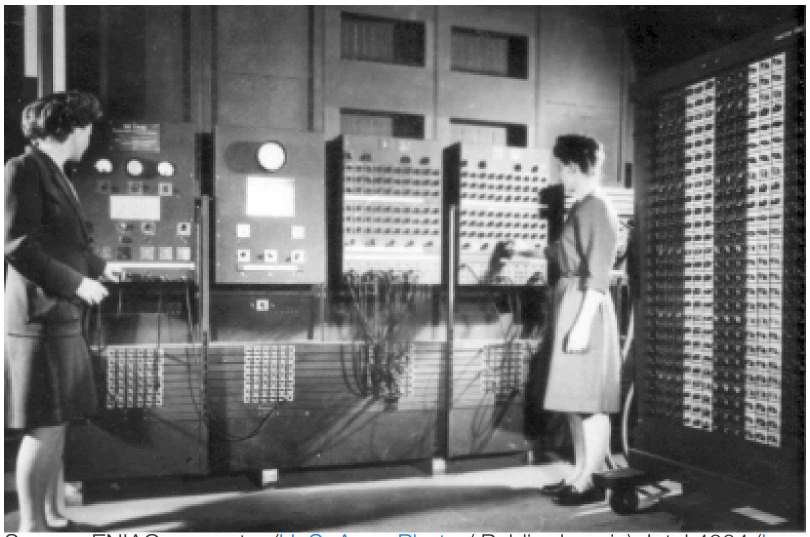
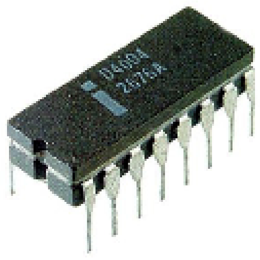
Chapter 1 - Introduction

Section 1.1 - What is an embedded system?

Early computers of the 1940s and 1950s occupied entire rooms. The 1960s and 1970s saw computers shrink to the size of bookcases. Shrinking led to the first single-chip computer processor, Intel's 4-bit 4004 in 1971, known as a ***microprocessor*** ("micro" meaning small). Shrinking through the 1970s and 1980s brought about the era of personal computers. The shrinking also enabled microprocessors to be embedded into other electrical devices, such as into clothes washing machines, microwave ovens, and cash registers; in fact, the first microprocessor was designed for a printing calculator. In the 1990s, such systems became known as embedded systems.

Figure 1.1.1: Examples of early computers and miroprocessors.





Source: ENIAC computer (U. S. Army Photo / Public domain), Intel 4004 (LucaDetomi at it.wikipedia (Transfered from it.wikipedia)
/ GFDL or CC-BY-SA-3.0 via Wikimedia Commons)

An ***embedded system*** is a computer system embedded in another device. The embedded system usually performs just one or a few dedicated functions. Embedded systems are often designed under stringent power, performance, size, time, and cost 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 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)

Figure 1.1.2: Examples of embedded systems.

Source: zyBooks

P Participation Activity

1.1.1: Embedded systems.

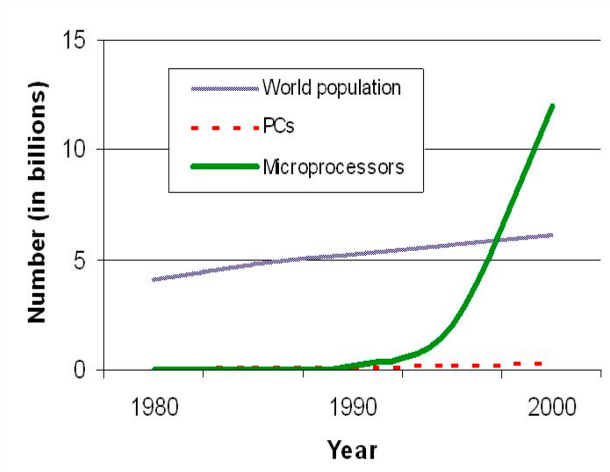
Indicate which are embedded systems.

#	Question	Your answer
1	Desktop PC	True False
2	Electronic drums (musical instrument)	True False
3	Elevator (lift)	True False
4	Cloud server	True False
5	Cardiac pacemaker	True False

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

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) becomes 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).

