Computerized Train Control System

Team Members:
Dan Allen
Brad Bomer
Nick Forck

Advisors:
Dr. Malinowski
Dr. Schertz

05/14/04
Abstract:

The 2004 webtrain (computerized control of a model train) project is based on a complex combination of hardware and software. The main hardware components include a Xilinx XS-4005E FPGA development board, an 80535 EMAC development board, a server PC, and N-Scale train engines and track layout. The software languages used to program these hardware pieces are VHDL, Assembly, and Java, respectively. This project focused on programming the above mentioned hardware to provide a basic hardware and software structure that allows Internet control of N-Scale model trains. An additional project objective was to make the hardware and software structure easily comprehensible and easily expandable for future projects.
# Table of Contents

Abstract......................................................................................................................................ii

Introduction................................................................................................................................1

System Description....................................................................................................................1

Block Diagrams ........................................................................................................................2
  Overall Block Diagram...........................................................................................................2
  Motherboard Block Diagram.................................................................................................3
  Current System Block Diagram............................................................................................4

Circuit Diagram .......................................................................................................................5

Program Flowcharts ................................................................................................................6
  Assembly Flowcharts...........................................................................................................6
  VHDL Flowcharts................................................................................................................7

Analysis of Results ..................................................................................................................11

Conclusion ...............................................................................................................................11

Appendix A (Standards) .........................................................................................................13

Appendix B (Assembly Code) ...............................................................................................15

Appendix C (VHDL Code) ....................................................................................................18
Introduction:

Model railroad sets have provided entertainment to many interested individuals for decades. The trains continue to become more and more complex due to integrated circuitry that allows precise control over multiple trains on the same track. With the increase in complexity of the trains comes a drastic increase in price of the models. This pricing increase leads to the problem that few people are actually able to afford elaborate train systems. For this reason, an Internet based model train control system was designed to allow model train owners to showcase their setup on the Internet. With model train owners placing their setup on the Internet, all individuals having access to the Internet will have full control of model trains.

System Description:

The following information describes the various goals and objectives that were set and changed throughout the project:

*Proposed Project Status (2003 Completion):*

The following objectives were completed by the previous team working on this project:

- Serial Communication from PC to EMAC board.
- Parallel Communication from EMAC 80535 board to Xilinx board.
- Xilinx board control to send DCC signals to trains.
- PC server with java applet to allow control of trains.
- Java applet functionality to allow viewing of trains.
- Parallel output from EMAC board to allow control of track switches.

*Original 2004 Project Primary Objectives:*

The following are the objectives to be implemented during the course of this project:

- Load working code to regain functionality of the system at a level equivalent to the previous team’s end product.
- Port assembly language from EMAC 80535 board to 8031 on the Xilinx XS40 development board.
- Design a motherboard with integrated digital I/O ports and Xilinx daughterboard.
- Design a logarithmic amplifier to sense position of trains on track.
- Develop remote track switching capabilities for train path of travel control.

*Original 2004 Project Secondary Objective:*

- Design optical sensors for decoupling control.

After starting the project, it was very apparent that the proposed project status from 2003 was not accurate by any means. The reason for the inaccurate status was due to a power failure over the summer of 2003 that caused the loss of working code for the FPGA and 80535 microcontroller. Furthermore, the code that did remain resembled the functioning code, but was too convoluted to follow. The code was not adequately commented to be understood.

The 2003 Project Status and 2004 Project Objectives were modified due to reasons described above.
**Modified Project Status (2003 Completion):**
The following information describes the actual status of the beginning of the 2004 project:

- PC server with Java applet to allow control of trains (send packet information over PC serial port).
- Java applet functionality to allow viewing of trains.
- Non-functioning 8051 assembly code to forward packet information from the server to the Xilinx XS40 development board.
- Non-functioning VHDL to convert the digital formatted packet into DCC and transmit the signal to the H-bridge.

**Modified 2004 Project Objectives:**
The following goals are the adjusted objectives of the 2004 webtrain project:

- Modify the Java code for integrating the new server PC.
- Modify the Java code for integrating the new webcams.
- Rewrite 8051 assembly code for acquiring data from the server PC over the serial port.
- Rewrite 8051 assembly code to transmit acquired data to Xilinx board over custom designed parallel link.
- Rewrite VHDL for acquiring digital packet information from 80535 microcontroller over custom designed parallel link.
- Rewrite VHDL to convert digital packet information to DCC signal format and transmit serially to H-Bridge.
- Continue with any original 2004 project objectives if time permits.

