it is commonly used in desktop programming to represent the integer value of an 8-bit ASCII *character*. Note however that char is actually an integer. Also, 8-bits is sometimes called a *byte*.

In C, the word "signed" is optional for a signed type, so "`char I1;`" is the same as "`signed char I1;`". However, for program clarity, we avoid that shortcut. Also, the word "int" may follow the words short or long, but that word is superfluous so we usually omit it.

Unfortunately, although the above widths are quite common, C actually defines the above widths as *minimum* widths, so a compiler could for example create a long as 64 bits. Thus, a programmer should never assume an exact width, e.g., a program should not increment an "unsigned char" and expect it to roll over from 255 to 0, because the char could be 16 bits. Another unfortunate fact is that C allows a variable to be declared merely as type "int", where the width is compiler dependent. *Due to the unpredictability of int, we avoid using the int type almost entirely*. Following these conventions improves code **portability**, which is the ability to recompile code for a different microprocessor without undesirable changes in program behavior.

The underlying representation of each data type is binary. For an 8-bit unsigned char ucI1:

```
ucI1 = 1;   // underlying bits will be 00000001
ucI1 = 12;  // underlying bits will be 00001100
ucI1 = 127; // underlying bits will be 01111111
ucI1 = 255; // underlying bits will be 11111111
```

In binary, the rightmost bit has weight 2^0, the next bit 2^1, then 2^2, etc. In other words, from left to right, the 8 bits have weights 128, 64, 32, 16, 8, 4, 2, and 1. 0001100 is thus 8 + 4 = 12.

Signed data types in C use two's complement representation. At the bit level, a variable of type *char* set to 127 would have an internal representation of 01111111, while -128 would be 10000000, and -1 would be 11111111. For the curious reader -- the binary representation of a negative number in two's complement can be obtained by representing the number's magnitude in binary, complementing all the bits, and adding 1. For example, -1 is 00000001 (magnitude is 1) --> 11111110 (complement all bits) --> 11111110+1 = 11111111 (add 1). -128 is 10000000 (magnitude is 128) --> 011111111 (complement all bits) --> 011111111+1 = 100000000 (add 1). Note that the eighth bit will always be 1 for a negative 8-bit number and is thus called the *sign bit*. The programmer generally need not deal directly with the binary representations of signed numbers, because compilers/assemblers automatically create the proper constants (e.g., "myVar = -1;" would store the two's complement representation in myVar). However, knowing whether an item is signed or unsigned is important when assigning values, and for determining a variable's range. (Wikipedia: Two's complement)

The following provides some examples of choosing an appropriate data type for a variable based on the variable's intended purpose:

Purpose	Variable declaration
Store a person's age in years	unsigned char age; // Not <0, not >255
Store an airplane's speed	unsigned short speed; // Not <0, not >64K
Store the remaining joules of energy in a car battery	unsigned long energy; // Not <0, not >4Gig
Store feet of elevation above/ below sea level of land	signed short elevation; // Could be <0, not >64K // or <64K.

RIMS implicitly defined input/output variables

In RIMS, the microcontroller's inputs and outputs are implicitly defined as global variables "unsigned char A0;", "unsigned char A1;", ..., "unsigned char B0;", etc. As mentioned above, an item intended to represent a single bit, such as B0 in RIMS, should be set to only 0 or 1, e.g., "B0 = 1;".

RIMS also implicitly defines two additional global variables A and B:

```
unsigned char A; // built-in variable A, representing
                 // RIM's 8 inputs as a single variable

unsigned char B;// built-in variable B, representing
                // RIM's 8 outputs as a single variable
```

Thus, setting all of RIM's outputs to 1s can be accomplished by one statement, "B = 255;" (255 is "11111111" in binary), rather than the eight statements: "B0 = 1; B1 = 1; ...; B7 = 1;" To set RIM's outputs to the number 7 in binary (00000111), the statement "B = 7;" could be used. To set RIM's outputs to RIM's inputs, the statement "B = A;" could be used. Such grouping of bits is uncommon in desktop programming, but quite standard in C environments for microcontrollers.

Because B is a global variable, a program can write as well as read that variable. However, A is automatically written by the microcontroller and should never be written by a program, only read. In RIMS, writing to A results in a runtime error, causing execution to terminate.