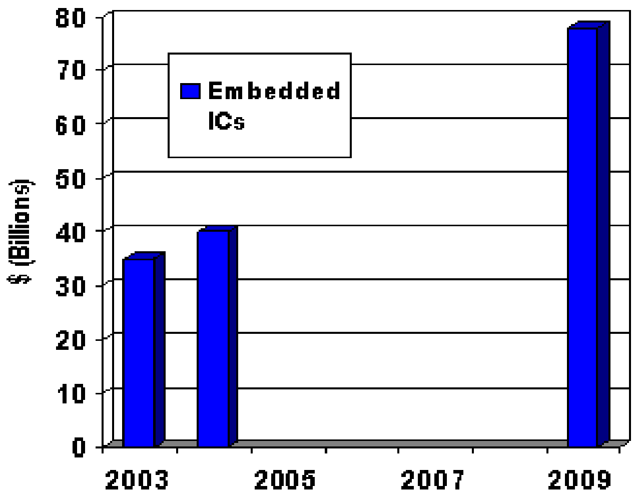
Figure 1.1.3: Microprocessors per year.



Source: Study of Worldwide trends and RD programmes in Embedded Systems by FAST GmbH.

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

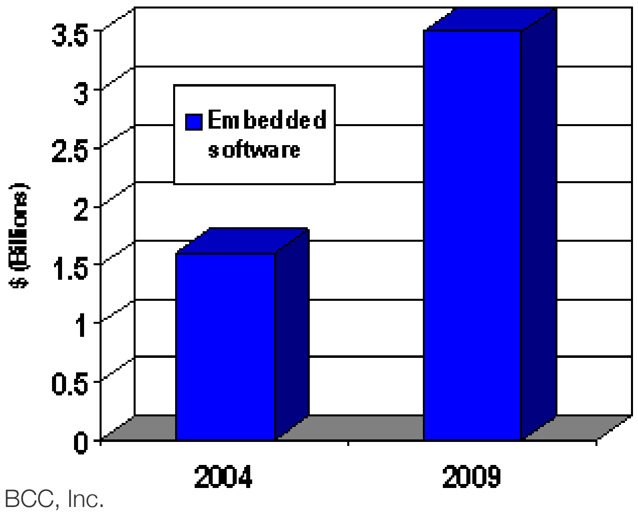
Figure 1.1.4: Sales of embedded ICs.



Source: BCC, Inc.

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

Figure 1.1.5: Sales of embedded software.



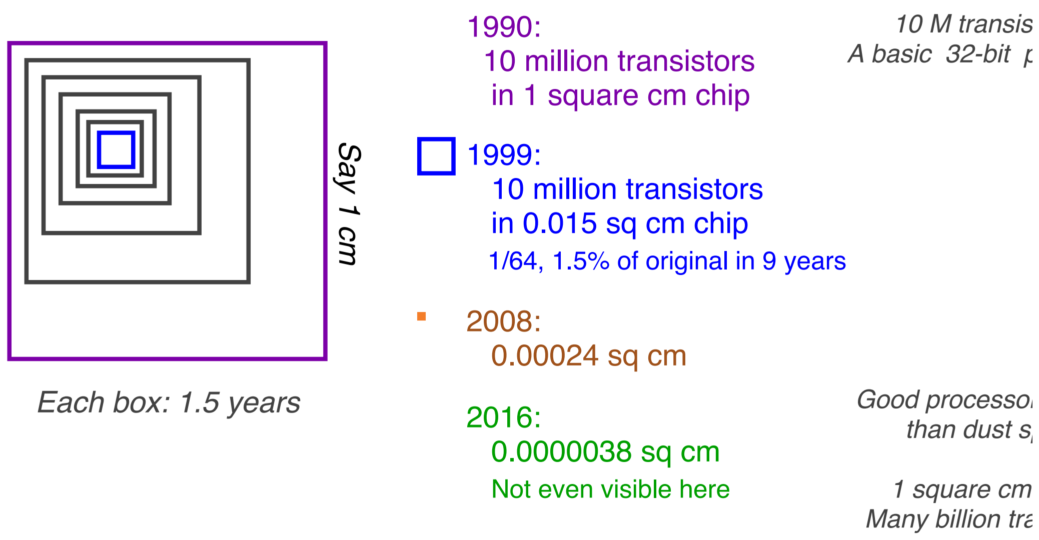
Source: BCC, Inc.

Gross sales of embedded software is also growing rapidly.

P Participation Activity

1.1.2: Moore's Law.

Start



Try 1.1.1: Origami.

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 1.1.2: Inventions.

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

P Participation Activity

1.1.3: Moore's Law.