Refer to Appendix A on page 13 for information regarding the standards followed and a thorough explanation of the DCC format.

**Block Diagrams:**
The desired system (completion of original 2004 project objectives) utilizes several hardware components. Figure 2-1 below shows the block diagram of the entire system that would result from completion of the original 2004 project objectives.

**Figure 2-1:** Overall block diagram of train control system.
microcontroller and the FPGA. Finally, the remaining lines from the FPGA carry the bitstream to the train in a serial method over one line to the H-Bridge, and over two lines from the H-Bridge through the commercial amplifier to the tracks.

Figure 2-1 shows an overall block diagram, but it does not provide much detail in certain areas. Figure 3-1 below shows the motherboard configuration in more detail.

**Figure 3-1:** Block diagram of the motherboard configuration proposed.

The motherboard shown above is designed to perform the majority of the work involved in making the train control system function properly. The server PC forwards four bytes of data to the microcontroller. The description of each of the four bytes is as follows:

- The first byte is a data classification byte to determine what the next three bytes signify.
- The second byte is an address byte that targets a particular train.
- The third byte is a command byte, which contains control information for the targeted train.
- The fourth byte is a parity or error detection byte (Exclusive OR of address and command bytes) that is used to inform the train if errors have occurred during data transmission.

The 8031 microcontroller (located on the Xilinx XS40 development board) is utilized to forward the train control information from the server to the FPGA. It is also needed to supply various other services that have yet to be implemented. Such services include the track switching and current sensing that can be seen in Figure 3-1.

Again, the method of output from the FPGA to the H-Bridge is a single wire that is toggled high or low. The H-Bridge has two output lines toggling between +12 V and 0 V based on whether the input line is high or low. Each line carries a signal that is the polar opposite of the other. In other words, when one line is at +12 V, the other line is at 0 V. The reason for this polarity switching is because the train uses one line to acquire the data and then rectifies the two lines for
+12 VDC for power. Lastly, the two lines from the H-Bridge run through the commercial amplifier and then to the tracks (the commercial amplifier is used to make sure the signals reaching the tracks are not able to damage the expensive trains).

Unfortunately, the aforementioned information is only design based and not fully implemented. Figure 4-1 below shows the current block diagram of the hardware layout.

**Figure 4-1:** Actual System block diagram as of 2004 project completion.

Currently, the 8031 microcontroller on the Xilinx development board is not being utilized for the microcontroller needs. Instead, the EMAC 80535 development board was chosen due to the default serial communication options. The method of communication between the microcontroller and FPGA remains the same.

Track switching and current sensing have not been implemented due to the change in project objectives described earlier.
Circuit Diagram:

The circuit diagram is similar to the block diagram, but the different communication methods are broken down into the various wires connected between the hardware components. Figure 5-1 below shows the circuit diagram for the functioning hardware system.

**Figure 5-1:** The entire circuit diagram for the current webtrain system.

The Internet is used for the two-way communication between the client and server. The server transmits the control data to the EMAC development board over the serial port. The EMAC sends the data to the FPGA over a custom 16 bit communication bus in three clock cycles, one byte at a time. The two extra lines from the EMAC are used to regulate the communication. The output methods from the FPGA were already discussed earlier with the block diagrams.
Flowcharts:

Figure 6-1 shows the assembly language flowchart that is currently in use.

**Figure 6-1:** Assembly language flowchart.

The assembly language flowchart shows the currently implemented feature of train control packet forwarding, and the flowchart also shows two features that were original objectives (current sensing and track switching). At the moment, the EMAC is only utilized to forward the three bytes necessary to control the train. However, future projects will supply code for having the EMAC provide switch control and current sensing. Figure 7-1 contains a detailed flowchart of the functioning assembly code.
**Figure 7-1:** Detailed flowchart of functioning assembly code (Refer to Appendix B on page 15 for the actual assembly code).