Try: Write a C program in RIMS that repeatedly executes "B = 7;" Note that outputs B2, B1, and B0 become 1s, because 7 is 00000111 in binary. Set the input switches such that A3=1, A2=0, A1=0, and A1=1, with the other inputs 0, and note below the input pins that RIMS indicates the value of A to be 9, because 00001001 is 9 in binary.

Try: Write a C program for RIMS that sets B equal to A plus 1.

Try: Write a C program for RIMS that sets B = 300. Note that the value actually output on B is not 300, because an unsigned char has a range of 0 to 255.

Variables can be *initialized* when declared, e.g., "unsigned char i1 = 5;".

The keyword "const," short for constant, can precede any variable declaration, e.g., "`const unsigned char i1 = 5;`". A **constant variable**'s value cannot be changed by later code, and thus can help to prevent the introduction of future errors. A constant variable *must* therefore be initialized when declared. Above, 5 is a constant, and i1 is a constant variable.

Try: A car has a sensor that sets A to the passenger's weight (e.g., if the passenger weighs 130 pounds, A7..A0 will equal 10000010). Write a RIM C program that enables the car's airbag system (B0=1) if the passenger's weight is 105 pounds or greater. Also, illuminate an "Airbag off" light (by setting B1=1) if weight > 5 pounds but weight < 105 pounds.

Hexadecimal

Commonly an 8-bit unsigned item isn't used as a number but rather just a eight distinct bits. For example, if RIM's eight outputs connected to eight light bulbs and we wanted to light all the bulbs, we could write "B = 255;" (because 255 is 11111111 in binary), but "255" does not directly convey our intent. Ideally, we could write "B = b11111111;" or something similar, but C unfortunately has no binary constant support. But fortunately, C does support hexadecimal constants, which are close to the ideal.

Hexadecimal (or "**hex**") is a base 16 number, where each digit can have the value of 0, 1, ..., 8, 9, A, B, C, D, E, or F. A is ten, B is eleven, C is twelve, D is thirteen, E is fourteen, and F is fifteen. Hex values have the following binary representations: **0:0000, 1:0001, 2:0010, 3:0011, 4:0100, 5:0101, 6:0110, 7:0111, 8:1000, 9:1001, A:1010, B:1011, C:1100, D:1101, E:1110, F:1111**. In the C language, a hex constant is preceded by *"0x"* (the first character 0 is a zero, not the letter O). Thus, "0xFF" represents "11111111" in binary. *Each hex digit corresponds to four bits* (four bits is called a **nibble**). "0xff" may also be used; hex constants are not case sensitive. (Wikipedia: Hexadecimal)

Thus, "B = 0xFF;" sets all RIM outputs to 1s. The intent of 0xFF is clearer than 255. Likewise, "B = 0xAA" sets the outputs to 10101010, having *much* clearer intent than "B = 170;".

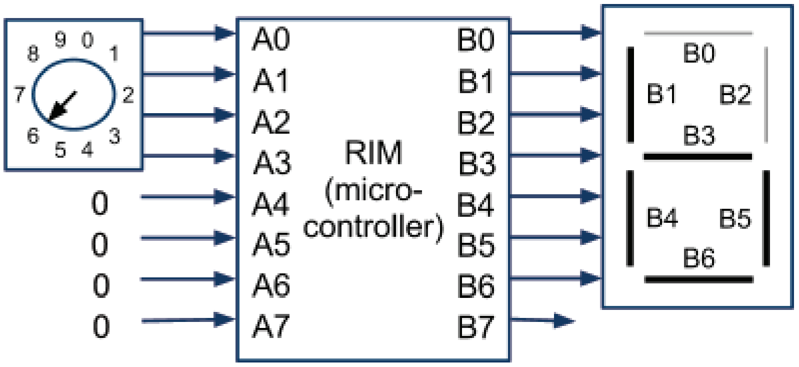
Try: Write a single statement for RIM that sets B7-B4 to 1s and B3-B0 to 0s, using a hex constant.

Try: Write a single statement for RIM that sets B0 to 1 if all eight A inputs are 1s, using a hex constant.