#	Question	Your answer
1	Moore's Law suggests IC capacity doubles about every ___ months.	
2	Consider a chip in a smartphone today. If Moore's Law holds, that chip in 6 years would hold how many times more transistors?	
3	Consider a chip in a smartphone today. If Moore's Law holds, the transistors within that chip in 12 years would occupy what fraction of the chip's current size? Write answer as a fraction in the form: 1 / 4	

Exploring further:

- Microprocessors from Wikipedia.
- Intel's 4004 microprocessor
- A tiny sampling of embedded systems from Wikipedia:
    - Leap Motion Controller
    - Samsung Galaxy Gear
    - Oculus Rift
    - Kinect
    - Google Glasses
    - Lego Mindstorms
    - Roomba
    - Collision avoidance system

Section 1.2 - 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's changing value over time is called a digital ***signal***. "Digital" refers to the signal having discrete rather than continuous possible values.

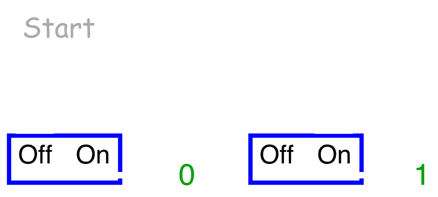
Switch and push button

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.

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.

P Participation Activity

1.2.1: Abstraction of a switch.

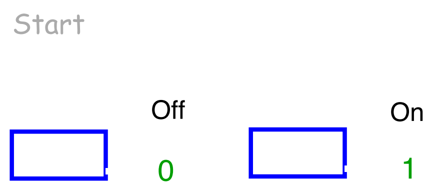


A push button operates similar to a simple switch, having a pair of electrical contacts and two mechanically controlled states: closed or open. Unlike a simple switch, the push button enters a closed state when being *pressed*. The moment the pressing force is removed, the push button reverts to, and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

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.

P Participation Activity

1.2.2: Abstraction of a button.

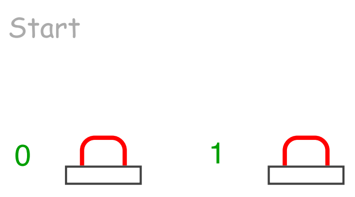


LED

A ***light emitting diode*** or ***LED*** is a semiconductor with a pair of contacts. When a small electrical current is applied to the LED contacts, the LED illuminates.

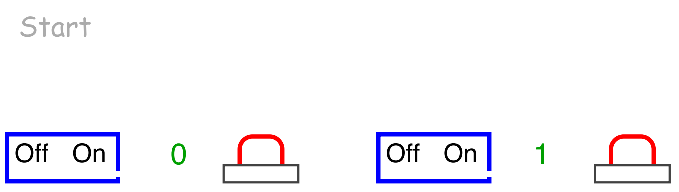
An abstraction of an LED is a component with a single bit input that can be either 0 or 1. When the input is 0, the LED does not illuminate. When the input is 1, the LED illuminates.

P Participation Activity | 1.2.3: Abstraction of an LED.



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

P Participation Activity | 1.2.4: Switch/LED system.



This system falls short of being an embedded system because it lacks computing functionality. For example, the system can't be easily modified to illuminate the LED when either of two switches are set to on. A component that can execute 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.

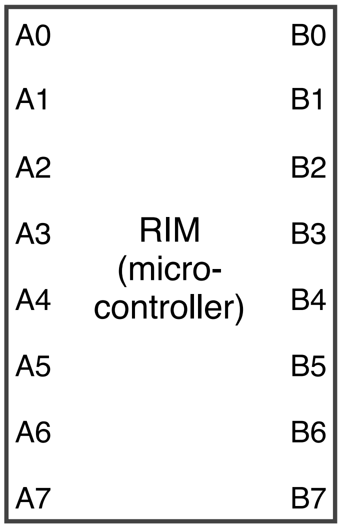
Figure 1.2.1: A "PIC" microcontroller.

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. RIM is able to execute C code that can access those inputs and outputs as implicit global variables (this material assumes reader proficiency with C programming).

Set inputs A2, A1, and A0 below to enable the output B0.

P Participation Activity | 1.2.5: The RIM executes C code.

Run program

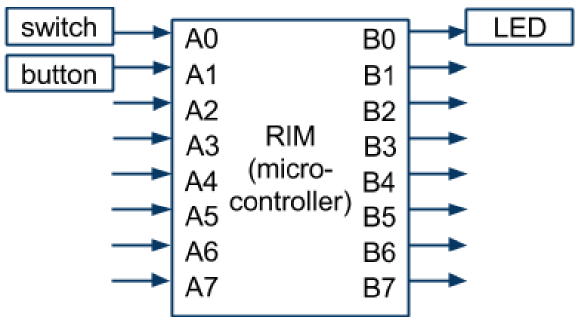


```
#include "RIMS.h"

int main(void) {
   while (1) {  // Repeat fo
      B0 = A2 && A1 && A0;
   }

   return 0;  // Never reach
}
```

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.