The above flowchart shows the steps that the microcontroller goes through to forward the train control information from the server to the FPGA. The operation is as follows:

1. The EMAC waits in a loop for input from the server via serial port.
2. The received data is placed in a queue until all four bytes of data are received.
3. The microcontroller is coded to skip the first byte, and transmit bytes two, three, and four as address, command, and parity bytes to the FPGA.
4. The definitive address (address to specify to the FPGA which byte is being broadcast) is placed on port 2 (the upper address bus).
5. The corresponding data is placed in the accumulator.
6. The contents of accumulator are transferred to the FPGA using the MOVX command (the accumulator is broadcast over the data bus).
7. The broadcasting steps are repeated until all three bytes are sent (return to step 4).
8. The system returns to the main loop (step 1).
The flowcharts for the VHDL code are considerably more complex than the assembly language flowcharts. The extra complexity comes from the fact that the VHDL code must perform two completely separate tasks simultaneously. Figure 8-1 below shows the entire VHDL flowchart. Since the flowchart is difficult to read, it has been broken into two independent sections (marked off by the boxes) and shown in Figures 9-1 and 11-1.

**Figure 8-1:** Entire VHDL flowchart (Refer to Appendix C on page 18 for the actual VHDL code).
The first section discussed is the data acquisition section that is responsible for receiving and storing the train control data from the EMAC. Figure 9-1 shows a slightly more readable view of the acquisition process.

**Figure 9-1: Data acquisition process written in VHDL**

Currently, the VHDL has been written according to specification, and functions properly. The method of communication established between the EMAC and FPGA is setup to take place in
three clock cycles from the microcontroller. When the falling edge of WR* occurs, the FPGA checks to see that EXTIO* is low. If EXTIO* is low, then the FPGA accepts what is present on the data bus as the specified byte according to what is transmitted across the upper address bus as described below. Each of the three bytes is stored in memory and referenced every time the FPGA needs to output the signal to the track.

The EMAC uses external I/O addressing to communicate with the FPGA. The FPGA is programmed to respond to the consecutive addresses of 70h, 71h, and 72h. Therefore, when the EMAC writes to external address 70h, the data lines contain the 8-bit address of a valid train. The EMAC will write to the following two consecutive addresses (71h and 72h) with the command and parity bytes, respectively. The FPGA associates all three bytes (address, command, and parity) and stores them so that the DCC bitstream can be formulated and transmitted to the tracks.

When the FPGA receives an indicated address byte (70h over address bus and address byte on data bus), the FPGA quickly determines what to do with the command and parity bytes to follow by setting a dummy variable (train_pick – as can be seen in the VHDL code in Appendix C). For example, when the received address is 00h, the FPGA knows that a broadcast address has been received that is mandating the stopping of all trains on the track. Therefore, all address bytes that are stored on the FPGA are set to 00h, and the command and parity bytes to follow apply to all stored train bitstreams.

Presently, the VHDL software only supports the use of two trains, so the data acquisition process only expects to receive two addresses other than 00h. Upon the reception of an address other than 00h (01h and 03h are programmed to be accepted), the FPGA will set the address bytes in both stored train bitstreams and set a dummy variable to identify which train bitstream is to be affected by the following command and parity bytes.

The reason that both addresses are set when the first address is received is to prevent an address of 00h from reaching the rails. If one train address is set and being broadcast while the other broadcast address is still set to 00h, the moving train will recognize the 00h address as a general broadcast address and respond to the following command and parity bytes. A “stop-go-stop-go” situation will occur, which causes poor performance of the trains.

The data conversion and output process occurs constantly. The flowchart for the process is shown in Figure 11-1. First, a preamble is broadcast over the track to inform the trains and a control bitstream is being output to the tracks. The preamble consists of ten to twelve consecutive 1s being output. After the preamble, a zero separator bit is output. The control bytes are then converted to DCC one bit at a time with a zero separator bit inserted between each byte until the final byte. After the final byte is output, a 1 ending bit is inserted to signify the end of the bitstream. The next train bitstream is then output to the tracks in the same method just described.
Figure 11-1: Data conversion and output process for VHDL code
Analysis of Results:

The original goals of the project as outlined at the beginning of the project were not met. Due to circumstances with respect to the ending status of the project from the 2003 group, the 2004 project goals were adjusted to reflect a more physically possible set of tasks. In other words, since the 2003 project no longer operated properly, the main goals of the 2004 project were set to regain functionality of the system and prevent another loss of functioning code while improving the ability for other people to understand and improve upon the code structures.

The project has encountered success in producing a control system for an n-scale model train. Currently, the communication flow from client to track is exactly as described previously. The client is able to communicate successfully with the server, and the server through the serial port to the EMAC. The FPGA is also able to receive information from the EMAC, and then convert and send the information to the track.