Example: The following program sets B to 00000000 when A1A0=00, to 01010101 when A1A0=01, to 10101010 when A1A0=10, and to 11111111 when A1A0=11:

```
#include "RIMS.h"

// B=00000000 when A1A0=00, B=01010101 when A1A0=01,
// B=10101010 when A1A0=10, and B=11111111 when A1A0=11
void main()
{
   while (1) {
      if (!A1 && !A0) {
         B = 0x00; // 0000 0000
      }
      else if (!A1 && A0) {
         B = 0x55; // 0101 0101
      }
      else if (A1 && !A0) {
         B = 0xAA; // 1010 1010
      }
      else if (A1 && A0) {
         B = 0xFF; // 1111 1111
      }
   }
}
```

Example: Consider the following embedded system with a dial that can set A3..A0 to binary 0 to 9, and a 7-segment display (Wikipedia: 7-Segment Display) connected to B6..B0 as shown:



Below is a (partial) RIM C program that appropriately sets the display for the given dial position:

```
#include "RIMS.h"

void main()
{
   while (1) {
      switch( A )
      {
         case 0 : B = 0x77; break; // 0111 0111 (0)
         case 1 : B = 0x24; break; // 0010 0100 (1)
         case 2 : B = 0x5d; break; // 0101 1101 (2)
         //...
         case 9 : B = 0x6f; break; // 0110 1111 (9)
         default: B = 0x6b; break; // 0101 1011 (E for Error)
      }
   }
}
```

Try: Complete the above program using hex constants to produce the correct display for dial settings 3..8. Test in RIMS.

Bitwise operators and masks

An important programming skill for an embedded C programmer is manipulating bits within an integer variable. For this, C's bitwise operators are needed:

- **& : bitwise AND**
- **| : bitwise OR**
- **^ : bitwise XOR**
- **~ : bitwise NOT**

Bitwise operators operate on the operands' corresponding bits, as shown:

```
 0x0F & 0xF2:   0x0F | 0xF2:    0x0F ^ 0xF2:    ~0x0F:
   00001111       00001111        00001111       00001111
 & 11110010     | 11110010      ^ 11110010      ~
-----------     -----------     -----------     ----------
  00000010        11111111        11111101       11110000
```

Try: Compute "0xAA *OP* 0x33" for OP being &, then |, then ^.

In contrast, Boolean operators &&, ||, and ! (there is no Boolean XOR operator) treat operands as zero (false) or non-zero (true). So "0x0F *&* 0xF0" (bitwise AND) evaluates to 0 because each AND of corresponding operand bits evaluates to 0, whereas "0x0F *&&* 0xF0" (Boolean AND) evaluates to 1 because both operands are non-zero. Likewise, if unsigned char x has the value of 1, then ~x is "11111110", while !x is just 0 (the "!" of any non-zero value becomes zero).

Below are some examples of using bitwise operators:

```
B = ~A; // Invert each bit in A, e.g., 00001111 -> 11110000
B = A & 0x0F; // Sets B3..B0 to A3..A0, and B7..B4 to 0000
B = A | 0xF0; // Sets B3..B0 to A3..A0, and B7..B4 to 1111
B = A & 0xF7; // Sets B to A, except that bit #3 set to 0
B = A | 0x04; // Sets B to A, except that bit #2 set to 1
x = x | 0x04; // For unsigned char x, sets bit #2 to 1
if (A & 0x03) { // Sets B0 to 1 if A1A0 are 11
   B0 = 1;
}
else {
   B0 = 0;
}
```

(Note: bit #0 refers to the least significant bit, *LSB*, so bit #2 is the third bit from the right)

Try: Write a single C statement for RIM that sets B to A except that bit #7 and bit #6 are set to 1s.

A **mask** is a constant value having a desired pattern of 0s and 1s, typically used with bitwise operators to manipulate a value. In the above examples, 0x0F, 0xF0, 0xF7, and 0x04 are masks. Masks are typically used based on the following ideas (below, assume "a" is a single bit):

- To force a bit position to 0, AND with a mask having 0 in that position (a & 0 = 0)
- To force a bit position to 1, OR with a mask having 1 in that position (a | 1 = 1)
- To pass a bit position through, AND with a mask having 1 in that position (a & 1 = a), or OR with a mask having 0 in that position (a | 0 = a)

Masks are sometimes defined as constant variables:

```
const unsigned char MaskLoNib1s = 0x0F;
B = A | MaskLoNib1s; // Passes high nibble, sets low
                     // nibble to 1111
```

The term *mask* comes from the role of letting some parts through while blocking others, like a mask someone wears on his face letting the eyes and mouth through while blocking other parts.