Figure 1.2.2: The RIM connected to switches, buttons, and LEDs.



P Participation Activity

1.2.6: Basic components.

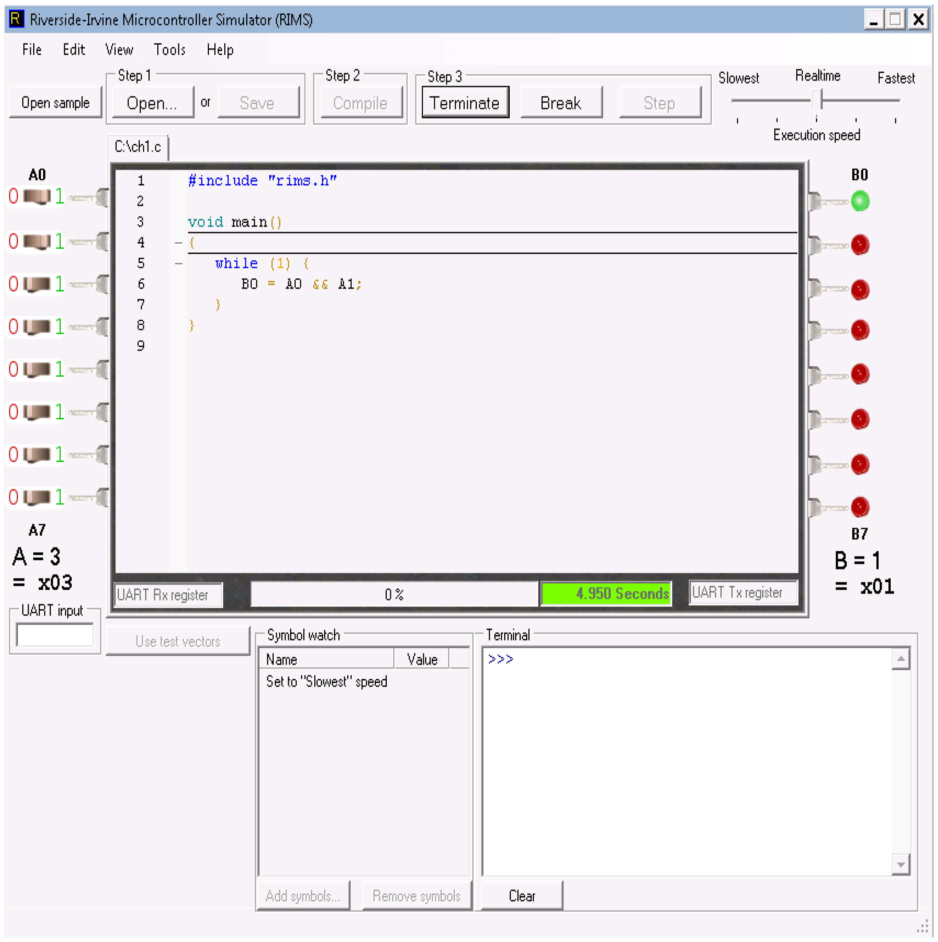
#	Question	Your answer
1	A button outputs 1 Volt when pressed, 0 Volts otherwise.	True
		False
2	An LED's electronics are designed to turn on when the input voltage exceeds 1 Volt.	True
		False
3	A microcontroller performs computations that convert input values to output values.	True
		False
4	A microcontroller cannot be programmed.	True
		False

Exploring further:

- Wikipedia: Microcontroller
- Wikipedia: Atmel AVR
- Wikipedia: C language
- Wikipedia: Sensor
- Wikipedia: Actuator
- Wikipedia: Digital signal
- Wikipedia: Switch
- Wikipedia: Push_button