Simple control of two trains exists over the Bradley electrical engineering network. The track can be viewed from over the Internet, but specific control is unavailable over the general Internet due to the University’s networking issues as of late.

The other small goals of installing a new server PC and webcams were accomplished. The new server only required minor changes to the Java code. The USB 2.0 and FireWire cameras were installed and evaluated for a performance comparison. The FireWire camera consistently showed less CPU usage with equivalent image quality in comparison to the USB 2.0 camera. For this reason, the FireWire camera will be used to broadcast the track image over the Internet.

Conclusion:

Since the objectives for the 2004 webtrain project were dramatically changed during the year, the initial project objectives needed to be revised to reflect a more relevant starting point. Since there was very little basis left from the 2003 project, the objective of the 2004 project necessarily became to re-establish a working code basis for the web controlled model train. Consequently, the result of the 2004 webtrain project is an easily understandable and expandable code base for future webtrain projects.
Appendix A

Standards:

The two most important standards used in this project are the N-Scale model train standard and the Digital Command Control (DCC). The N-Scale standard is simply an engine and track size standard. All N-Scale engines and cars are based on the ratio 1:160. This means that every 1 foot of full size train equates to 1.9 mm of N-Scale train. N-Scale utilizes a 9 mm-gauge track.

The most important standard used for this project is the DCC standard. A complete description of the DCC standard can be found on the National Model Railroad Association (NMRA) website located at www.nmra.org/standards.

Digital Command Control is accomplished by sending a specified bitstream to the N-Scale tracks. Once the signal has been transmitted to the tracks, it is the job of the decoder inside the engine to determine what action (if any) to take. The bitstream that is to be transmitted to the track has a specific configuration and will hereby be referred to as a packet.

The method of transmission for individual bits of information is based on timing not level as it is in TTL. The DCC standard dictates that a logical 1 have a duty cycle of 58µs. Therefore a logical one has a total pulse width of 116µs (see Figure 12-1 below). The tolerance limit on a 1 bit is ± 3µs. Consequently any 1 bit must have a duty cycle within the range of 55µs to 61µs. A logical 0 bit has a more lax tolerance than 1 bits. A 0 bit can have a duty cycle between 95µs and 9900µs, but the total period must not exceed 1200µs. For this project, the FPGA code uses 116µs as the duty cycle of a 0 bit. This was a natural choice since it is simply twice the duty cycle of a 1 bit. This choice made some of the FPGA coding more simplistic.

Figure 12-1: Examples of a logical 1 and 0 transmitted in DCC

The packet structure for DCC is also very specific. The basic packet used to instruct engines is as follows: preamble, separator bit, train address byte, separator bit, command byte, separator bit, and error byte. Therefore a typical packet can be illustrated as {preamble}0{address}0{data}0{parity}1, where the preamble is ten or more bits and address, data, and parity are all 8-bit quantities.
The address byte mentioned above must be of the form 0AAAAAAA, where A represents the address of the train for which the packet is intended. Every DCC N-Scale train is required to be able to retain and recognize its own address. The data byte must have the form 01DCSSSS. In this byte D stands for train direction (‘1’ being forward), C can refer to the least significant speed bit or in older engines controls the headlight, and SSSS dictates the speed of the train. The parity byte is used for error detection and takes the form EEEEEEE, where E represents the bitwise XOR of the address and data bytes. If the engine computed XOR is not identical to the transmitted XOR the packet is ignored.

The DCC standard states that packets should be repeated as frequently as possible. For that reason, the FPGA code continually sends packets regardless of data change. In other words, even if no data changes, the FPGA sends another packet immediately following the previous packet’s completion.
Appendix B

EMAC Code:

;Daniel Allen
;This code is fully functional.
;In this file serial information is placed on a stack and then sent to the FPGA.

$nomod51
#include(reg515.inc)

;constants for the DUART chip

MRA equ 00h
SRA equ 01h
CSRA equ 01h
BRGTST equ 02h
CRA equ 02h
RHRA equ 03h
THRA equ 03h
ACR equ 04h
ISR equ 05h
IMR equ 05h