Two more bitwise operators are commonly used:

- **<< : left shift**
- **>> : right shift**

For unsigned integer types, shift operators move their first operand's bits left/right by the number of positions indicated by their second operand (the shift amount), as shown:

```
0x0F << 2:       0x0F >> 3:
   00001111          00001111
<< 2             >> 3
------------     ------------
   00111100              00000001
```

Note that vacated positions on the right (for left shift) or left (for right shift) have 0s shifted in. Below are some examples of using shift operators:

```
B = A << 1; // Sets B7=A6, B6=A5, ..., B1=A0, B0=0
B = A >> 4; // Sets B7..B4=0000, and B3..B0=A7..A4
B = A & (0x0F << 2); // Passes A's middle bits to B
```

The shift amount should be between 0 and the number of left-operand bits, inclusive.

Try: Compute "B = A << 6;" and "B = A >> 5" for A being 0xFE.

Try: Test the above shift operator examples using RIMS, by creating a distinct program for each.

Try: Write a single C statement for RIM that sets B3-B0 to A5-A2 and sets other output bits to 0s.

Example: The following program treats A7..A0 as one 4-bit binary number and A3..A0 as another 4-bit binary number, and outputs the sum of those two numbers on B:

```
#include "RIMS.h"

const unsigned char LoNib1s = 0x0F;
const unsigned char HiNib1s = 0xF0;

unsigned char op1;
unsigned char op2;

void main()
{
   while (1) {
        op1 = A & LoNib1s; // 0000a3a2a1a0
        op2 = (A & HiNib1s) >> 4; //a7a6a5a40000 -->  0000a7a6a5a4
        B = op1 + op2;
    }
}
```

Try: Run the above program. Press "Break", then add symbols op1 and op2 to symbols being watched. Press "Continue" and then move the speed slider to "Slowest." Now set values on A3..A0 and A7..A4 and observe their values in the symbol watch area, e.g., set A3..A0 to 0011 and set A7..A0 to 0100, and note that op1 is 3 and op2 is 4, with B being 7.

Try: A parking lot has eight spaces, each with a sensor connected to A7..A0 (1 means a car is detected in the space). Spaces A7 and A6 are reserved handicapped parking. Write a RIM C program that: (1) Sets B0 to 1 if both handicapped spaces are full, and (2) Sets B7..B5 equal to the number of available non-handicapped spaces.

Shifting can be performed on signed integer types too, but we do not recommend such use. Such shifting was previously popular because shifting a binary number left or right is equivalent to multiplying or dividing by 2, respectively (just as shifting a decimal number left or right is equivalent to multiplying or dividing by 10), and shifting could result in faster code execution than the slower * and / operations on some processors. However, modern compilers automatically replace * and / by shifts when possible, so today the programmer can emphasize understandable code rather than such low-level speedup attempts. For the curious reader, shifting a signed number performs an "arithmetic" shift that preserves the number's sign, rather than a "logical" shift that merely shifts all bits. We will never shift signed types (Wikipedia: Arithmetic Shift Wikipedia: Logical Shift).

Bit access functions

Defining C functions that perform common bit manipulation tasks can be quite useful.

The following function returns a value in which the kth bit of an unsigned char x is set to 1:

```
unsigned char set_bit1(unsigned char x, unsigned char k)
{
     return (x | (0x01 << k));
}
```

The mask 0x01 is shifted left k positions to get the sole 1 bit into the kth position, and then bitwise ORed with x so that in the result the kth bit is 1 (because single-bit a | 1 = 1) while x's bits pass through to the remaining positions (because a | 0 = a). If k is 2, the mask will be shifted to become 00000100. Recall that the rightmost bit is position 0, not position 1. Similar functions can be created for short or long integer types.

The following function returns a value in which the kth bit of an unsigned char x is set to 0:

```
unsigned char set_bit0(unsigned char x, unsigned char k) {
    return (x & ~(0x01 << k));
}
```

The mask 0x01 is shifted left k positions to get the sole 1 bit into the kth position, and then bitwise complemented using "~" to yield a mask with a 0 in the *k*th bit and 1s in the other bits. For example, when *k* is 1, the shifted constant will be 00000010, which will then be complemented into 11111101. The resulting mask is bitwise ANDed with x, so that in the result the kth bit is 0 (because single-bit a & 0 = 0), while x's bits pass through to the remaining bits (because a & 1 = a).