Section 1.3 - RIMS

Figure 1.3.1: A screenshot of RIMS.



```
#include "rims.h"

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

A = 3
= x03

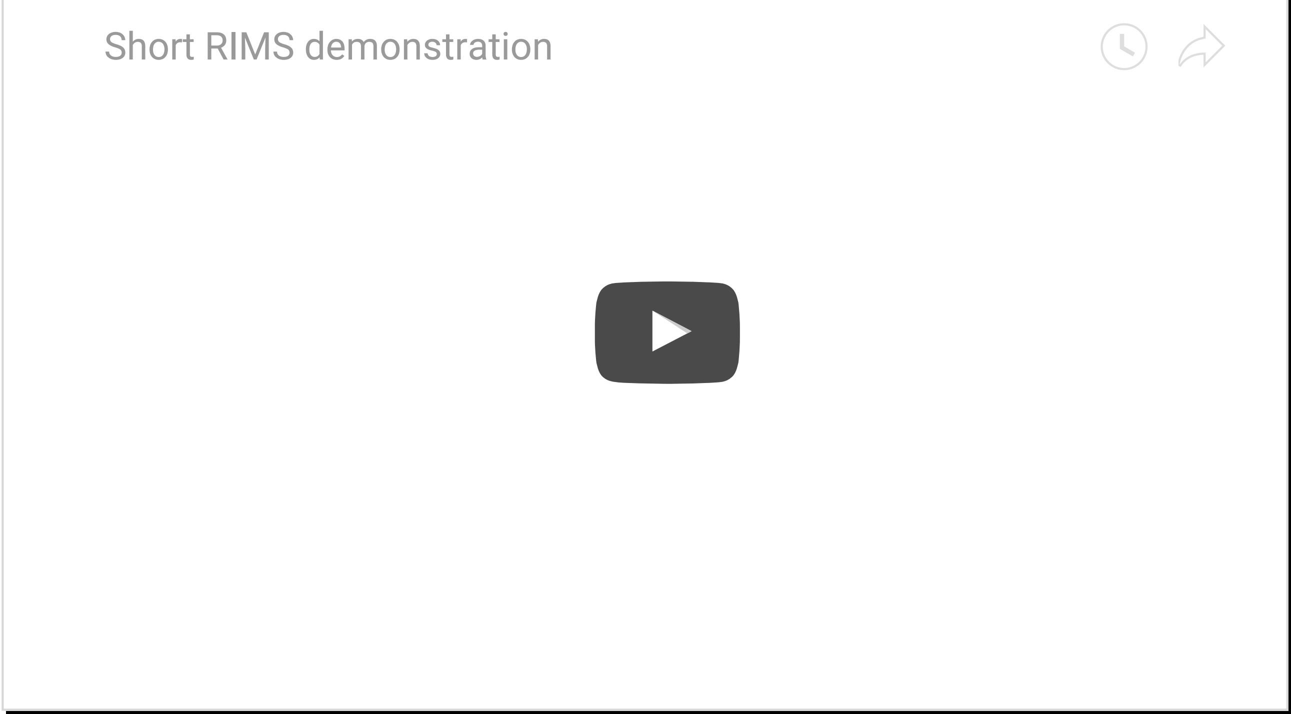
B = 1
= x01

Set to "Slowest" speed

```
>>>
```

RIMS (RIM simulator) is a graphical PC-based tool that supports C programming and simulated execution of RIM. RIMS is useful for learning to program embedded systems. A screenshot of RIMS is shown in the following figure. 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.

Video 1.3.1: RIMS video demonstration.



C code can be written in the center text box. The line *#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 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".

The user can right-click on a switch to convert the switch to a button. The user can right-click on an LED to change the LED to a speaker.

Try 1.3.1: Download and use RIMS.

Download, install, and execute RIMS (see http://ritools.cs.ucr.edu). 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 make them both 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 1.3.2: RIMS two-or-more program.

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 (||) in addition to logical AND. Run the program in RIMS to test the program.

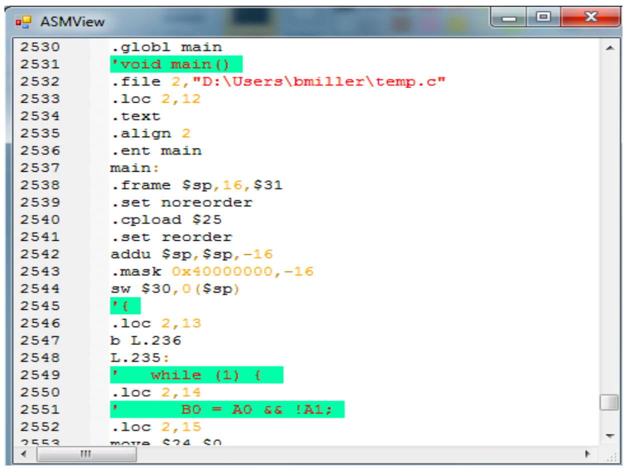
Pressing "Break" temporarily stops the running (and the button changes to "Continue") and shows an arrow next to the current C statement. Each press of "Step" then executes one C statement. Pressing "Continue" resumes running. Pressing "Terminate" ends the program, and re-enables editing of the C code.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

Try 1.3.3: Stepping in RIMS.

For the following program, 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.