;Start of code

intr equ 8000h
org 8000h
ljmp gogo

;---------------------------------------------------------
; Interrupt Vector Table
; Area
;---------------------------------------------------------
cseg at intr+0BH ; 0BH=addr for Timer 0
reti

cseg at intr+13h ; External interrupt 1.
reti

cseg at intr+1BH ; Timer 1 interrupt.
reti

cseg at intr+23H ; Serial interrupt
reti

cseg at intr+2BH ; Timer 2
reti

cseg at intr+43H ; IADC interrupt.
reti

cseg at intr+4BH ; IEX2 interrupt.
reti

cseg at intr+53H ; IEX3 interrupt.
reti

cseg at intr+5BH ; IEX4 interrupt
reti

cseg at intr+63H ; IEX5 interrupt.
reti

cseg at intr+6BH ; IEX6 interrupt.
reti

gogo:
    mov sp,#60h    ;This section initializes each dataspace and the
    stack location.
    mov r5,#50h ;The base of the stack is r5
    mov r6,#50h     ;The current location of the stack is r6
    mov r7,#00h     ;The length of the stack is r7
    setb  eal     ;This line was added so that runtime mode could be
canceled from the connected PC.
        ;It hasn't helped yet.
    ;Initialize the serial DUART

serinit:
    mov  p2,#CRA    ;Selects the location to send the data. (These are
    all controls for the DUART)
    mov  p2,#CRA
    mov  A,#10110000b ;Moves the bits needed to send to the location.
    movx  @r0,a       ;Sends the information, here bank MR0 is selected.
    mov  p2,#MRA
    mov  p2,#MRA
    mov  A,#00110000b
    movx  @r0,a       
    mov  p2,#CRA
    mov  p2,#CRA
    mov  A,#00010000b
    movx  @r0,a       ;bank MR1 is selected.
    mov  p2,#MRA
    mov  p2,#MRA
    mov  A,#00010011b
    movx  @r0,a       
    mov  p2,#CSRA
    movx  @r0,a       ;sets transmitter and receiver speeds to 9600 baud.
    mov  p2,#ACR
    mov  p2,#ACR
    mov  A,#01110000b
    movx  @r0,a       ;external timer divided by 16.
    mov  p2,#CRA
    movx  @r0,a       ;Bank MR1 is selected.
    mov  A,#000000101b
    movx  @r0,a       ;Transmitter and Receiver are enabled.
        ;mov  40h,#19h   ;These lines were used for autoloading certain data in
        a previous test mode
        ;mov  41h,#01h   ;of the program
        ;mov  42h,#6Fh
        ;mov  43h,#6Eh

serin:
    mov  p2,#SRA     ;Point to the status register
    mov  r1,#40h    ;Ready a storage location for r1
serinloop:
    movx  a,@r0      ;mov data in from status register
    jnb acc.0,serinloop ;confirm that data is being sent (a fact which is
recorded on the status register)
    mov  p2,#RHRA   ;Point to the receiver hold register (A) which has the
serial input data
    movx  a,@r0      ;Get the data from the serial port.
    mov  b,r6
    mov  r0,b

mov @r0,a ;Store data in the dataspace
mov @r1,a ;I don't remember why this line is here.
inc r6
inc r7
inc r1
cjne r7,#04h,serin ;When the length is 4 all the bytes have been
sent and the program continues on to
output the data.

;mov r1,#40h ;Allows internal data storage (testing purposes)
output:
mov b,r5
mov r6,b ;Sets the stack pointer to point to the first stack
location.
mov r1,b ;Sets r1 to point to the first stack location
inc r1 ;Sets r1 to the second stack location.
mov a,@r1 ;Moves stack #2 (address) to acc.
mov p2,#70h ;Sets address to FPGA (address)
movx @r0,a ;Sends info to FPGA
inc r1 ;Sets r1 to third stack location
mov a,@r1 ;Moves stack #3 (command) to acc.
mov p2,#71h ;Sets address to FPGA (command)
movx @r0,a ;Sends info to FPGA
inc r1 ;Sets r1 to fourth stack location.
mov a,@r1 ;Moves stack #4 (parity) to acc.
mov p2,#72h ;Sets address to FPGA (parity)
movx @r0,a ;Sets address to FPGA
mov b,r5 ;Resets the stack pointer.
mov r6,b ;Resets the length of the stack
ljmp serin ;jumps back to the top
end
Appendix C

VHDL Code:

-- Bradley University Webtrain
-- Nick Forck and Brad Bomer
-- 04/14/04

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity memory is
  port(
    clk   : in  std_logic;    -- input of hardware clock for clock division
    extio     : in  std_logic;    -- another active low signal that is triggered when the train information is sent
    wr        : in  std_logic;    -- write bar from EMAC, indicates when data byte present (goes low)
    cmd   : in  std_logic_vector(7 downto 0);  -- Command byte which is either train address (0AAAAAAA), command (01DCSSSS), or error (EEEEEEEE) byte depending on addr
    addr   : in  std_logic_vector(7 downto 0);  -- Address byte which determines which byte is on cmd, explained during memup process
    reset     : in  std_logic;    -- resets the system...
    sig   : out std_logic    -- Output to H-Bridge
  );
end memory;

architecture train of memory is

  signal clk2       : std_logic;     -- modified internal clock (stepped down from hardware clock)
  signal cnt        : integer range 0 to 725;   -- used in dividing the clock signal down to something useful
  signal bitstate   : integer range 0 to 33;   -- controls the output location of the 24 bit bitstream
  signal tmp_train  : std_logic_vector(23 downto 0);  -- temporary 24 bit singal storing the currently output bitstream
  signal bit_cnt    : integer range -1 to 23;   -- determines which of the 24 bits is currently being output
  signal train_sel  : integer range 0 to 1;   -- determines which train bitstream is output
  signal train_0    : std_logic_vector(23 downto 0);  -- stores the bitstream for a train
  signal train_2    : std_logic_vector(23 downto 0);  -- stores another bitstream for a different train
  signal address0   : std_logic_vector(7 downto 0);  -- address byte for train 0
  signal command0   : std_logic_vector(7 downto 0);  -- command byte for train 0
  signal parity0    : std_logic_vector(7 downto 0);  -- error check byte for train 0
  signal address2   : std_logic_vector(7 downto 0);  -- address byte for train 2
end architecture;
signal command2 : std_logic_vector(7 downto 0); -- command byte for train 2
signal parity2 : std_logic_vector(7 downto 0); -- error check byte for train 2

constant addressa : std_logic_vector(7 downto 0) := x"70"; -- signifies that the address byte is being sent
constant addressc : std_logic_vector(7 downto 0) := x"71"; -- signifies that the command byte is being sent
constant addressp : std_logic_vector(7 downto 0) := x"72"; -- signifies that the parity byte is being sent

signal train_pick : integer range 0 to 2; -- determines which train bitstream is being updated; 0=both, 1=train0, 2=train2

begin

train_0 <= address0 & command0 & parity0; -- constant concatenation of the three bytes into the stored bitstream
train_2 <= address2 & command2 & parity0;

--set different train addresses, command and parity bytes here
--trains will be alternately addressed
train_select : process (train_sel) is
begin
  case train_sel is
    when 0 =>
      tmp_train <= train_0;
    when 1 =>
      tmp_train <= train_2;
    when others =>
      tmp_train <= x"ffffff";
  end case;
end process train_select;

--end clock process

memup : process (wr,reset) is
begin
if (reset = '0') then -- resets system
    when 0 present
    address0 <= "00000000";
    command0 <= "00000000";
    parity0 <= "00000000";
    address2 <= "00000000";
    command2 <= "00000000";
    parity2 <= "00000000";
    train_pick <= 0;
elsif (falling_edge(wr) and extio = '0') then
    -- When WR(bar) from EMAC goes low, data is being received
    if (addr = addressa) then -- receiving the constant "addressa" (refer to declarations above) signifies that the byte received from the command/data bus is a train address
        if (cmd = "00000000") then -- When the sent address is all zeros, both trains must immediately stop
            address0 <= cmd;
            address2 <= cmd; -- addresses for train0 and train2 are set to "00000000" since it is a broadcast command
            address0 <= cmd; -- the broadcast address will always affect all trains
            address2 <= cmd;
            train_pick <= 0; -- train_pick = 0 signifies that the next two bytes affect both trains
        elsif (cmd = "00000001") then -- the received address is for train0 only
            address0 <= cmd;
            address2 <= "00000011"; -- train2 address is also set to avoid sending out the broadcast address after the train0 info
            train_pick <= 1; -- the two following bytes are with respect to train0
        elsif (cmd = "00000011") then -- the received address is for train2 only
            address2 <= cmd;
            address0 <= "00000001"; -- train0 address is also set to avoid sending out the broadcast address after the train2 info
            train_pick <= 2; -- the two following bytes are with respect to train0
        end if;
    elsif (addr = addressc) then -- receiving the constant "addressc" signifies that the byte received from the command/data bus is a train command byte
        if (train_pick = 0) then
            command0 <= cmd; -- since train_pick was set to 0 above, the byte on the data bus applies to both trains' command byte
            command2 <= cmd;
        elsif (train_pick = 1) then
            command0 <= cmd; -- since train_pick was set to 1 above, the byte on the data bus applies to only train0
            command2 <= cmd; -- the only other option for train_pick is 2, which means the byte on the data bus applies to only train2
        else
            command2 <= cmd; -- the only other option for train_pick is 2, which means the byte on the data bus applies to only train2
        end if;
    elsif (addr = addressp) then -- receiving the constant "addressp" signifies that the byte received from the command/data bus is a train parity byte
        if (train_pick = 0) then
            parity0 <= cmd; -- since train_pick was set to 0 above, the byte on the data bus applies to both trains' parity byte
            parity2 <= cmd;
        elsif (train_pick = 1) then
            parity0 <= cmd; -- since train_pick was set to 1 above, the byte on the data bus applies to only train0
            parity2 <= cmd;
        else
            parity2 <= cmd; -- since train_pick was set to 2 above, the byte on the data bus applies to only train2
        end if;
train_pick <= 0; -- resetting train_pick to 0
else
    parity2 <= cmd; -- the only other option
end if;
end if;
end if;
end process memup;