Example: The following statements set B to A except for setting B7 to 0:

```
unsigned char tmp;
tmp = A;
tmp = set_bit0(tmp, 7);
B = tmp;
```

The above functions can be used to create another function that takes as a parameter the bit value to be set .

```
unsigned char SetBit(unsigned char x, unsigned char k,
                     unsigned  char b)
 {
   return (b ? set_bit1(x, k) : set_bit0(x, k));
 }
```

The function uses C's **ternary conditional operator (?:)** , which returns its second operand when the first operand is non-zero, else it returns its third operand. So for "m = (n<5) ? 44 : 99;", if n < 5 then m will be assigned 44, else m will be assigned 99.

An example of using the SetBit function involves setting all B bits to the value of A0:

```
unsigned char i;
for (i=0; i<8; i++) {
   B = SetBit(B, i, A0);
}
```

Finally, the following function gets (rather than sets) the value of a particular bit in an integer variable:

```
unsigned char GetBit(unsigned char x, unsigned char k) {
```

```
          return ((x & (0x01 << k)) != 0);
      }
```

The function creates a mask *m* containing a 1 in position *k* and 0s in all other positions, then performs a bitwise AND to pass the kth bit of x through, resulting either in a zero result if the kth bit was 0, or a non-zero result if the kth bit was 1. The function compares the result with 0, then returning either a 1 or a 0.

Example: A parking lot has eight parking spaces, each with a sensor connected to input A. The following program sets B to the number of occupied spaces, by counting the number of 1s using the GetBit function:

```
#include "RIMS.h"

      unsigned char GetBit(unsigned char x, unsigned char k) {
         return ((x & (0x01 << k)) != 0);
      }

void main()
{
   unsigned char i;
   unsigned char cnt;
   while (1) {
      cnt=0;
      for (i=0; i<8; i++) {
         if (GetBit(A, i)) {
            cnt++;
         }
      }
      B = cnt;
   }
}
```

Note that the above bit access functions do not perform error checking (e.g., you can attempt to set the 9th bit of a variable).

The examples using the SetBit and GetBit functions may seem inefficient due to computing the mask, but today's optimizing compilers handle these very efficiently. Furthermore, the **inline** keyword can be prepended to each function declaration (e.g., "inline unsigned char GetBit(...)") to encourage compilers to inline the function calls (though many compilers would do so anyways). **Inlining** means to replace a function call by the function's internal statements. Compiler optimizations may then eliminate most of the statements within the GetBit and SetBit functions.

A programmer may wish to copy the bit-access functions to the top of a file whose program performs bit manipulation, as in the below example program that copies A's bits to B's bits in reverse order (with function SetBit performing the set to 1 and 0 directly rather than using the set_bit1 and set_bit0 functions),

```
// Bit-access functions
inline unsigned char SetBit(unsigned char x, unsigned char k,
                            unsigned char b)
{
   return (b ? x | (0x01 << k) : x & ~(0x01 << k));
}

inline unsigned char GetBit(unsigned char x, unsigned char k) {
   return ((x & (0x01 << k)) != 0);
}

void main(){
   unsigned char i;
   while(1){
      for (i=0; i<8; i++) {
         B = SetBit(B, 7-i, GetBit(A,i));
      }
   }
}
```

(Wikipedia: Bit Manipulation Wikipedia: Bitwise Operation Wikipedia: Mask
Wikipedia: ?: (Ternary) Operator)

Try: Write a C program for RIMS that sets B0=1 if a sequence of three consecutive 1s appears anywhere on input A (e.g., 11100000 and 10111101 have such sequences, while 11001100 does not), using a C *for* loop and the GetBit() function.

Exercises

Assume the RIMS environment for all exercises below.

1. Write a C program to set B3-B0 to A7-A4, and B7-B4 to A3-A0.
2. Write a C program that treats A1A0, A3A2, and A5A4 as three 2-bit unsigned binary number. The program should output the sum of those three numbers onto B.
3. Write a C program that rotates input A right once and outputs the result on B (rotate right is the same as shift right, except the LSB becomes the MSB). Use a variable X to store A, and use the right shift operator to avoid having to explicitly set each bit with a unique statement.
4. Write C statements that set B to the reverse of A, such at B7 = A0, B6 = A1, etc. Rather than writing 8 assignment statements, instead write a for loop that makes use of the GetBit and SetBit functions.

5. Write a C program that interprets the input A as an 8-bit unsigned binary number representing a temperature in Fahrenheit, and outputs the temperature in Celsius on B.