```
#include "RIMS.h"

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

Figure 1.3.2: ASMView.



```
.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
move $24,$0
```

During the compilation process, the C code is translated to a low-level assembly language known as 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, then 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 specific input, output, or global variables in the C code.

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

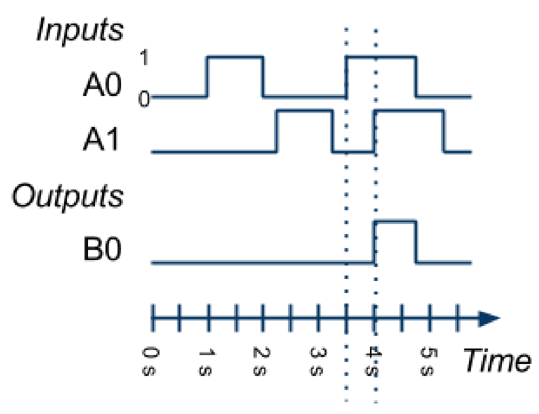
Exploring further:

- Wikipedia:MIPS

Section 1.4 - 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*** 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 example1.c program that continually computes B0 = A1 && A0. A0 is 0 from time 0 ms to 1 ms, when its value changes to 1. A0 stays 1 until 2 ms, when its value changes to 0. And so on. The 1 and 0 values are labeled for signal A0, but are usually implicit as for A1.

Figure 1.4.1: Two-input one-output timing diagram.



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 1.4.1: Timing diagram showing all combinations of three inputs.

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 1.4.2: Expected timing diagram for a program.

A program should set B0 to 1 if exactly one of A0, A1 is 1. Draw a timing diagram illustrating this behavior. Start by drawing waveforms for A0 and A1 that cover all four possible value combinations. Then draw the waveform for B0.

A change on a signal is called an ***event***. Events typically refer to a one-bit signal changing from 0 to 1 or from 1 to 0, but can refer to changes on an integer (multi-bit) or other signal. In the above timing diagram, A0's signal has four events, A1's has four, and B0's has two. If a signal changes from 0 to 1, the event is called ***rising***. 1 to 0 is called ***falling***. A ***pulse*** is a signal portion started by a rising event and ended by a falling event (looks like a camel hump). Above, A0's signal has two pulses, A1's has two, and B0's has one.

P Participation Activity

1.4.1: Timing diagram basics.

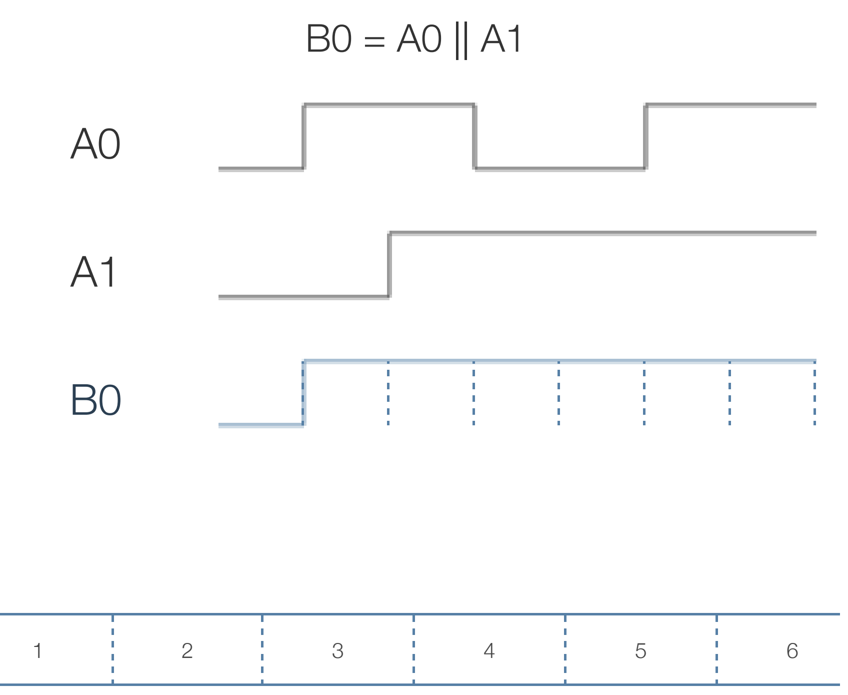
Questions refer to the above timing diagram.