--end of memory process

----------------------------------------------------- start of the encoding process to output the bitstream in DCC

bitstream : process (clk2,reset) is
begin
  --++--reset
  if (reset = '0') then
    bitstate <= 0;
    bit_cnt <= 23;
    sig <= '0';
  --++--bitstream
  elsif(clk2'event and clk2 = '0') then
    case bitstate is
      -- Preamble -- outputs 11 ones
      when 0 =>
        sig <= '1';
        bitstate <= 1;
      when 1|3|5|7|9|11|13|15|17|19|21 => -- the preamble requires 10 to 12 DCC ones, states 1-21 produce 11 ones
          sig <= '0';
          bitstate <= bitstate + 1;
        when 2|4|6|8|10|12|14|16|18|20 =>
          sig <= '1';
          bitstate <= bitstate + 1;
      -- End Preamble
      -- 0 separation bit
      when 22|23 =>
        sig <= '1';
        bitstate <= bitstate + 1;
      when 24|25 =>
        sig <= '0';
        bitstate <= bitstate + 1;
      -- End separation bit
      -- start address byte output
      when 26 =>
        sig <= '1';
        if (tmp_train(bit_cnt) = '0') then -- determines to skip the next two states if the current bit to output is a one
          bitstate <= 27;
        else
          bitstate <= 29;
        end if;
        bit_cnt <= bit_cnt - 1;
      when 27 =>
        sig <= '1';
        bitstate <= bitstate + 1;
      when 28 =>
        sig <= '0';
        bitstate <= bitstate + 1;
    end case;
  end if;
end process;
when 29 => -- determines if the entire
address or command byte has been completely output
    sig <= '0';
    if (bit_cnt = 15) then -- if bit_cnt = 15, then
        the entire address byte has been output and the command byte is next
        bitstate <= 22; -- will output a zero
        separator bit (state 22) and then start to output the command byte
    elsif (bit_cnt = 7 ) then -- if bit_cnt = 7, then
        the entire command byte has been output and the parity byte is next
        bitstate <= 22; -- will output a zero
        separator bit (state 22) and then start to output the parity byte
    elsif (bit_cnt = -1) then -- if bit_cnt = -1, then
        the entire parity byte has been output and the bitstream is complete
        bit_cnt <= 23; -- resets the bit count
        for the next run through the bitstream
        bitstate <= 30; -- since the parity byte is finished, jumps to state 30 to output the ending 1 bit
    else
        bitstate <= 26; -- if here, then no
        bytes have been completely output, reset to state 26 to output the next bit of
        the byte
        end if;
-- End of three byte outputs (address, command, and error check byte)
-- 1 bit end bit
when 30 => -- start of the final
    bit output of a 1
    sig <= '1';
    bitstate <= bitstate +1;
when 31 =>
    sig <= '0';
    bitstate <= 0; -- resets the
    bitstream state for the next bitstream
    -----------------------------
    if (train_sel = 0) then -- switches
        between the two stored
        train bitstreams
        train_sel <= 1;
    elsif train_sel = 1 then
        train_sel <= 0;
        end if;
    -----------------------------
-- end of bitstream
    when others =>
        end case;
    end if;
    end process bitstream;
end train;