#	Question	Your answer
1	A0 is 1 at time 1.5 s.	True False
2	A0 and A1 are both 1 for just over 0.5 s.	True False
3	The vertical dotted line indicates that B0 changing to 1 causes A1 to change to 1.	True False
4	B0 exhibits three events: 0, 1, and 0.	True False
5	A0 exhibits five pulses: 0, then 1, 0, 1, and finally 0 again.	True False

P Participation Activity

1.4.2: Set output B0 to match the equation and given inputs.

Start

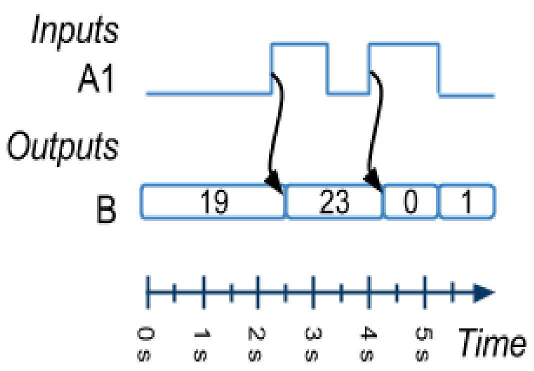


Check Next

Timing diagrams display numerical values by writing the value within a rectangle, as shown below where B's value is 19 for the first 2.5 seconds, then 23, then 0.

Also, timing diagrams may include an arrow to indicate that a particular input event ***triggered*** or caused some output event, as shown below where each rise of A1 triggered a change on output B.

Figure 1.4.2: A timing diagram with an integer output, and with indicated triggering events.



P Participation Activity

1.4.3: Timing diagrams with integers and also triggering events.

Questions refer to the above timing diagram.

#	Question	Your answer
1	The timing diagram indicates that shortly after time 2 s a rising event on A1 caused B to change from 19 to 23.	True False
2	The timing diagram indicates that at around time 4 s a change on B from 23 to 0 triggered a rising event on A1.	True False
3	The timing diagram indicates that just after time 5 s the falling event on A1 caused B to change from 0 to 1.	True False
4	For the time during which B is 0 or 1, the timing diagram could have shown the signal as high or low, as for A1.	True False

Exploring further:

- Wikipedia: Digital timing diagram

Section 1.5 - 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 be set to 00, then 01, then 10, then 11. B0 should only output 1 in the second case where A1A0 is 01, which causes B0's LED to be green),.

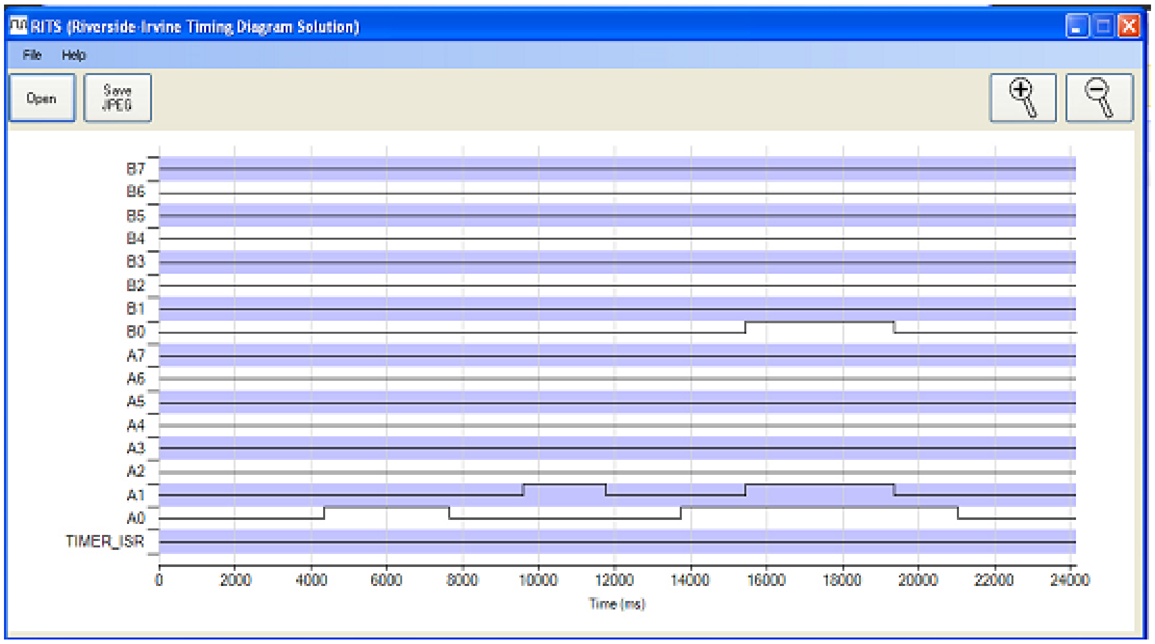
For most code, too many possible input combinations exist to test all those combinations. For example, completely testing `B0 = A0 && A1 && A2 && A3 && A4 && A5 && A6 && A7` would require 256 unique input value combinations. A system with 32 1-bit inputs would have over 4 billion input combinations. And when the system contains internal states (covered in later sections) meaning sequences of inputs matter, the number of possible input sequences is huge.

Because testing usually can't cover all input combinations, testing should cover ***border cases***, which are fringe or extreme cases such as all inputs being 0s and all inputs being 1s, and then various normal cases. For the above example, testing might test two borders, A7..A0 set to 00000000 (output should be 0) and to 11111111 (output should be 1), and then perhaps a dozen 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***.

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).

Figure 1.5.1: RITS timing diagram.



The above shows a RITS timing diagram for a program that executes: `B0 = A0 && A1`. 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 file*** (value change dump). 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.

Try 1.5.1: Generating a timing diagram.

Save, compile, and run the below program. 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.

```
#include "RIMS.h"

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

If a program fails tests, a good first step is for the programmer to manually trace the program. To ***manually trace*** a program means to mentally execute the program, perhaps with the aid of paper and pencil. In doing so, the programmer may detect the error.

A helpful debugging practice is the use of trace statements to track program execution. A ***trace statement*** is a statement that prints information about a program's execution, such as what region of code is currently executing, or such as the value of a variable at some point in the program. For example, a message `Entering function1()` could identify a currently-executing subroutine. RIMS supports the standard C printf() function for trace statements. The basic method for printing a trace statement is to use `printf("Trace text here.\r\n");`. printf() supports arguments for printing the values of different data types, like char, int, etc. Printed items appear in the terminal output window of RIMS. A programmer might temporarily insert print statements during debugging, comment those statements out when not currently debugging, and eventually delete those statements. Great care should be taken to avoid introducing new bugs when adding trace statements.

Try 1.5.2: Using printf.

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

Try 1.5.3: Seat belt warning system.

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.

Test vectors

Input value combinations, known as ***test vectors***, can be described in RIMS rather than each input value combination being generated by clicking on switches. Clicking the "Use test vectors" button towards the bottom-left corner of RIMS activates the test-vector panel. The user can describe a sequence of events to perform in order. The possible events include:

- Set input – set the inputs A0-A7 to a specific value.
  - b00000010
  - 0x02
- Wait – wait for a specific amount of time.
  - wait 15 ms
  - wait 3 s
- Check output – Assert that the output B matches a specific value.
  - assert b11110000
  - assert 0xf0

The check output event is also known as an ***assertion***. An assertion compares the B7-B0 outputs to the given expected value, and prints a warning if those values do not match. Assertion statements provide a mechanism for detecting when a program is not behaving as intended.

All values can be specified in binary (b00000111), hexadecimal (0x07), or decimal (7) format.

Below is a simple set of test vectors that can be used to test a program that executes the statement `B0 = A0 && !A1`. The first test vector sets inputs A1A0 to 00, waits 100 ms, and checks that B0 is 0. The second test vector then sets A1A0 to 01 and asserts that B0 is 1.

Figure 1.5.2: A simple test vector.

```
b00000000
wait 100 ms
assert b00000000
b00000001
wait 100 ms
assert b00000001
```

Try 1.5.4: Update the test vector to test each value of A1A0.

Use RIMS to test the `B0 = A0 && !A1` program. Copy the test vector from above, then add events to check for correct behavior when A1A0 is 10 or 11.

P Participation Activity

1.5.1: Testing.

#	Question	Your answer
1	A minimum testing requirement is to test all possible input combinations of a system.	True False
2	A test vector is a particular combination of input values.	True False
3	A border case is a typical input value to a program.	True False
4	A trace statement prints information about an executing program.	True False

Exploring further:

- Wikipedia: Software Testing
- Wikipedia: Software Debugging
- Wikipedia: printf
- Wikipedia: Value Change Dump file